инженерия программного обеспечения

Трифанов Виталий Юрьевич,
Цителов Дмитрий Игоревич

УДК 004.451.2

ДИНАМИЧЕСКИЕ СРЕДСТВА ОБНАРУЖЕНИЯ ГОНОК В ПАРАЛЛЕЛЬНЫХ ПРОГРАММАХ

Аннотация

Состояния гонки (data race) являются одними из самых трудновоспроизводимых и сложных в обнаружении ошибок многопоточного программирования. Они наступают, когда в параллельной программе происходит два несинхронизированных обращения к одному и тому же участку памяти, из которых одно является записью данных. Обычно гонки ведут к повреждению глобальных структур данных, а их «ручное» обнаружение сильно затруднено. Теме автоматизированного поиска гонок посвящено множество различных исследований, но она продолжает оставаться актуальной. Популярен динамический подход к обнаружению гонок, когда анализ и поиск гонок происходит прямо во время выполнения программы. В данной статье рассматривается эволюция существующих подходов к динамическому обнаружению гонок, анализируются их достоинства и ограничения. Отдельное внимание уделяется вопросу поиска гонок в Java-приложениях.

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

ВВЕДЕНИЕ

Состояние гонки (data race) возникает в многопоточной программе, когда различные потоки обращаются к одному участку памяти без промежуточной синхронизации и хотя бы одно из обращений является записью данных [31]. В подавляющем большинстве случаев наличие состояния гонки в программе свидетельствует об ошибке программирования. Такие ошибки очень тяжелы в обнаружении, поскольку они проявляются лишь при определённом чередовании операций в потоках. Кроме того, гонки зачастую повреждают глобальные структуры данных, но не приводят к немедленному отказу программы. Наконец, гонки очень слабо локализованы, и их последствия могут приводить к труднообъяснимым сбоям в различных частях программы.

Подходы к обнаружению гонок традиционно принято классифицировать по времени (относительно этапа выполнения программы) осуществления анализа. Выделяют следующие типы детекторов гонок:

– статические (*static, ahead-of-time*) – не требуют запуска программы, анализируют исходный код программы или скомпилированные файлы, не запуская их;

– *post-mortem* – собирают информацию во время выполнения программы, но анализируют её уже после завершения работы;

– динамические (*dynamic, on-the-fly*) – выполняются одновременно с программой, отслеживая синхронизационные события и обращения к разделяемым переменным.

Данная статья является продолжением работы [43], где мы привели обзор статических и post-mortem подходов к обнаружению гонок. В этой работе мы сосредоточились на обзоре динамических детекторов, по-прежнему делая акцент на обзоре инструментов для Java-приложений.

Динамические детекторы гонок анализируют программу во время её работы, собирая частичную трассу – историю обращений к памяти и выполнения операций синхронизации. Собираемая информация отражает реальный путь выполнения программы, поэтому такие детекторы анализируют только достижимые (возможные) пути. Для каждого конкретного запуска программы такие детекторы могут собрать максимально полную информацию и поэтому теоретически могут не допускать ложных срабатываний и находить только реально существующие гонки. В общем случае число всех возможных путей выполнения программы очень быстро растет с увеличением объема программы, а также количества потоков и переменных. Поэтому с помощью динамических детекторов невозможно доказать отсутствие гонок в программе. На практике анализируется лишь малая часть всех возможных путей, поэтому в крупных приложениях значительная часть гонок может остаться необнаруженной. Во многих системах (например, в операционных системах или драйверах) многие фрагменты кода выполняются лишь при очень редких условиях.

Алгоритмы, которые используются в динамических детекторах гонок, бывают трех типов [37]:

– *happens-before* – алгоритмы, опирающиеся на отслеживание порядка обращений различных потоков к общим участкам памяти (частичное отношение порядка на множестве событий называется happens-before); эти алгоритмы точны, но «чувствительны» к очерёдности выполнения инструкций в потоках и обладают низкой производительностью [18, 22, 30, 41];

– *lockset* – алгоритмы, проверяющие соответствие программы определённым правилам блокировки, которые кратко можно описать как «каждая разделяемая область памяти защищена переменной блокировки»; lockset-алгоритмы обладают существенно меньшими накладными расходами, чем happens-before, но производят значительное количество ложных срабатываний (то есть сигнализируют о гонках в ситуациях, когда их в действительности нет); по большей части это связано с тем, что они не поддерживают механизмы синхронизации, не связанные с захватом блокировок [21, 32, 35, 40];

– *гибридные (hybrid)* алгоритмы, полученные путем комбинирования первых двух (например, уточнение результатов работы lockset может выполняться с помощью happens-before) с целью использования преимуществ каждого из них и нивелирования их недостатков[1] [14, 33, 34, 42].

В табл. 1 кратко перечислены утилиты, обзор которых приводится в данной статье.

Табл. 1. Утилиты для динамического обнаружения гонок

Название утилиты	Вид алгоритма	Язык программирования	Год появления
–	happens-before	Modula-3, Smalltalk, C, C++ и др.	1990-е
Eraser	lockset	C++	1997
TRaDe	happens-before	Java	2001
Object race detection	lockset	Java	2001
IBM MSDK (ex-MTRAT)	hybrid	Java	2002
Multirace	hybrid	C++	2003

[1] Поскольку lockset-алгоритмы разрабатывались как более производительная, но менее точная альтернатива happens-before, эти два класса алгоритмов действительно хорошо «дополняют» друг друга. Более подробное рассуждение на эту тему можно найти в работе [34].

Это далеко не полный список существующих динамических детекторов. Выбирая подходы и инструментальные средства для включения в этот обзор, мы руководствовались стремлением дать исторический обзор развития области и в то же время описать её текущее состояние, в частности, уделить особое внимание динамическому обнаружению гонок в Java-программах. Кроме того, нас интересовали утилиты, которые можно использовать на индустриальных приложениях. Поэтому в обзоре мы уделяем внимание вопросам производительности. В следующем разделе мы делаем обзор динамических методов обнаружения гонок, а в заключении подводим итоги нашего исследования.

АЛГОРИТМ HAPPENS-BEFORE

Для обнаружения гонок нужно уметь отвечать на следующий вопрос: правда ли, что любые два обращения к разделяемому участку памяти из различных потоков, где одно из них является обращением на запись, упорядочены с помощью синхронизационных операций. Более формально, программа считается свободной от гонок, если в ней между любыми двумя обращениями разных потоков T1 (на запись) и T2 (на чтение или запись) к одной переменной происходит синхронизация (устанавливается *барьер памяти*): поток T2 *видит* изменения, сделанные потоком T1 [42]. В спецификации Java отношение, формируемое парами таких синхронизационных событий, называется «*synchronized-with*» (отношение *синхронизированности*) и формально определяется следующим образом:

– освобождение монитора всегда синхронизировано с его последующим захватом;

– запись volatile-переменной всегда синхронизирована с её последующим чтением;

– операция запуска потока всегда синхронизирована с первым действием в этом потоке;

– последнее действие потока T1 всегда синхронизировано с любым действием потока T2, который «знает», что поток T1 остановился (с помощью Thread.join() или Thread.isAlive());

– если поток T1 прерывает поток T2, прерывание синхронизировано с любым действием в любом потоке, который «знает», что поток T1 был прерван (перехватил InterruptedException или вызвал Thread.isInterrupted());

– запись значения по умолчанию каждой переменной всегда синхронизирована с первым действием любого потока.

Каждой паре синхронизированных событий соответствует определённый *синхронизационный объект* – например монитор, volatile-переменная или сам поток. Первый элемент пары событий мы в дальнейшем будем называть *захватом синхронизационного объекта*, а второй – его *освобождением* (по аналогии с частным случаем – захватом и освобождением блокировки).

Расширение отношения «synchronized-with» на множество всех событий программы называется «happens-before» (отношение *предшествования*):

– если событие A синхронизировано с событием B, то A happens-before B;

– действия в одном потоке, произошедшие одно после другого, находятся в отношении happens-before;

– завершение конструктора объекта предшествует запуску его финализатора (*finalizer*);

– отношение happens-before транзитивно замкнуто.

Два обращения A и B из различных потоков к одному участку памяти не образуют гонку, если A happens-before B или B happens-before A [24].

Для отслеживания отношения happens-before используется предложенный Лампортом в 1989 году механизм векторных часов [28]. Векторные часы представляют собой массив чисел, по длине равный количеству потоков в программе: каждому потоку соответствуют одни часы (одно число), увеличивающиеся по мере совершения потоком операций. Каждый поток хранит свою локальную копию векторных часов, синхронизируясь с копиями часов других потоков во время синхронизационных операций. Передача часов между потоками осуществляется с помо-

щью вспомогательных часов, ассоциированных с синхронизационными объектами.

Векторные часы хранят информацию о каждом потоке в системе и потому очень «дороги»: если в программе n потоков, то каждая операция над векторными часами требует $O(n)$ времени, а для хранения часов нужно $O(n)$ памяти. Поэтому, хотя теоретически и возможен сбор информации о программе с любой степенью точности, на практике эта точность сильно ограничена необходимостью обеспечивать эффективность как по потреблению ресурсов, так и по скорости выполнения программы. Динамический анализ приводит к большим накладным расходам, поэтому его крайне сложно применять для высоконагруженных приложений.

Алгоритм, основывающийся на отслеживании отношения happens-before, будем называть алгоритмом happens-before. В листинге 1 представлено его формальное описание.

Детекторы, основанные на этом алгоритме, часто называют *точными*, поскольку они ищут только произошедшие гонки и не производят ложных срабатываний. Первые детекторы гонок, основывавшиеся на алгоритме happens-before [7, 15, 16, 19, 30, 41], искали гонки на низком уровне – на уровне участков памяти. Это приводило к колоссальным накладным расходам. Например, ранние версии утилиты Intel Thread Checker [27] замедляли скорость работы программы в 200 раз. Кроме того, использование таких детекторов существенно замедляло разработку приложений, поскольку для перехвата синхронизационных событий и обращений к памяти в программе приходилось либо инструментировать весь исходный код, спецификация которого очень сложна, либо инструментировать исполняемые файлы на уровне ассемблера или машинных команд.

В настоящее время широкое распространение получили объектно-ориентированные языки с автоматическим управлением памятью и виртуальной средой исполнения (Java, C# и др.). Модель исполнения в таких языках проще реального машинного кода и, как следствие, детекторы для таких языков мо-

Листинг 1

Обозначения: V(x) - векторные часы, связанные с сущностью x. Компонента этих часов, соответствующая потоку t_i, обозначается как $V(x)(t_i)$. Индексы i и j принимают значения от 1 до общего количества потоков в программе.

Инициализация:
Для каждых потока t, разделяемой переменной v и синхронизационного объекта l создаются векторные часы и инициализируются следующим образом:
$\forall i,\ \forall j \neq i\colon V(t_i)(t_i) := 1,\ V(t_i)(t_j) := 0.$
$\forall v,\ \forall i\colon V(v)(t_i) = 0.$
$\forall l,\ \forall i\colon V(l)(t_i) = 0.$

Захват синхронизационного объекта l потоком t:
$\forall i\colon V(t)(t_i) := \max\{V(t)(t_i), V(l)(t_i)\}$

Освобождение синхронизационного объекта l потоком t:
1. $V(t)(t) := V(t)(t) + 1.$
2. $\forall i\ V(l)(t_i) := \max\{V(t)(t_i), V(l)(t_i)\}.$

Обращение потока t к разделяемой переменной v:
1. Проверить, что верно условие $\forall i\colon V(t)(t_i) \geq V(v)(t_i)$ и $\exists j\colon V(t)(t_j) > V(v)(t_j)$. Если условие не выполнено, обнаружена гонка.
2. $V(v) := V(t).$

Замечание. Как правило, множественные одновременные чтения без каких-либо блокировок считаются допустимыми. Для корректной обработки таких ситуаций нужно для каждой разделяемой переменной хранить двое часов – для обращений на запись и на чтение – и соответствующим образом модифицировать последнюю часть алгоритма.

гут искать гонки на более высоком уровне, что существенно увеличивает их производительность.

Так, авторы подхода TRaDe (topological race detection) [18], основанного на happens-before алгоритме, использовали для анализа *байт-код* виртуальной машины Java (JVM), который достаточно хорошо структурирован. Например, в нем существуют лишь четыре инструкции, позволяющие создать новый объект. Согласно спецификации Java [24], большая часть инструкций не связана с гонками и может быть абсолютно безопасно проигнорирована. У каждого потока есть своя стековая память, в которой он создает локальные переменные и параметры для каждого вызова функции, и только этот поток манипулирует своими локальными данными, поэтому инструкции, которые оперируют только данными локальной памяти, не могут привести к гонке. Из 181 инструкции JVM лишь 20 являются потенциально «опасными» в смысле появления гонок. Ключевая идея подхода TRaDe заключалась в том, чтобы начинать анализ этих инструкций только после того, как объект перестанет быть *локальным*, то есть станет доступен нескольким потокам. Эта идея привела к существенному улучшению производительности и оказалась достаточно проста в реализации на уровне анализа байт-кода Java за счёт его высокоуровневости. Авторы подхода TRaDe встроили свой детектор в JVM, что дало им дополнительную возможность оптимизации – во время сборки мусора они определяли, какие разделяемые объекты доступны только одному потоку, и не проводили их анализа. Ещё одна проблема, которую решили авторы TRaDe, – это проблема динамического создания потоков. Простота создания потоков в Java привела к тому, что многие программы в большом количестве порождают короткоживущие потоки. Поскольку размер векторных часов динамически растет с увеличением числа потоков в программе, нужно уметь отслеживать окончание работы потока и удалять соответствующие ему компоненты векторных часов. Но это нельзя делать сразу же после завершения работы потока, потому что появляется риск пропустить гонку, возникшую между записью переменной в этом потоке и последующим обращением к этой же переменной из другого потока. Авторы TRaDe разработали концепцию «гибких часов» (accordion clock) [17], использование которых помогло им преодолеть эту проблему. Ключевой идеей «гибких часов» является периодическая проверка для каждого завершившегося потока, остались ли разделяемые объекты, последнее обращение к которым было из этого потока. Если таких объектов нет, то часы завершившегося потока можно удалять. Авторы TRaDe проверили свою утилиту на ряде приложений и указывают, что производительность исходной программы падает в 4–15 раз, что на два порядка лучше производительности первых happens-before детекторов. Наша пилотная реализация алгоритма happens-before с учетом оптимизаций подхода TRaDe, представленная в работе [6], также подтвердила эти результаты.

В 2009 году в работе [22] было показано, что алгоритм векторных часов можно значительно улучшить: его производительность может быть практически такой же, как у неточных детекторов, и при этом удастся избежать потери точности и ложных срабатываний. Исследования авторов показывают, что в типичных приложениях лишь 3–4% перехватываемых операций являются операциями синхронизации, а остальные 96–97% оказываются обращениями к разделяемым объектам на чтение или запись. Для подавляющего большинства обращений к разделяемым объектам не нужно хранить полные векторные часы, а достаточно хранить лишь компоненту часов и номер последнего потока, который обращался к объекту (совокупность двух этих чисел авторы назвали *эпохой – epoch*). Соответственно, кроме существенной экономии памяти, получается значительный прирост по скорости – операции проверки на гонку и слияния часов теперь занимают не $O(N)$ времени, а $O(1)$. Согласно исследованиям авторов, полные часы необходимы лишь для 0.1% обращений на чтение и 0.1% обращений на запись. Авторы разработали алгоритм FastTrack [22], позволяющий динамически переключаться меж-

ду часами и эпохами, и протестировали его в сравнении с основными существующими алгоритмами – Djit+ [34] (эталонная реализация алгоритма happens-before), Goldilocks [21], TRaDe [18] и Lockset [40]. Полученный алгоритм оказался более производительным, чем первые три (с учетом того, что, в частности, оригинальный Goldilocks встроен в JVM), и сопоставимым по производительности с четвёртым. Кроме того, утилита FastTrack была протестирована на Eclipse (авторы запускали разные операции в Eclipse – анализировалось от 23000 до 290000 строк кода) и снова показала хорошие результаты, сравнимые с результатами основного lockset-алгоритма [40] (более производительный, но менее точный класс алгоритмов подробнее описывается в следующем разделе). Производительность исходной программы падала всего в 1.5–2.5 раза, причем значительная часть замедлений происходила из-за динамического инструментирования байт-кода, что говорит о том, что уровень накладных расходов понижается через некоторое время после запуска программы.

АЛГОРИТМ LOCKSET

Проблемы с производительностью happens-before детекторов привели к созданию *неточных* детекторов, которые позволяли ценой потери части гонок и выдачи «ложных срабатываний» уменьшить нагрузку на систему.

Основным механизмом защиты данных в многопоточной среде является заключение их в критические секции и защита этих секций с помощью мониторов, которые захватываются потоками перед входом в критическую секцию и освобождаются сразу же после выхода из неё. В 1997 году в работе [40] было предложено проверять, что программа следует определённым правилам блокировки, которые сводятся к принципу «каждая разделяемая переменная должна быть защищена монитором».

Алгоритм, основывающийся на этой идее, получил название «lockset» и был впервые реализован в утилите Eraser [40]. Для каждой разделяемой переменной Eraser отслеживает множество мониторов, которые защищают все обращения к ней на чтение и на запись из различных потоков. Перед любым таким обращением к переменной Eraser проверяет, захватил ли поток какой-либо монитор из этого множества, и удаляет из него мониторы, не захваченные потоком. Если это множество становится пустым, то это означает, что никакие мониторы не защищают эту переменную, и Eraser сигнализирует об ошибке. В листинге 2 представлено формальное описание алгоритма lockset (детали см. в [40]).

Но этот подход приводит к большому количеству ложных срабатываний: как при использовании альтернативных механизмов синхронизации (fork/join, атомарных операций), так и в ситуациях, когда правила блокировки не выполняются, но гонки на самом деле нет – например, в случае использования подхода «один писатель, много читателей» или при защите переменной разными мониторами в разные промежутки времени.

Листинг 2

Инициализация. Для каждой переменной v, множество C(v) содержит все переменные блокировки в программе.

Обращение к переменной v из потока t:
1. Пусть L – множество всех блокировок, удерживаемых потоком t.
2. C(v) = C(v) ∩ L.
3. Если C(v) = ∅, то обнаружена гонка.

Замечание. Как правило, множественные одновременные чтения без каких-либо блокировок считаются допустимыми. В этом случае после шага 1 в L добавляется мнимая переменная блокировки readers_lock - считается, что каждый поток удерживает её в любой момент времени.

Но, несмотря на это, даже первые реализации алгоритма lockset в формате «proof-of-concept» без дополнительных оптимизаций, представленные в [13, 40], находили значительную часть гонок и демонстрировали при этом существенное снижение накладных расходов по сравнению с точным алгоритмом на векторных часах. Утилита Eraser была первым динамическим детектором, который запускался и демонстрировал удовлетворительную производительность на промышленных системах [40].

Помимо выигрыша в производительности, алгоритм lockset лучше алгоритма happens-before ещё и тем, что не зависит от чередования потоков. В самом деле, в то время как алгоритм happens-before может быть «сбит с толку» случайным возникновением упорядоченности между двумя обращениями к незащищённой переменной, lockset обнаружит эту незащищённость, потому что он оперирует множествами ассоциированных мониторов.

Алгоритм lockset стал популярным и впоследствии много раз модифицировался и дополнялся [14, 32, 35]. В частности, алгоритм, представленный в [35], ищет гонки на уровне объектов, а не фрагментов памяти, и существенно выигрывает в производительности по сравнению с Eraser.

Ещё одна модификация алгоритма lockset описана в [21]. В этой работе представлен алгоритм Goldilocks – точный lockset алгоритм, к которому добавлены механизмы, позволяющие корректно обрабатывать все остальные способы синхронизации, отличные от постоянной защиты критических секций мониторами. Речь идет о корректной публикации локальных данных потока в разделяемую память, о защите разделяемых данных разными мониторами в разные промежутки времени и о защите полей объекта посредством защиты всего объекта с помощью переменной блокировки. Утилита для Java-программ, основанная на алгоритме Goldilocks, корректно обрабатывает операции start и join за счёт наследования множеств блокировок, удерживаемых потоками, но всё же не поддерживает механизм низкоуровневой синхронизации с помощью средств пакета java.util.concurrent [20], который очень часто используется в Java-приложениях.

КОМБИНИРОВАННЫЕ АЛГОРИТМЫ

Таким образом, если сравнивать алгоритмы lockset и happens-before, то каждый имеет свои достоинства и недостатки [34]. Последний точен, но приводит к значительным накладным расходам на потребляемые ресурсы и скорость выполнения программы. Алгоритм lockset, в свою очередь, обладает лучшей производительностью, независим от чередования потоков, но выдает очень большое количество ложных срабатываний. В этом свете естественной выглядела идея разработки гибридных алгоритмов, которые бы брали все достоинства от двух основных и взаимно компенсировали их недостатки.

Результаты одного из первых исследований на эту тему содержатся в работе [33]. Авторы этой работы сохраняют историю порождения, временных приостановок и завершения потоков, дополнительно позволяя пометить некоторые пары Java-методов как устанавливающие отношение happens-before. Для операций чтения и записи информация для happens-before алгоритма не хранится, что на практике означает оперирование с очень малым количеством событий в потоках. В итоге получается подмножество отношения happens-before, но авторы показывают, что его достаточно для устранения большинства ложных срабатываний. Кроме того, авторам удалось получить дополнительный прирост производительности за счёт применения *escape-анализа* – проверки объектов на принадлежность к нескольким потокам. Пока все обращения к объекту осуществляются лишь одним потоком, никакая история операций для этого объекта не хранится. Эта оптимизация может привести к пропуску гонок, но значительно улучшает производительность. Авторы проверяли свой алгоритм как на стандартных тестах производительности [23], так и на таких популярных контейнерах Web-приложений, как Apache Tomcat [9] и Resin [38], и пришли к заключению, что периодическое

подключение алгоритма happens-before в рамках их подхода фактически не ухудшает производительность программы, но при этом существенно снижает количество ложных срабатываний и увеличивает количество реально обнаруженных гонок.

В статье [14] авторы предложили новый комбинированный подход к обнаружению гонок, который более эффективен и точен по сравнению с предыдущими – время выполнения программы увеличивается лишь на 13–42 % на микротестах производительности. Ключевая идея этой работы – отслеживать отношение *weaker-than*, что позволяет выделить обращения к памяти, которые не могут вызвать новые гонки, и исключить их из анализа. Другой оптимизацией является то, что подход реагирует не на все обращения к памяти, которые вовлечены в гонку, но гарантированно выдается, как минимум, одно такое обращение для каждого участка памяти, по которому была обнаружена гонка. Авторы утверждают, что почти все найденные гонки соответствуют реальным ошибкам в программе. Также этот детектор может быть легко модифицирован для работы в режиме post-mortem (анализ информации, собранной во время работы программы, осуществляется уже после её завершения) посредством ведения журнала обращений к памяти с последующим анализом этого журнала. Авторы не предоставляют информации о производительности утилиты на крупных проектах.

На результатах этих двух работ ([14], [33]) основывается утилита IBM MSDK [25, 36] – единственная бесплатная утилита для динамического обнаружения гонок в Java-программах, обнаруженная нами. Производительность MSDK на микротестах имеет тот же порядок, что и у оригинальных утилит, представленных в этих работах, но на больших приложениях (WebSphere Application Server [26], Apache Tomcat [9]) скорость работы программы падает в 3–30 раз, что подтверждает аналогичные наблюдения быстрого снижения производительности с увеличением размера программы и количества потоков в ней, сделанного авторами [34]. Последняя версия MSDK была выпущена несколько лет назад, и она всё ещё не поддерживает значительную часть низкоуровневых средств синхронизации из популярного пакета java.util.concurrent, появившегося в Java 1.5 (2004 год), а также содержит ряд ошибок, связанных с потреблением памяти, что не позволяет применять её на длительно работающей системе[1].

Впоследствии гибридный подход получил активное развитие. Например, в работе [42] представлена утилита RaceTrack для .NET-программ, которая использует алгоритм happens-before для очистки множеств ассоциированных блокировок, которыми оперирует алгоритм lockset. Это позволяет избежать разрастания этих множеств по мере выполнения программы и приводит к росту производительности.

Ещё одна идея интеграции алгоритмов happens-before и lockset представлена в работе [34]. В ней представлена утилита MultiRace, предназначенная для обнаружения гонок в многопоточных программах на языке C++. Представленный подход инструментирует исходный код программы и базируется на векторных часах, но, в силу использования алгоритма lockset, существенно сокращается количество ресурсоёмких операций с векторными часами и снимается зависимость от чередования потоков. Кроме того, MultiRace стала первым детектором, который динамически переключает уровень детализации поиска гонок без потери точности, что положительно сказывается на производительности. Он начинает работу на уровне страниц памяти и при необходимости переключается на уровень объектов, но не ниже. Такой подход оправдан, поскольку если программа не содержит гонок на высоком уровне, то и на более низком уровне их тоже нет. MultiRace также предоставляет программисту возможность регулировать уровень детализации поиска. Утилита была проверена на приложениях с 1–8 потоками и показала увеличение накладных расходов не более чем в 2–3 раза. Однако авторы упо-

[1] В свете недавней реорганизации проекта IBM alphaworks, в рамках которого развивалась эта утилита (и её предшественница MTRAT), ее будущее выглядит очень туманно.

минают, что при большем количестве потоков производительность быстро падает.

ДРУГИЕ ПОДХОДЫ

Динамический анализ Java-программ традиционно осуществлялся посредством инструментирования байт-кода с использованием соответствующих библиотек [8, 10]. С распространением аспектно-ориентированного подхода [39], призванного упростить решение подобных задач (перехват работы программы и внедрение в неё дополнительной сквозной функциональности), возникла идея применить этот подход к обнаружению гонок. Так, в АОП-систему AspectJ [11] были добавлены точки соединения (*pointcuts* – точки в программе, удовлетворяющие определённым предикатам) «lock», «unlock» и «mayBeShared», которые упрощают использование AspectJ для отслеживания и обнаружения гонок. Однако последняя точка соединения обладает недостаточно высокой точностью и может указывать на объекты, которые не являются разделяемыми. Кроме того, механизмы сихнронизации, отличные от захвата переменных блокировки, всё равно нужно отслеживать «вручную».

Одним из популярных в последнее время способов снижения накладных расходов при динамическом поиске гонок является семплирование (*sampling*). В рамках этого подхода анализируются не все обращения к памяти, а лишь некоторая их часть, например, 1–3 %, удовлетворяющая определённым критериям. Авторы утилиты LiteRace [29] рассмотрели несколько эмпирических критериев и показали, что, несмотря на кажущуюся неэффективность – ведь, для того чтобы обнаружить гонку, нужно, чтобы два обращения к одному и тому же участку памяти были зафиксированы, – их утилита хорошо подходит для поиска гонок в «холодных регионах» (*cold regions,* участки кода, которые редко выполняются). Утилита начинает работу с коэффициентом семплирования 100 % (то есть, отслеживает все операции) и понижает его для каждого участка кода с каждым последующим его посещением до минимального уровня. Таким образом, редко выполняемые участки кода анализируются фактически всегда, а часто выполняемые – редко, что «уравнивает шансы». Авторы проверили свою утилиту на ряде приложений, в частности, с коэффициентом семплирования 2 % они обнаружили порядка 70 % известных гонок в браузере Firefox.

Утилита Pacer, представленная в работе [12], также использует семплирование, но предоставляет гарантии пропорционального обнаружения гонок. Если обычные детекторы ищут самые ранние гонки, то Pacer находит гонки со скоростью, пропорциональной уровню семплирования. Разумеется, многие обращения к памяти и, как следствие, гонки остаются необработанными, но авторы, проанализировав множество программ, пришли к выводу, что единожды произошедшая гонка довольно часто повторяется впоследствии, так что рано или поздно она, вероятно, будет обнаружена. При коэффициенте семплирования 1–3 % накладные расходы колеблются в интервале 52–86 %. Для проверки своей гипотезы о повторяемости гонок авторы запускали утилиту по нескольку раз на одной и той же программе с разными коэффициентами семплирования и замеряли, какой процент гонок повторялся. Также авторы Pacer сравнили свою утилиту с LiteRace, описанной выше, и пришли к выводу, что разные подходы к семплированию обладают примерно равными возможностями.

ЗАКЛЮЧЕНИЕ

В данной работе мы рассмотрели различные динамические подходы к обнаружению гонок в многопоточных программах, уделив особое внимание программам, написанным на языке Java. В таблице 2 представлены все утилиты, которые были рассмотрены в данной статье, и приведён сравнительный анализ этих утилит по производительности. Для каждой утилиты приводятся данные о том, на сколько процентов (%) или во сколько раз (x) в среднем замедлялось выполнение стандартных тестов производительности; если авторы приводят результаты апробации своей утилиты на более крупных приложениях, это указывается дополнительно.

Динамические детекторы основываются либо на отслеживании упорядоченности обращений потоков к памяти, либо на проверке следования этих обращений определённым правилам блокировок. Алгоритмы первого вида обладают высокой точностью и не производят ложных срабатываний, алгоритмы второго вида меньше зависят от чередования потоков и обладают более высокой производительностью. Последние работы в области динамических детекторов привели к созданию более эффективных комбинированных алгоритмов. Было разработано множество прототипов и несколько полноценных детекторов, а также проведена их апробация на стандартных тестах производительности и, в редких случаях, на нескольких больших приложениях. В последних случаях, как правило, сообщается о резком снижении производительности при увеличении размера программной системы. По-видимому, подобные эффекты связаны с возникающими проблемами разрастания внутренних структур данных и увеличением времени жизни и числа потоков в программе. Среди способов повышения производительности динамических детекторов можно выделить следующие:

– комбинирование различных методик – например использование escape-анализа в динамическом подходе или гибридизация lockset и vector-clock алгоритмов;

– эмпирическое выделение наиболее часто выполняющихся участков алгоритма и их дальнейшая оптимизация;

– сокращение анализируемой области программы, зачастую за счёт потери точности (семплирование);

– предоставление детектору дополнительной информации об анализируемой программе – как правило, с помощью аннотаций.

Однако, по нашему мнению, только возможность использовать эти наработки в совокупности, гибко меняя стратегию оптимизации в зависимости от специфики приложения, может позволить эффективно применять детектор при разработке и использовании «долгоживущих» нагруженных систем. Несмотря на значительное число результатов в области динамического обнаружения гонок, подавляющее большинство исследований завершались разработкой прототипа и апробацией его на модельных примерах, и лишь очень малое количество утилит доступно для непосредственного использования. Так, среди обнаруженных нами утилит для динамического обнаружения гонок в Java только IBM MSDK [25] продемонстрировала удовлетворительные результаты работы на ряде модельных проектов (порядка 1000 классов, 10 потоков). Мы провели детальное исследование этой утилиты и выявили ряд внутренних ошибок, которые не позволили использовать её на реальных высоконагруженных приложениях. Динамических детекторов для Java-программ с открытым исходным кодом нам найти не удалось. Учитывая схожее устройство платформ Java и .NET, в принципе, возможен перенос .NET-утилит (например, [29, 42]) на Java-платформу, но у них также закрыт исходный код. Ещё одной существенной проблемой рассмотренных решений является частичная или полная неспособность учитывать протоколы синхронизации, построенные непосредственно на базе основных примитивов (таких как CAS и его производные) и учитывать высокоуровневые контракты, обеспечиваемые библиотеками потокобезопасных структур, такими как пакет java.util.concurrent.

Таким образом, актуальной является задача разработки динамического детектора обнаружения гонок для промышленных Java-приложений, при разработке которых стоит также уделить особое внимание пользовательскому интерфейсу, посредством которого программист будет взаимодействовать с детектором. В связи с этим нам видится перспективным использование DSM-подхода (*Domain Specific Modeling*) для визуализации графа потоков управления, фрагментов программ при аннотировании, при отображении найденных гонок и т.д.[1]

[1] DSM-средства для .NET-платформы представлены в работах [2, 3], для Java в [4], интересная отечественная разработка представлена в работе [5], перспективный подход к разработке средств навигации по визуальным моделям изложен в [1].

Табл. 2. Сравнительный анализ утилит для динамического обнаружения гонок

Название утилиты	**Краткое описание**	**Алгоритм**	**Производительность**	**Год появления**
–	Множество реализаций в начале 90-х гг., работавших на уровне отслеживания обращений к участкам памяти. Сейчас не представляют интереса ввиду неактуальности и огромных накладных расходов.	happens-before	До 200х, потребление памяти не позволяло их запускать даже на достаточно серьёзных тестах	1990-е
Eraser	Первая реализация неточного алгоритма lockset, появившегося на смену медленному и потребляющему много ресурсов алгоритму happens-before. Впоследствии неоднократно улучшался.	lockset	10–30х	1997
TRaDe	Прирост производительности – работает на уровне байт-кода Java, отслеживает объекты, которые перестали быть разделяемыми несколькими потоками.	happens-before	4–15х, рост потребления памяти – в три раза	2001
Object race detection	Модификация алгоритма lockset. Выполняет escape-анализ (отслеживает объекты, которые перестали быть разделяемыми, то есть стали принадлежать конкретному потоку). Ищет гонки на уровне объектов.	lockset	16–129 %, не менее 3х на приложениях среднего размера	2001
IBM MSDK (ex-MTRAT)	Многофазный детектор – сначала применяет статический анализ, затем динамический. Дополнительно осуществляет escape-анализ и использует все оптимизационные наработки своего времени.	lockset	2х, до 30х на крупных приложениях	2002
Multirace	Одна из первых реализаций идеи комбинирования алгоритмов happens-before и lockset. Первый использован для примитивов синхронизации, не связанных с захватом блокировок, второй – для остальных случаев.	hybrid	1–2.5х на приложениях с количеством потоков не более 8, гораздо хуже при большем количестве потоков	2003
RaceTrack	Наряду с использованием lockset, хранятся векторные часы для каждого потока, которые используются только тогда, когда lockset-алгоритм находит гонку.	hybrid	до 3х	2005
Goldilocks	Алгоритм lockset модифицирован для поддержки механизмов синхронизации, не связанных с захватом блокировок. Утилита может использовать результаты работы некоторых статических детекторов.	lockset	до 11х	2007
Racer	Использует АОП-подсистемы для внедрения в программу.	любой	10–20х	2008
FastTrack	Проведена оптимизация векторных часов, что приводит к производительности, сравнимой с производительностью неточных детекторов.	happens-before	В 3 раза быстрее «чистого happens-before», сравнима с lockset	2009
LiteRace/ Pacer	Использует «семплирование» – выборочное отслеживание вызовов методов в программе. Существенно увеличивает производительность ценой возможной потери гонок.	любой	1–2.5х/52–86 % при коэффициенте семплирования 1–3 %	2009/2010

Литература

1. *Кознов Д.В.* О спецификации диаграммных преобразований в графических редакторах. Вестник Санкт-Петербургского Университета. Сер. 10. Вып. 3. 2011. С. 100–111.

2. *Павлинов А., Кознов Д., Перегудов А. и др.* О средствах разработки проблемно-ориентированных визуальных языков // Системное программирование. Вып. 2: Сб. статей / Под ред. А.Н.Терехова, Д.Ю.Булычева. СПб.: Изд-во СПбГУ, 2006. С. 116–141.

3. *Павлинов А., Кознов Д., Перегудов А., Бугайченко Д., Казакова А., Чернятчик Р., Фесенко Т., Иванов А.* Комплекс средств разработки проблемно-ориентированных визуальных языков. Вестник Санкт-петербургского университета. Серия 10. Информатика, № 2, 2007. С. 86–96.

4. *Сорокин А.В., Кознов Д.В.* Обзор Eclipse Modeling Project // Системное программирование. Вып. 5: Сб. статей / Под ред. А.Н.Терехова, Д.Ю.Булычева. 2010. С. 6–31.

5. *Терехов А.Н., Брыксин Т.А., Литвинов Ю.В., Смирнов К.К., Никандров Г.А., Иванов В.Ю., Такун Е.И.* Архитектура среды визуального моделирования QReal // Системное программирование. Вып. 4: Сб. статей / Под ред. А.Н.Терехова, Д.Ю.Булычева. 2009. С. 171–196.

6. *Трифанов В.Ю.* Динамическое обнаружение гонок в Java-программах с помощью векторных часов // Системное программирование. Вып. 5: Сб. статей / Под ред. А.Н.Терехова, Д.Ю.Булычева. С. 95–116.

7. *Adve S., Hill M., Miller B., Netzer R.* Detecting data races on weak memory systems. In Proceedings of the 18th Annual International Symposium on Computer Architecture, 1991. P. 234–243.

8. Apache Byte Code Engineering Library, http://commons.apache.org/bcel/ (дата обращения: 26.12.2011).

9. Apache Tomcat Project, http://tomcat.apache.org/ (дата обращения: 26.12.2011).

10. ASM Java Bytecode Manipulation and Analysis Framework, http://asm.ow2.org/ (дата обращения: 26.12.2011).

11. AspectJ Project, http://www.eclipse.org/aspectj/ (дата обращения: 26.12.2011).

12. *Bond M., Coons K., McKinley K.* Pacer: Proportional Detection of Data Races. In Proceedings of 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010), Toronto, June 2010. P. 255–268.

13. *Cheng G., Feng M., Leiserson C., Randall K., Stark A.* Detecting data races in Cilk programs that use locks. In?Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures, 1998. P. 298–309.

14. *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.

15. *Choi J., Miller B., Netzer R.* Techniques for debugging parallel programs with flowback analysis. ACM Transactions on Programming Languages and Systems. Vol. 13. Issue 4, 1991. P. 491–530.

16. *Choi J., Min S.* Race Frontier: reproducing data races in parallel programs debugging. In Proceedings of the third ACM SIGPLAN Symposium on Principles & practice of parallel programming, April 1991. P. 145–154.

17. *Christiaens M, Bosschere K.* Accordion Clocks: Logical Clocks for Data Race Detection Lecture Notes in Computer Science. Vol. 2150, 2001. P. 494–503.

18. *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.

19. *Dinning A., Schonberg E.* Detecting access anomalies in programs with critical sections. In Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging, 1991. P. 85–96.

20. Documentation of java.util.concurrent package, http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html (дата обращения: 26.12.2011).

21. *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.

22. *Flanagan C., Freund S.* FastTrack: Efficient and Precise Dynamic Race Detection. In ACM Conference on Programming Language Design and Implementation, 2009. P. 121–133.

23. Java Grande Forum Multi-Threaded Benchmarks, http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html (дата обращения: 26.12.2011).

24. Java Language Specification, Third Edition. Threads and Locks. http://java.sun.com/docs/books/jls/third_edition/html/memory.html (дата обращения: 26.12.2011).

25. IBM Multicore Software Development Kit. Temporary URL: https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=9a29d9f0-11b1-4d29-9359-a6fd9678a2e8 (дата обращения: 26.12.2011).

26. IBM WebSphere Application Server, http://ibm.com/software/webservers/appserv/was/ (дата обращения: 26.12.2011).

27. Intel Thread Checker, http://software.intel.com/en-us/intel-thread-checker/ (дата обращения: 26.12.2011).
28. *Lamport L.* Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM. Vol. 21, Issue 7, 1978. P. 558–565.
29. *Marino D., Musuvathi M., Narayanasamy S.* LiteRace: Effective Sampling for Lightweight Data-Race Detection. PLDI '09 Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. Vol. 44, Issue 6, 2009. P. 134–143.
30. *Mellor-Crummey J.* On-the-fly detection of data races for programs with nested fork-join parallelism. In Proceedings of the 1991 ACM/IEEE conference on Supercomputing, 1991. P. 24–33.
31. *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.
32. *Nishiyama H.* Detecting data races using dynamic escape analysis based on read barrier. In Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium. Vol. 3, 2004. P. 10.
33. *O'Callahan R., Choi J.-D.* Hybrid Dynamic Data Race Detection. In PPOPP, 2003. P. 167–178.
34. *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.
35. *Praun C., Gross T.* Object race detection. In ACM SIGPLAN Notices. Vol. 36, Issue 11, 2001. P. 70–82.
36. Qi Y., Das R., Luo Z., Trotter M. MulticoreSDK: a practical and efficient data race detector for real-world applications. In Proceedings Software Testing, Verification and Validation (ICST), IEEE, 21-25 March 2011. P. 309–318.
37. *Raza A.* A Review of Race Detection Mechanisms. Lecture Notes in Computer Science. Vol. 3967, 2006. P. 534–543.
38. Resin Application Server, http://www.caucho.com/resin/ (дата обращения: 26.12.2011).
39. *Safonov V.O.* Using aspect-oriented programming for trustworthy software development. Wiley Interscience, John Wiley & Sons, Inc., 2008.
40. *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.
41. *Schonberg E.* On-the-fly Detection of Access Anomalies. In Proceedings of the SIGPLAN '87 symposium on Interpreters and interpretive techniques, 1987. P. 285–297.
42. *Yu Y., Rodeheffer T., Chen W.* RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In SOSP, 2005. P. 221–234.
43. *Трифанов В.Ю., Цителов Д.И.* Статические и post-mortem методы обнаружения гонок в параллельных программах // Компьютерные инструменты в образовании, 2011. № 5. С. 3–13.

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 evolution of existing approaches is considered and their advantages and drawbacks are analyzed. Special attention is paid to automatic race detection in Java applications.

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.