Динамическое обнаружение гонок в Java-программах с помощью векторных часов

В. Ю. Трифанов
vitaly.trifanov@gmail.com

Наличие в параллельной программе состояний гонки — одна из наиболее распространенных и тяжелых в обнаружении ошибок многопоточного программирования. Гонки слабо локализованы, трудновоспроизводимы и ведут к повреждению глобальных структур данных. В статье обсуждается вопрос автоматизированного динамического обнаружения гонок для многопоточных программ, написанных на языке Java, предлагается адаптация для Java-программ точного алгоритма DJIT+, основанного на вычислении отношения «happens-before» с помощью векторных часов, рассматриваются результаты апробации получившегося алгоритма, обсуждаются требования по доработке данного детектора для применения к большим промышленным Java-приложениям.

Введение

Используемые в настоящее время вычислительные устройства всё в большей степени являются многоядерными или многопроцессорными, что позволяет разрабатывать для них эффективные параллельные программы. Однако разработка таких программ является сложной задачей, поскольку разработчикам приходится размышлять в терминах нескольких потоков управления, одновремен-

но выполняющих разные действия. Это приводит к большому количеству ошибок в программах. К самым частым и тяжелым в обнаружении ошибкам многопоточных программ относятся состояния гонки.

Состояние гонки (data race) наступает, когда два или более потока в паралелльной программе одновременно обращаются к одной структуре данных, между этими обращениями нет принудительного упорядочивания по времени, и хотя бы одно из них — обращение на запись [28]. В подавляющем большинстве случаев состояния гонки нежелательны, поскольку их появление может нарушить ход выполнения программы и привести к непредсказуемым результатам. Поскольку конкретный порядок выполнения потоков, приводящий к состоянию гонки, специфичен для каждого запуска программы и зависит от порядка выполнения операций в потоках, определяемого операционной системой, то 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, включая ее зависание и пр. — программа будет продолжать выполняться, что может привести к загадочным сбоям впоследствии.

В связи с этим автоматическое обнаружение состояний гонки в программах является важной и актуальной задачей. Исследования в данной области ведутся уже более двух десятков лет, создано множество детекторов гонок, использующих различные подходы [12, 13, 14, 20, 34, 37]. Однако нет «идеальных» детекторов, находящих все гонки в программах и не производящих при этом ложных срабатываний, а также не налагающих чрезмерных накладных расходов на вычислительные ресурсы. Существующие динамические детекторы по большей части либо являются точными, но медленными, либо быстрыми, но производящими много ложных срабатываний.

В статье обсуждается вопрос автоматизированного динамического обнаружения гонок для многопоточных программ, написанных на языке Java, и предлагается адаптация для Java-программ точного алгоритма DJIT+ [33], основанного на вычислении отноше-

ния «happens-before» с помощью векторных часов. В статье представлены также результаты апробации получившегося алгоритма, обсуждаются требования по доработке результата для применения к большим промышленным Java-приложениям.

1. Состояния гонки

В многопоточной программе каждому потоку при его запуске выделяется отдельный фрагмент памяти, доступный только этому потоку и называемый *собственной памятью потока*. Изначально в памяти потока хранятся значения разделяемых переменных, скопированные из общей памяти программы. Когда поток обращается к переменной на чтение, на самом деле он обращается к своей копии переменной, находящейся в его памяти. При изменении значения 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. На рис. 1 представлен пример работы программы с такой организацией памяти. Из этого примера видно, что без осуществления должной синхронизации возможны проблемные ситуации, когда потоки не видят изменений, сделанных друг другом. В данном случае каждый поток увеличил значение переменной `i` на единицу, но в итоге она оказалась равной 1, а не 2. Таким образом, мы имеем состояние гонки в программе.

Гонки могут быть очень опасными — например, описанная в примере выше ситуация может произойти с банковским счетом, который два разных потока пытаются одновременно увеличить 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. Если же речь идет о медицинских системах, то гонки могут привести еще к куда более серьезным последствиям — например, от передозировок, допущенных аппаратом лучевой терапии Therac-25 по причине гонок, скончались как минимум двое человек [23].

Программа считается свободной от гонок, если в ней между любыми двумя обращениями разных потоков T_1 (на запись) и T_2 (на чтение или запись) к одной переменной происходит синхронизация (устанавливается *барьер памяти*): поток T_2 *видит* изменения, сделанные потоком T_1. Формально это описывается отношением частичного порядка «happens-before» на множестве всех событий программы. Два обращения A и B к переменной X связаны отношением «happens-before», если:

- A и B произошли в одном потоке, и событие A произошло перед событием B;
- событие A — это сохранение потоком значения переменной X в разделяемую память программы, а событие B — это чтение другим потоком значения переменной X из нее.

Отношение «happens-before», будучи отношением частичного порядка, транзитивно замкнуто, антисимметрично и нерефлексивно. Программа считается свободной от гонок в том случае, если все события в ней упорядочены с помощью отношения «happens-before».

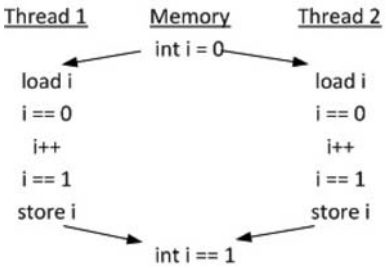

Рис. 1. Пример гонки между двумя процессами

2. Обзор существующих подходов

Задача проверки существования потенциальных состояний гонки в программе и родственная ей задача по обнаружению тупиков при помощи статического анализа кода программы алгоритмиче-

ски неразрешимы в общем случае и NP-полны для конечных графов выполнения [29, 30]. Поэтому единственным способом гарантировать отсутствие ошибок синхронизации является полное исключение доступа к механизмам возникновения таких ошибок в языках программирования. По этому пути идут такие перспективные языки как Erlang, Jocaml, Scala и пр. К сожалению, использование второго языка программирования в проекте часто является нецелесообразным, особенно при поддержке и расширении существующих проектов. Другие подходы используют специальные библиотеки классов и аннотации для облегчения разработки корректных параллельных программ [11, 19], что во многих случаях позволяет избежать состояний гонки, однако не дает гарантию их отсутствия [1, 5, 6, 15].

Исследовательские работы в данной области сфокусированы на динамическом [7, 8, 12, 21, 25, 32, 35, 37, 38] и статическом [9, 13, 20, 27, 39] подходах к обнаружению гонок. Статические подходы анализируют исходный код программы, не запуская ее, динамические подходы рассматривают трассу приложения во время его выполнения или после завершения.

2.1. Статические подходы к обнаружению гонок

Наиболее распространенный подход для статического обнаружения состояний гонки — проведение анализа исходного кода программы во время компиляции и оповещение обо всех потенциальных гонках, которые теоретически могут произойти.

Существует множество работ, посвященных исследованию статического анализа программы для обнаружения ситуаций гонки. К самым первым статическим средствам обнаружения гонок можно отнести систему Warlock [39], предназначенную для C-программ, а также ESC [10] и ESC/Java [22] (для программ, написанных на Modula-3 и Java, соответственно). Для уменьшения количества ложных срабатываний эти средства требуют создания множества аннотаций кода, что приводит, фактически, к невозможности их использования для больших приложений. Эта проблема отчасти решается авторами подхода [13]: их утилита RacerX осуществляет межпроцедурный анализ возможных потоков выполнения програм-

мы, используя особые алгоритмы ранжирования для отсеивания ложных срабатываний. Она была успешно использована для проверки исходных кодов ОС Linux 2.5.62 и обнаружила там несколько серьезных ошибок. Для программ, написанных на Java, хорошие результаты на больших приложениях показывает утилита Chord [27] — она осуществляет многофазный анализ, сокращая объем исследуемых данных от фазы к фазе. Кроме того, существует ряд средств, основанных на формальной проверке модели программы — наиболее известна утилита Java PathFinder [18], получившая поддержку у NASA.

В общем случае невозможно точно определить множество всех возможных чередований выполнения команд в потоках. Это приводит к рассмотрению упрощенных моделей исполнения программ (для достижения приемлемой скорости анализа), что, в свою очередь, влечет обнаружение большого количества ложных гонок, которые не могут возникнуть ни в каком пути выполнения программы. Этот факт также связан с недостатком статической информации о входных данных программы и отдельных ее частей. Поэтому на практике оказываются востребованными динамические методы.

2.2. Динамические подходы к обнаружению гонок

Использование динамических методов для решения задачи обнаружения состояний гонки подразумевает сбор частичной трассы (истории обращений к памяти и синхронизационных операций) во время работы программы и ее последующий анализ. Поскольку трасса представляет собой реальный путь выполнения программы, то количество ложных состояний гонки среди обнаруженных существенно меньше, чем у анализаторов, использующих статические методы. С другой стороны, анализу подвергаются не все пути выполнения программы, а лишь те, которые соответствуют ее конкретным запускам, поэтому динамические методы не могут гарантировать отсутствие состояний гонки в программе. Кроме того, динамические методы, частично встраиваясь в сами программы (сбор трасс), увеличивают накладные расходы на время ее выполнения и размер используемой памяти. Все это нужно держать в разумных пределах, при этом собирая достаточно информации о трассах.

Динамические детекторы гонок можно разделить на две категории, в зависимости от того, допускают они ложные срабатывания или нет. *Точные* детекторы гонок никогда не производят ложных срабатываний, поскольку они вычисляют точное представление отношения «happens-before» для наблюдаемой трассы программы и сообщают об ошибке только в том случае, если трасса содержит состояние гонки. Как правило, отношение «happens-before» вычисляется с помощью *векторных часов* [24], например, как в детекторе DJIT+ [33, 34]. 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 по ходу выполнения программы. Каждый поток хранит свою локальную копию векторных часов, синхронизируясь с копиями часов других потоков во время синхронизационных операций. Но векторные часы «дороги», потому что они хранят информацию о каждом потоке в системе: если в программе n потоков, то каждая операция над векторными часами требует $O(n)$ времени, а для хранения часов нужно $O(n)$ памяти. По этой причине было разработано множество *неточных* детекторов, которые обеспечивают лучшую производительность, но могут производить ложные срабатывания — сообщать об несуществующих ошибках.

Например, алгоритм LockSet, использующийся в детекторе Eraser [37], проверяет соответствие программы определенным правилам блокировки и сообщает об ошибке, если нужная блокировка не была соответствующим образом захвачена при обращении к некоторому фрагменту памяти. Однако Eraser зачастую выдает ложные срабатывания — в частности, на программах, которые используют альтернативные механизмы синхронизации, такие как fork-join или барьерную синхронизацию. Некоторые детекторы гонок на основе алгоритма LockSet вычисляют отношение «happens-before» для улучшения точности в таких ситуациях [31, 40]. В подходе MultiRace [33, 34] использована комбинация этих методов для увеличения производительности.

Последние исследования показывают [14], что при отслеживании отношения «happens-before», в подавляющем большинстве случаев можно снизить расходы на векторные часы с $O(n)$, где n —

число потоков в программе, до $O(1)$, что в грубом приближении уравнивает производительность неточных и точных детекторов и делает последние более привлекательными для использования ввиду отсутствия ложных срабатываний. Ниже мы приведем описание точного алгоритма DJIT+, взятого за основу в данной работе.

2.3. Алгоритм DJIT+

В алгоритме DJIT+ для каждого потока хранится копия глобальных векторных часов, соответствующая состоянию системы с его точки зрения. На множестве векторных часов заданы отношение частичного порядка и операция загрузки одних часов в другие:

$$(VC_1 < VC_2) \equiv (\forall t\ VC_1(t) \leq VC_2(t)\ \&\ \exists t'\ VC_1(t') < VC_2(t')),$$

$$(VC_1.load(VC_2)) \equiv (\forall t\ VC_1(t) = max\{VC_1(t), VC_2(t)\}),$$

где VC_1 и VC_2 — это векторные часы одной длины, а t и t' — номера потоков.

Изначально все элементы этих часов равны нулю. После каждой операции освобождения блокировки часы совершают *тик* — элемент часов, соответствующий данному потоку, увеличивается на единицу. Дополнительно хранятся векторные часы L для каждой переменной блокировки l: изначально все элементы этих часов также равны нулю, обновление часов происходит при выполнении синхронизационных операций: при захвате потоком блокировки часы переменной блокировки загружаются в часы потока t, а при отпускании блокировки — наоборот, часы потока t загружаются в них. Таким образом, отслеживается отношение «happens-before»: если события A потока T_1 и B потока T_2 упорядочены этим отношением, то и векторные часы VC_1 потока T_1 в момент наступления события A и векторные часы VC_2 потока T_2 в момент наступления события B будут упорядочены определенным выше отношением на множестве векторных часов, т. е.:

$$VC_1(A) < VC_2(B).$$

Это неравенство справедливо, поскольку существует переменная блокировки l, такая, что сначала часы VC_1 первого потока были загружены в ее часы, а потом ее часы были загружены в часы VC_2 второго потока. На рис. 2 показан пример для двух потоков: пусть в программе два потока, и на момент захвата блокировки первым потоком его часы равны $[x_1, y_1]$, а часы второго — $[x_2, y_2]$. Тогда непосредственно перед отпусканием блокировки первым потоком его часы «тикнут» (увеличат свое значение на 1) и будут загружены в часы переменной блокировки. Теперь, когда блокировку захватит второй поток, он загрузит часы переменной блокировки в свои часы и, таким образом, получит обновленное значение компоненты часов для первого потока $(x_1 + 1)$. Аналогично, после того как он отпустит блокировку, его часы «тикнут» и будут загружены в часы переменной блокировки, откуда их загрузит первый поток, когда снова захватит блокировку, и получит обновленную компоненту часов второго потока $(y_2' + 1)$.

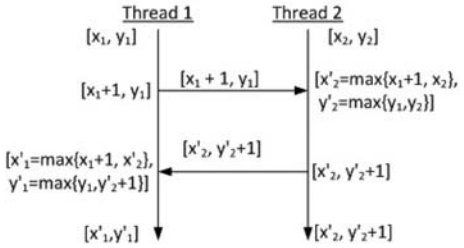

Рис. 2. Изменения векторных часов при взаимодействии потоков

Для обнаружения конкурирующих обращений к разделяемым переменным алгоритм DJIT+ хранит векторные часы VC_X для каждой такой переменной X: для каждого потока эти часы хранят часы последнего обращения потока к этой переменной. Обращение потока T к переменной не создает гонки, если происходит после предыдущего обращения к ней:

$$VC_T > VC_X.$$

Ниже показано, как алгоритм будет работать на свободном от гонок

фрагменте программы (рис. 3), и как он будет работать на фрагменте, содержащем гонку (рис. 4).

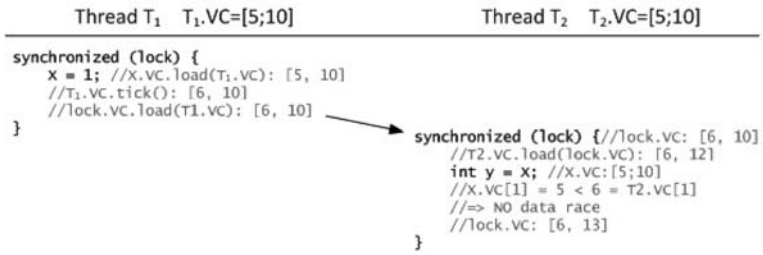

Рис. 3. Работы алгоритма при отсутствии гонок

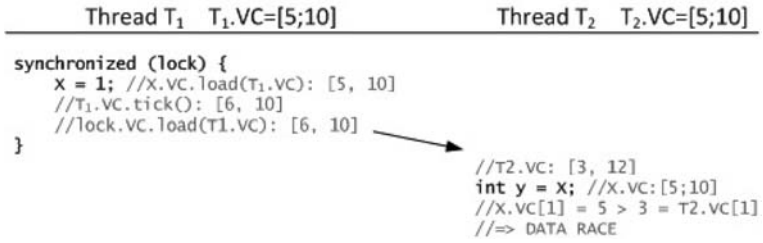

Рис. 4. Пример обнаружения гонки

3. Адаптация алгоритма DJIT+ к Java-программам

В языке Java, согласно Java Language Specification [17], отношение «happens-before» (предшествования) устанавливается следующим образом:

- освобождение монитора всегда предшествует его последующим захватам;
- запись volatile-переменной всегда предшествует ее последующему чтению;
- запуск потока всегда предшествует первому действию в потоке;

- последнее действие потока T_1 всегда предшествует любому действию потока T_2, который *знает*, что поток T_1 остановился (с помощью `Thread.join()` или `Thread.isAlive()`);
- если поток T_1 прерывает поток T_2, прерывание «happens-before» любого действия в любом потоке, который *знает*, что поток T_1 был прерван (поймал `InterruptedException` или вызвал `Thread.isInterrupted()`);
- запись значения по умолчанию каждой переменной всегда предшествует первому действию любого потока;
- действия в одном потоке, произошедшие одно после другого, находятся в отношении «happens-before»;
- отношение «happens-before» транзитивно замкнуто.

Для вычисления этого отношения и отслеживания обращений к разделяемым переменным так, как того требует алгоритм DJIT+, необходимо внедриться в процесс выполнения программы. Для Java наиболее рационально и эффективно это можно сделать с помощью инструментирования байт-кода. Во-первых, как известно, в конечный комплект поставки Java-приложений/библиотек гарантированно входит лишь байт-код class-файлов (как правило, упакованных в jar-архив), в то время как исходный код не включается фактически никогда. Во-вторых, байт-код, будучи достаточно высокоуровневым, специфицирован и структурирован существенно строже, чем сам язык программирования Java.

Существует ряд библиотек, позволяющих читать и изменять class-файлы непосредственно из Java. В описываемой реализации мы использовали open-source библиотеку ASM [2] версии 3.0, позволяющую обрабатывать class-файлы с помощью шаблона Visitor.

Однако инструментирование имеет ряд ограничений — не все операции, фигурирующие в определении отношения «happens-before», удается перехватить в клиентском коде, не затрагивая сторонние библиотеки или JRE. Поэтому мы построим упрощенную

Например, не удается гарантированно передать векторные часы завершившегося потока T_1 потоку T_2, ожидавшему его завершения посредством `T1.join()`. Управление к потоку T_2 переходит уже после завершения T_1, и часы потока T_1, хранившиеся в `ThreadLocal`, оказываются недоступными к этому моменту. Кроме того, не удастся отследить синхронизацию в стороннем коде — например, через `SwingUtilities.invokeLater()`.

модель, в которой отразим лишь те операции, которые можно эффективно перехватить. Остальных операций и иных возникающих проблем мы коснемся в следующем разделе, посвященном анализу созданного нами детектора гонок.

Адаптированный нами алгоритм DJIT+ будет работать на модели Java-программы, основанной на следующем множестве векторных часов:

- T — векторные часы потоков;
- L — векторные часы переменных, по которым осуществляется синхронизация;
- F — множество векторных часов всех разделяемых полей — тех, к которым в программе существует доступ из нескольких разных потоков, один из которых — доступ на запись.

Состояние модели определяется как состояние всех используемых векторных часов. Состояния будут обозначаться индексированными или штрихованными маленькими буквами `s`. Обращение к элементам состояния `s` будет обозначаться с помощью точки. Например, $\texttt{s.t}_\texttt{i}\texttt{.vc[j]}$ — индекс, соответствующий потоку, у которого `tid==j` в векторных часах потока с `tid==i` в состоянии модели `s`.

В табл. 1 приведены переходы модели для интересующих нас событий в программе.

Таблица 1. **Операции модели, соответствующие операциям программы**

Операция программы	**Операция модели**
`t locks l`	`s.t.vc.load(s.l.vc);`
`t unlocks l`	`s.t.vc.tick();s.l.load(s.t.vc)`
`t returns from wait on l`	`s.t.vc.load(s.o.vc)`
`t starts` t_1	t_1`.vc.load(t.vc)`
`t access f`	`s.f.vc.load(s.t.vc)`

Точки программы, в которых нужно перехватывать выполнение, можно разделить на два вида — синхронизационные операции

Под операцией `t.tick()` понимается увеличение на единицу компоненты векторных часов потока, соответствующей этому потоку. Например, `t1.tick()` увеличит первую компоненту векторных часов `t1` на единицу.

и операции обращения к разделяемым переменным. Синхронизационным операциям типа synchronized соответствуют две инструкции JVM (Java Virtual Machine): `MONITORENTER` (захват монитора) и `MONITOREXIT` (освобождение монитора). Соответственно, эти две инструкции нужно инструментировать. В случае Java-методов, объявленных как synchronized, нужно перехватывать, во-первых, первую инструкцию метода, и, во-вторых, инструкции выхода из метода — `RETURN` или `ATHROW`.

Немного сложнее дело обстоит с синхронизационными операциями, предоставляемыми Java в виде методов и с обращением к volatile-полям. Для вызова нестатических методов классов в JVM существует две инструкции: `INVOKESPECIAL` и `INVOKEVIRTUAL`. Соответственно, нужно перехватывать их все и проверять, не является ли вызванный метод методом `Object.wait()`. Отметим, что достаточно проверить класс-владелец метода, имя метода и его сигнатуру, поскольку метод `Object.wait()` объявлен как `final` и не может быть переопределен в потомках.

В случае с volatile-переменными нужно перехватывать все обращения к глобальным разделяемым переменным (четыре инструкции: `GETFIELD`, `PUTFIELD`, `GETSTATIC`, `PUTSTATIC`) и проверять, не имеет ли переменная, к которой произошло обращение, модификатора `volatile`. Обращения к локальным переменные проверять не нужно — они не могут иметь модификатора `volatile` согласно спецификации Java.

Для передачи часов потока-предка потоку-потомку достаточно перехватить вызов `Thread.start()` (инструкция `INVOKEVIRTUAL`) и сохранить часы предка в специальном поле, из которого поток-потомок заберет их при инициализации собственных часов.

Для хранения векторных часов потоков мы использовали класс Java API `java.lang.ThreadLocal`, решающий задачу хранения отдельных наборов однотипных данных для каждого потока и предоставляющий достаточно удобный интерфейс для их получения и изменения. В случае хранения векторных часов для разделяемых полей, средств, подобных `ThreadLocal`, в Java API нет, поэтому пришлось хранить часы вместе с полями:

- для каждого глобального поля заводится «обертка» (box), сохраняющая исходное поле и векторные часы, связанные с ним;

- перехватываются инструкции обращения к глобальным полям и явно, с помощью инструментирования, делегируются их «оберткам»;
- при обращении к «обертке» производится проверка, была ли уже создана эта обертка, если не была — создается и сохраняется с помощью инструкции `PUTFIELD`, иначе идет обращение к векторным часам.

Для каждого класса <`package.path`>`.T` (в том числе и для массива), такого, что в программе существует глобальное поле типа `T`, создается класс `$box.`<`package.path`>`.T`, поле типа `T` заменяется на ссылку на «обертку», и все операции делегируются этой «обертке».

«Обертки» удобно создавать отдельным проходом по class-файлу — при первом проходе создаются «обертки», при втором инструментируется байт-код. Это вполне подходит для решения локальной задачи, более того, применительно к массивам оно подразумевает создание отдельной обертки как для массива, так и для каждого его элемента. Это оправдано, поскольку и массив, и каждый его отдельный элемент могут являться независимыми объектами, разделяемыми разными потоками. Однако с массивами связана следующая проблема, решение которой выходит за рамки данной работы. Java API предоставляет для массивов много различных методов копирования (например, `System.arrayCopy()` или `Arrays.asList(T...a)`), а также методов, которые представляют коллекции в виде массива (например, `List.toArray()`). При вызове этих методов необходимо соответствующим образом копировать значения векторных часов, что труднореализуемо, поскольку полный список таких методов не специфицирован.

Рассмотрим еще один тонкий момент, связанный с синхронизацией обращений к векторным часам. Часы потоков не требуют синхронизации обращения к ним, поскольку доступ к часам конкретного потока имеет только этот поток. Но к часам разделяемых/синхронизационных переменных обращаться могут разные потоки, поэтому потребуется некоторая «синхронизированность» действий с ними. Фактически, подойдет использование любых примитивов синхронизации, поскольку хотя это и добавит «синхронизированности» в исходную программу, добавятся дополнительные

точки синхронизации памяти потоков с общей памятью программы. Однако с точки зрения алгоритма эти обращения абсолютно прозрачны, поскольку они не будут инструментированы, поэтому разумно использовать самый удобный в реализации вариант — синхронизацию с помощью ключевого слова `synchronized`.

Векторные часы синхронизационных переменных имеют концептуально иную природу, по сравнению с часами разделяемых переменных. Поэтому их целесообразно хранить отдельно (более того, это необходимо, поскольку установить взаимосвязь между полем, по которому вызывается инструкция `MONITORENTER`, и его box-ом невозможно). Для их хранения мы использовали класс `java.util.HashMap` из множества ссылок на переменные в множество соответствующих им векторных часов. Операции обновления векторных часов из потока защищены той синхронизационной операцией, в рамках которой происходит обращение к часам. Единственный момент, требующий дополнительной синхронизации — это первичная инициализация часов. Для обеспечения ее защищенности в аспекте многопоточности была выделена отдельная lock-переменная.

В табл. 2 подведен итог описанного выше — какие инструкции JVM нужно инструментировать и какие операции нужно выполнять, встретив в исходном коде соответствующую инструкцию. Все операции, проверяющие возникновение состояния гонки, вставляются *перед* встреченной инструкцией, что гарантирует немедленное обнаружение состояния гонки.

3.1. Тестирование и апробация

В табл. 3 приведены сравнительные результаты по производительности для предложенной реализации (DRD-утилита), а также для двух существующих популярных динамических детекторов гонок для Java-приложений — mtrat [26] и Goldilocks [12]. В столбце «Исходное время работы» указано время, полученное тестирующей программой для неинструментированного кода, в остальных столбцах — замедление инструментированного когда по отношению к неинструментированному. Проверка проводилась на стандартных многопоточных тестах из набора «Java Grande Forum Benchmark» [16].

Таблица 2. **Инструкции JVM, требующие инструментирования**

Инструкция JVM	**Условие**	**Операция**
GETFIELDGETSTATIC	volatile поле	t.vc.load(v.vc)
	Иначе	f.vc.load(t.vc) с проверкой на возникновение состояния гонки
PUTFIELDPUTSTATIC	volatile поле	t.vc.tick();v.vc.load(t.vc)
	Иначе	f.vc.load(t.vc) с проверкой на возникновение состояния гонки
INVOKESPECIAL INVOKEVIRTUAL	Thread.start() Thread.<init>	Сохранение часов предка, инициализация векторных часов потока из часов предка
	Object.wait() Object.wait(0)	t.vc.load(o.vc)
*RETURN ATHROW	метод объявлен как synchronized	⇔ MONITOREXIT по полю this/<ClassName>.class
MONITORENTER		t.load(l.vc)
MONITOREXIT		t.vc.tick(); l.load(t.vc)
первая инструкция метода	метод объявлен как synchronized	⇔ MONITORENTER по полю this/<ClassName>.class

Таблица 3. **Сравнение результатов тестирования различных динамических утилит**

Тест/Утилита	**Исходное время работы, сек**	**DRD**	**mtrat**	**Goldilocks**
Tsp	0.672	13.15	16.87	2.2
Moldyn	1.453	34.85	2.54	5.4
Montecarlo	3.8	4.13	1.4	2.2
Raytracer	2.93	42.7	26.89	17.9
Lufact	0.219	1.14	–	4.1

На ряде тестов (Lufact, Montecarlo) наша реализация показала достаточно конкурентоспособные результаты; на прочих — достаточно низкие, что обусловлено тем фактом, что тесты реализованы, фактически, полностью через глобальные переменные (как следствие, используется множество инструкций GETFIELD и SETFIELD). В общем же, результаты соизмеримы с аналогичными результатами mtrat и существенно хуже результатов Goldilocks. Последнее связано с тем, что алгоритм Goldilocks напрямую интегрирован в Kaffe VM, в то время как DRD инструментирует байт-код.

Также мы запускали наш детектор на крупных приложениях

(порядка 1000 классов), на которых он показал достаточно хорошие результаты — коэффициент замедления не превышал 1.4, что тоже сопоставимо с результатами, которые демонстрируют аналоги. Полученные результаты показывают применимость разработанной системы для автоматического интеграционного тестирования, однако очевидно, что для автоматического нагрузочного тестирования подобная производительность будет недостаточной. Кроме того, неизбежно возникнут проблемы с ростом размера системы — на больших приложениях все динамические детекторы дают очень сильное замедление из-за экспоненциального роста возможного числа путей выполнения программы при увеличении числа потоков и разделяемых объектов в программе. В следующем разделе мы сформулируем требования, которым должен удовлетворять детектор, который планируется использовать на больших нагруженных системах.

4. Требования к промышленному детектору

В этом разделе мы обсудим те проблемы, которые необходимо решить для того, чтобы реализовать эффективный динамический детектор гонок для Java-приложений, который бы смог работать на больших промышленных приложениях

Как уже упоминалось ранее, не все примитивы синхронизации, предоставляемые языком Java, могут быть эффективно перехвачены при помощи инструментации клиентского байт-кода. На самом деле, проблема более глубока: вызов клиентского кода может использоваться из стороннего потока, при этом в действительности гонок не будет, потому что этот поток будет синхронизирован с остальными потоками также в стороннем коде. В качестве примера можно привести интерфейсы стандартного пакета java.util.concurrent (Executor, Lock) и методы `invokeLater()` и `invokeAndWait()` класса `SwingUtilities`, входящие в состав JRE. Кроме того, подобным образом устроены все синхронизационные механизмы, предоставляемые java.util.concurrent. Клиент между обращениями к переменным вызывает соответствующие методы правильным образом в нужном порядке, за счет чего достигается упорядоченность этих обращений к переменным отношением

«happens-before». Но детектор не анализирует клиентский код (размер используемых библиотек вместе с размером JRE может в тысячи раз превышать размер приложения) и поэтому не имеет информации о подобной синхронизации.

Эту проблему можно решить введением возможности *«вручную» указать дополнительные точки синхронизации* — пары вызовов методов, которые форсируют отношение «happens-before». Для решения подобных задач существуют различные аспектно-ориентированные программные подсистемы [3, 4, 36] — они позволяют указать любую точку программы (для описания этих точек предоставляется мощный внутренний конфигурационный язык) и перехватить управление в ней. Однако в нашем случае подобный прямой подход не допустим ввиду сильных потерь в производительности, поэтому задача нахождения компромисса между гибкостью и скоростью является актуальной. Кроме того, проблема производительности неизбежно возникнет при попытках применить детектор на нагруженных системах — даже при минимальных накладных расходах на используемую память и близкой к константе скорости операций над векторными часами наличие множества потоков и гораздо большего количества разделяемых полей сделает задержки неприемлемыми. Поэтому помимо оптимизации алгоритма необходимо иметь возможность *явно указать множество объектов для анализа*.

Наконец, нужно предпринять шаги в направлении решения главной проблемы динамических детекторов — невозможности доказать с их помощью отсутствие гонок в программе. Похожая проблема связана с тестированием — никакой набор тестов не доказывает отсутствие ошибок в программе. Однако качество тестирования часто определяют как степень покрытия программы тестами. Похожий подход видится возможным и в нашем случае — можно ввести *счетчик обработанных трасс* и на его основании давать прогнозы по качеству программы в смысле отсутствия в ней гонок.

Следует также упомянуть проблему динамического инструментирования байт-кода (инструментирование «на лету» с помощью собственного загрузчика классов), а также мониторинг производительности с помощью Java-агента — без него невозможно точно оценить накладные расходы инструментации и работы алгоритма. По-

лученные оценки можно будет сравнить с теоретическими и предложить дальнейшие пути оптимизации.

Заключение

Состояния гонки в параллельных программах трудны в обнаружении. Точные детекторы не допускают ложных срабатываний, что позволяет использовать их в автоматическом тестировании. Но они имеют ряд ограничений по производительности, что препятствует применению их на нагруженных системах. Мы адаптировали и реализовали известный точный алгоритм DJIT+, основанный на векторных часах, для Java и предложили ряд возможных решений некоторых открытых вопросов реальной применимости точных детекторов. В дальнейшем планируется реализовать сформулированные требования и доработать предложенный детектор с тем, чтобы он был применим к крупным Java-проектам. Мы планируем оценить его работу на крупных open-source приложениях под нагрузкой. Мы полагаем, что детектор, удовлетворяющий перечисленным выше требованиям, продемонстрирует удовлетворительное сочетание производительности и точности.

Список литературы

[1] *Agarwal R. e.a.* Optimized Run-Time Race Detection and Atomicity Checking Using Partial Discovered Types. In Proceedings of The 20th IEEE/ACM international Conference on Automated Software Engineering. 2005. P. 233–242.

[2] ASM Java Bytecode Manipulation and Analysis Framework. http://asm.ow2.org/

[3] AspectJ Project. http://www.eclipse.org/aspectj/

[4] AspectWerkz AOP Framework for Java. http://aspectwerkz.codehaus.org/

[5] *Boyapati C., Rinard M.* A Parameterized Type System for Race-Free Java Programs. In Proceedings of The 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. 2001. P. 56–69.

[6] *Boyapati C., Lee R., Rinard M.* Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In Proceedings of The 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. 2002. P. 211–230.

[7] *Cheng G., e.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.

[8] *Choi J., Lee K., Loginov A., O'Callahan R., Sarkar V., Sridharan M.* Efficient and Precise Data Race Detection for Multithreaded Object-Oriented Programs. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. 2002. P. 258–269.

[9] *Choi J., Loginov A., Sarkar V.* Static Data Race Analysis for Multithreaded Object-Oriented Programs. Technical Report, IBM Research, Report RC22146. 2001. 18 p.

[10] *Detlefs D., Leino K. R. M., Nelson G., Saxe J.* Extended Static Checking. In Proceedings of The 30th International Conference on Software Engineering. 1998. P. 211–220

[11] Documentation of java.util.concurrent Package. http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html

[12] *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. 2007. P. 245–255.

[13] *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.

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

[15] *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.

[16] Java Grande Forum Multi-Threaded Benchmarks. http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html

[17] Java Language Specification, Third Edition. Happens-Before Order. http://java.sun.com/docs/books/jls/third_edition/html/memory.html#17.4.5

[18] Java PathFinder. http://javapathfinder.sourceforge.net/

[19] Intel Threading Building Blocks. http://www.threadingbuildingblocks.org/

[20] *Henzinger T., Jhala R., Majumdar R.* Race Checking by Context Inference. In Proceedings of The ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation. 2004. P. 1–13.

[21] *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. 127–138.

[22] *Leino K. M., Nelson G., Saxe J.* ESC/Java User's Manual. SRC Technical Note 2000-002. 2001. 86 p.

[23] *Leveson N., Turner C. S.* An Investigation of The Therac-25 Accidents. In IEEE Computer, Vol. 26, N 7. 1993. P. 18–41.

[24] *Mattern F.* Virtual Time and Global States of Distributed Systems. In Proceedings of The International Workshop on Parallel and Distributed Algorithms. 1989. P. 215–226.

[25] *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.

[26] Multi-Thread Run-Time Analysis Tool for Java. http://www.alphaworks.ibm.com/tech/mtrat

[27] *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.

[28] *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.

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

[30] *Netzer R., Miller B.* Improving the Accuracy of Data Race Detection. In Proceedings of the 1991 Conference on The Principles and Practice of Parallel Programming. 1991. P. 133–144.

[31] *O'Callahan R., Choi J.-D.* Hybrid Dynamic Data Race Detection. In PPOPP. 2003. P. 167–178.

[32] *Perkovic D., Keleher P.* Online Data-Race Detection via Coherency Guarantees. In Proceedings of The Second USENIX Symposium on Operating Systems Design and Implementation. 1996. P. 47–57.

[33] *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.

[34] *Pozniansky E., Schuster A.* MultiRace: Efficient on-the-fly Data Race Detection in Multithreaded C++ Programs. Concurrency and Computation: Practice and Experience, 19(3). 2007. P. 327–340.

[35] *Praun C., Gross T.* Object Race Detection. In ACM SIGPLAN Notices, Vol. 36, Issue 11. 2001. P. 70–82.

[36] *Safonov V. O.* Using Aspect-Oriented Programming for Trustworthy Software Development. Wiley Interscience, John Wiley & Sons, Inc. 2008. 338 p.

[37] *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.

[38] *Schonberg E.* On-the-fly Detection of Access Anomalies. In Proceedings of the Symposium on Interpreters and Interpretive Techniques (SIGPLAN'87). 1987. P. 285–297.

[39] *Sterling N.* WARLOCK — A Static Data Race Analysis Tool. In Proceedings of The Usenix Winter 1993 Technical Conference. 1993. P. 97–106.

[40] *Yu Y., Rodeheffer T., Chen W.* RaceTrack: Effcient Detection of Data Race Conditions via Adaptive Tracking. In SOSP. 2005. P. 221–234.