Вопросы индустриального применения синхронизационных контрактов при динамическом поиске гонок в Java-программах

В.Ю. Трифанов <vitaly.trifanov@gmail.com>,
Санкт-Петербургский государственный университет,
198504, Россия, Санкт-Петербург, Университетский пр., д. 28

Аннотация. Состояния гонки (data race) возникает в многопоточной программе при одновременном обращении нескольких потоков к разделяемым данным. Существует два основных подхода к обнаружению гонок – статический анализ программы (без её запуска) и динамическое обнаружение гонок в процессе работы программы. Ранее авторами был предложен точный высокопроизводительный динамический подход к обнаружению гонок на основании специальным образом составленных синхронизационных контрактов – частичных спецификаций поведения классов и наборов методов целевого приложения в многопоточной среде. Данная статья рассматривает вопрос индустриального применения концепции синхронизационных контрактов на крупных нагруженных многопоточных приложениях. Предложены метод обработки контрактов и архитектура соответствующего модуля динамического детектора jDRD, выявлены основные проблемные места и потенциальные точки падения производительности, разработано техническое решение, лишённое подобных проблем.

Ключевые слова: состояние гонки; многопоточность; динамический анализ; автоматическое обнаружение ошибок.

DOI: 10.15514/ISPRAS-2018-30(3)-4

Для цитирования: Трифанов В.Ю. Вопросы индустриального применения синхронизационных контрактов при динамическом поиске гонок в Java-программах. Труды ИСП РАН, том 30, вып. 2, 2018 г., стр. 47-62. DOI: 10.15514/ISPRAS-2018-30(3)-4

1. Введение

Многопроцессорные и многоядерные системы начали своё развитие в конце XX века и на текущий момент фактически вытеснили однопроцессорные вычислительные машины. Основным преимуществом подобных систем является возможность одновременного выполнения различных последовательностей инструкций параллельно. Наиболее популярной

программной архитектурой таких систем является модель разделяемой памяти, в которой несколько потоков управления обмениваются данными через общую память. Такая архитектура лежит в основе таких широко распространённых в индустрии языков как C++ и Java [1].

Организация корректного взаимодействия нескольких потоков является одной из самых сложных задач в программировании, здесь возможны серьёзные ошибки, такие как взаимные блокировки (deadlocks), голодание (thread starvation) и гонки (data races).

Гонка (или состояние гонки) возникает в многопоточной программе, когда несколько потоков одновременно обращаются к разделяемым данным, и хотя бы одно из этих обращений является записью данных [2]. Основная опасность гонок заключается в том, что они приводят к повреждению глобальных структур данных, но программа при этом зачастую продолжает работу, что приводит к сбоям и непредвиденным результатам. Ущерб от гонок может приводить к огромным финансовым потерям и к человеческим жертвам [3, 4]. Возникновение гонки зависит от чередования операций в потоках. При внешнем управлении потоками (а именно так устроено подавляющее большинство систем) переключение между ними происходит непредсказуемо, что затрудняет возможность воспроизведения гонок на стадии тестирования. Таким образом, задача автоматического обнаружения гонок актуальна, важна и находится в поле внимания исследователей на протяжении нескольких десятилетий.

Принято выделять два основных подхода к автоматическому обнаружению гонок. Первый – это статический подход, который предполагает анализ исходного кода программы без её запуска. Например, возможно построение графа выполнения программы (control flow graph, CFG) и расчёт множества удерживаемых потоками блокировок. Такой алгоритм называется lockset [5, 6] и обладает существенным недостатком – с его помощью можно отслеживать только операции синхронизации, основанные на блокировках. В современном программировании это становится все менее актуально, поскольку разработаны неблокирующие операции синхронизации и типовые структуры данных, такие как очередь и хеш-таблица [7]. В простых случаях можно успешно доказать корректность программ без блокировок с помощью тяжеловесных методов – например, алгоритма Деккера или Петерсона. Анализ использования объектов и контекста [8-11] также позволяет корректно обрабатывать некоторые другие способы синхронизации. Также возможен анализ с помощью проверки моделей [12-13] или использования дополнительных типов [14] и их вывода с помощью аннотаций [15-16].

Второй подход – динамический, в рамках которого анализ программы осуществляется во время её выполнения [17-24]. Он анализирует лишь текущий путь выполнения программы, но в нём может обрабатывать любые операции синхронизации и обладать стопроцентной точностью, без ложных срабатываний 1-го и 2-го рода. Однако на практике его точность сильно

ограничена производительностью, поскольку обработка всех операций синхронизации и обращений к разделяемым данным требует в сотни раз больше ресурсов, чем выполнение самой анализируемой программы. Следует отметить, что последние разработки в сфере динамического поиска гонок сводятся к сокращению области анализа с помощью различных техник, в том или ином виде жертвующих точностью анализа [25-26].

Ранее авторами был предложен подход *синхронизационных контрактов*, который позволяет существенно повысить производительность динамического обнаружения гонок без потери точности [27-29]. Этот подход показал хорошие практические результаты, но в процессе его индустриального применения обнаружился ряд трудностей. Во-первых, созданные разработчиками контракты могут содержать неточности и ошибки, а во-вторых, контракты нужно эффективно применять в процессе анализа кода, чтобы достичь экономии накладных расходов на сбор информации и проверку гонок во время работы приложения.

В данной статье предложен подход к анализу, верификации и динамическому применению контрактов, основанный на построении дерева контрактов (аналог графа потока управления), и продемонстрирована его целесообразность и практическая полезность.

2. Синхронизационные контракты и детектор jDRD

Основным точным алгоритмом динамического обнаружения гонок в программах, основанных на модели разделяемой памяти, является алгоритм happens-before [18, 30], являющийся, фактически, поиском гонок «по определению». Алгоритм, анализируя операции в программе, выделяется два подмножества: (i) операции с данными внутри потоков исполнения и (ii) операции синхронизации, передающие изменения между потоками и таким образом синхронизирующие данные различных потоков.

Рассмотрим работу алгоритма на примере языка Java, у которого, наряду с C/C++, модель памяти максимально проработана [1]. На множестве всех операций синхронизации существует отношение полного порядка synchronized-with. Все операции внутри каждого конкретного потока также упорядочены – это порядок по времени. Объединение и последующее транзитивное замыкание этих двух отношений даёт отношение частичного порядка, называемое happens-before. Согласно определению гонки, две операции с одними и теми же данными находятся в состоянии гонки, если они не упорядочены с помощью отношения happens-before [1].

Традиционно выполнение отношения happens-before отслеживается с помощью векторных часов Лампорта [31]. Такой подход точен, но обладает очень большими накладными расходами – скорость работы программы замедляется в 10-200 раз [32]. Для повышения производительности ранее авторами был предложен подход синхронизационных контрактов, в основе которого лежит два наблюдения.

1. Индустриальные приложения используют сторонние библиотеки и подсистемы, которые в совокупности могут превышать размер кода самого приложения в несколько десятков раз. Как правило, взаимодействие со сторонними библиотеками осуществляется через хорошо документированные интерфейсы. В частности, обычно хорошо документировано поведение методов, классов и интерфейсов в многопоточной среде.
2. При поиске гонок обычно стоит задача обнаружения ошибок в собственном коде приложения, поскольку выбор библиотек обычно осуществляется экспертами на основании больших объёмов данных об их предыдущем использовании. Это позволяет сделать вывод о существенно меньшей надёжности самого разрабатываемого приложения, чем используемых им библиотек и подсистем, и сфокусироваться на анализе собственного кода приложения.

На основе этих наблюдений был разработан метод синхронизационных контрактов, а также динамический детектор jDRD [27-29, 33].

Основная идея метода заключается в том, чтобы разделить весь программный код целевого приложения на две части – подлежащую анализу (обычно это собственный код приложения) и не подлежащую анализу, то есть считающуюся надёжной (обычно это сторонние библиотеки, подсистемы и модули). Далее необходимо определить интерфейсы и методы, посредством которых анализируемая часть взаимодействует с неанализируемой частью, и описать их поведение в многопоточной среде. Иными словами, нужно выделить используемые в приложении интерфейсы других компонент, проанализировать и классифицировать их классы и методы там в тех случаях, где про них что-то известно. В [33] предлагается следующая классификация:

1. вызов пары методов (или несколько пар методов одного класса) обеспечивают передачу отношения happens-before согласно документации;
2. метод или класс потокобезопасен, но не вовлечён в передачу отношения happens-before;
3. метод объекта не предназначен для вызова в многопоточной среде (требует внешней синхронизации), но может рассматриваться как немодифицирующий состояние объекта-владельца (то есть, может рассматриваться как операция чтения данных).

Единичное описание принадлежности метода (или всех методов класса) к какой-то категории и называется синхронизационным контрактом. Контракт для пары методов, обеспечивающих передачу отношения happens-before, называется happens-before контрактом. Точность и ограничения такого подхода описаны в работе [28], там же представлены основные виды контрактов и продемонстрировано повышение производительности динамического анализатора, увеличивающееся с сокращением анализируемой области и

ростом базы контрактов. В последующей статье [29] представлен язык описания контрактов, позволяющий разрабатывать синхронизационные контракты отдельных методов, пар методов или классов. Кроме того, язык содержит директивы, позволяющие включить контракты из других файлов. Это обеспечивает переиспользуемость контрактов – достаточно описать один раз контракты библиотеки и внести файл с ними в комплект поставки. На листинге 1 представлен пример контракта, указывающего наличие отношения happens-before между записью в потокобезопасную хеш-таблицу o по некоторому ключу p1 и последующими чтениями из неё по тому же ключу.

```
sync {
        key java.lang.Object=o, java.lang.Object=p1;
        send
java.lang.Objectjava.util.concurrent.ConcurrentMap.put(java.lang.Object,
java.lang.Object);
        receive
java.lang.Objectjava.util.concurrent.ConcurrentMap.get(java.lang.Object);
}
```

Листинг 1. Пример контракта
Listing 1. Contract example

В дальнейшем эти контракты используются на фазе динамического анализа: перед вызовом метода детектор определяет, есть ли контракт для этого метода, и если есть, то обрабатывает метод в соответствии с этим контрактом, что и даёт прирост производительности (в противном случае пришлось бы проводить анализ всего кода метода).

В остальном схема устройства jDRD достаточно стандартна. На фазе запуска анализируемого приложения к нему посредством стандартной технологии java-agent подключается модуль jDRD, который модифицирует загружаемые классы с помощью техники инструментирования байт-кода. В код классов вставляются инструкции для обработки операций синхронизации и обращений к разделяемым данным. Во время выполнения таких операций управление передаётся в jDRD, который динамически обсчитывает векторные часы и обнаруживает гонки.

3. Проверка корректности контрактов

На этапе индустриального внедрения описанной выше технологии возник ряд сложностей, требующих правильных архитектурных и технических решений. В основной массе они связаны с тем, что в общем случае контракты создаются на основе различных источников (конфигурационные файлы, аннотации в коде, документация и т.д.), и зачастую различными специалистами. Это неизбежно приводит к возможности различных ошибок – например, неполноте, противоречивости или некорректности совокупного набора контрактов – и поэтому после объединения всех контрактов воедино необходимо верифицировать полученный набор. Выходными данными модуля обработки

контрактов является изменённый код целевого приложения (инструментированный байт-код), учитывающий контракты при обработке операций в приложении. Таким образом, задача обработки контрактов состоит из двух следующих частей (процедур).

1. Загрузка, проверка корректности и размещение контрактов в памяти.
2. Применение контрактов и модификация кода целевого приложения.

Первая процедура осуществляется один раз при запуске приложения. Нет ограничений на производительность этой процедуры, её основной задачей является проверка корректности контрактов. Вторая процедура является критичной с точки зрения производительности и требует тщательной реализации: контракты будут применяться постоянно (сотни раз в секунду) на протяжении всего временя работы приложения.

Рассмотрим подробнее первую процедуру. Язык описания контрактов [29] предоставляет синтаксические директивы для спецификации контрактов, поэтому на стадии чтения этих контрактов необходимо провести соответствующие процедуры по их разбору и проверке корректности. Здесь возможны следующие ситуации.

1. Контракты могут противоречить друг другу; например, в одном контракте указано, что метод A синхронизирован с B, а во втором – что, наоборот, B синхронизирован с A; в подобных случая необходимо сигнализировать об ошибке.
2. Контракты классов могут быть неполными, т.е. описывать лишь часть публичных методов класса. В этом случае нужно выдать предупреждение.
3. Контракты могут дублироваться – в этом случае нужно убедиться в их семантической идентичности.

Для проверки непротиворечивости контрактов необходимо убедиться в отсутствии циклических контрактных зависимостей между ними. Для обнаружения таких зависимостей строится граф контрактов. Вершинами графа являются методы контрактов, а ребро между парой вершин A и B существует тогда и только тогда, когда A предшествует B. Согласно принципу подстановки Лисков [34] при наследовании потомок должен удовлетворять контракту предка. Следовательно, для каждого класса целевого приложения необходимо получить список его потомков и добавить соответствующие рёбра в граф контрактов. Наличие цикла в таком графе означает наличие цепочки противоречащих друг другу контрактов, а отсутствие циклов – непротиворечивость всего набора контрактов.

Для проверки контрактов классов на полноту на этапе загрузки контрактов анализируется каждый класс целевого приложения, для которого существует контракт, и выясняется, все ли его публичные методы упомянуты в контракте. Если это не так, то выдаётся предупреждение.

Таким образом, процедура загрузки и проверки корректности контрактов имеет следующий вид.

1. Построить граф контрактов.
2. Для каждого контракта на пару методов (f, g) классов A и B добавить в граф контракты (f', g') для всех наследников классов A и B.
3. Выполнить проверку циклов в графе.
4. (Опционально). Для всех классов, упомянутых в контрактах, проверить наличие их публичных методов, не упомянутых в контрактах.

Данная процедура реализована в виде компоненты, предварительно обрабатывающей контракты до запуска целевого приложения. Она выдаёт дерево контактов и другую информацию, используемую в дальнейших динамических проверках.

4. Архитектура модуля применения контрактов

Рассмотрим процедуру применения контрактов с точки зрения её оптимальной реализации. Точнее, опишем программный модуль, который реализует эту процедуру.

Динамический детектор должен в режиме реального времени определять наличие контракта для определённого Java-метода. При этом скорость работы детектора не должна деградировать с ростом программы, увеличением числа контрактов или иерархии классов. Обратим внимание, что речь идёт о сотнях и тысячах операций в секунду, поскольку вызовы методов в Java-программах происходят регулярно. Соответственно, решение данной задачи требует нестандартных подходов как к организации структуры данных, управляющей контрактами, так и к техническим решениям по их проверке в режиме реального времени.

В результате построения дерева контрактов для каждого метода становится известен набор контрактов, в которые он вовлечён. Поэтому во время вызова этого метода в процессе работы программы детектор jDRD может эффективно получить данную информацию. Для внедрения соответствующих инструкций в код целевого приложения используется техника инструментирования байт-кода. Механизм её работы заключается в следующем: на фазе загрузки классов целевого приложения в оперативную память, после загрузки очередного класса управление передаётся компоненте Instrumentator – специальному Java-агенту, который реализован в рамках jDRD и может модифицировать байт-код данного класса (см. рис. 1). Instrumentator анализирует список методов класса и, если для метода существуют контракты, встраивает в качестве первой инструкции метода обработку этого контракта. Во время работы модифицированное приложение автоматически обработает контракты всех методов (компонента RaceDetector). В практическом плане наличие контракта метода означает следующее.

1. Исходный код метода анализировать не нужно. Иными словами, во время выполнения тела метода детектор не должен проводить никаких операций и проверок. Для этого сразу после входа в тело метода детектор ставит специальный флаг в состояние «contract», а на выходе – сбрасывает. Перед тем, как выполнить очередную операцию, детектор проверяет состояние флага для текущего потока, и если его состояние равно «contract», то игнорирует операцию. Таким образом, флаг хранится для каждого потока отдельно. Управление флагами вынесено в отдельную компоненту (класс) FlagManager, основанный на стандартной Java-структуре ThreadLocal, которая предоставляет эффективный способ хранения локальных данных потока (см. рис. 1).
2. happens-before контракты подразумевают синхронизацию методов. Обработка этой информации реализована классическим способом – в виде векторных часов (компонента ClockStorage, см. рис. 1). Для хранения часов используется потокобезопасная хеш-таблица. Однако с учётом высокой нагрузки и большой частоты обращений традиционные потокобезопасные хеш-таблицы здесь не подходят. После ряда экспериментов была выбрана неблокирующая хеш-таблица [35]. Остаётся вопрос в выборе ключа для хранения часов. Ключ представляет собой отдельный объект, состоящий из полей, указанных в контракте. Поскольку набор полей от контракта к контракту может отличаться, классы объектов типа «ключ» генерируются автоматически, «на лету», посредством инструментирования байт-кода. На практике таких ключей нужно не более 20-30, поэтому на производительность динамического анализа это не оказывает существенного влияния. Отметим, что основная работа по анализу гонок производится в компоненте RaceDetector, которая работает только с векторными часами.

Единственным существенным недостатком описанного выше подхода является необходимость постоянного обращения к компоненте FlagManager. В частности, это регулярно происходит при работе самого jDRD, поскольку внутри него используются структуры данных, которые активно используют контракты. Одно такое обращение занимает порядка 1 мкс, но с учётом большого числа обращений (десятки-сотни тысяч в секунду) это существенно снижает производительность jDRD. Возможным решением может быть полный отказ от использования стандартных Java-классов во внутренних структурах детектора jDRD.

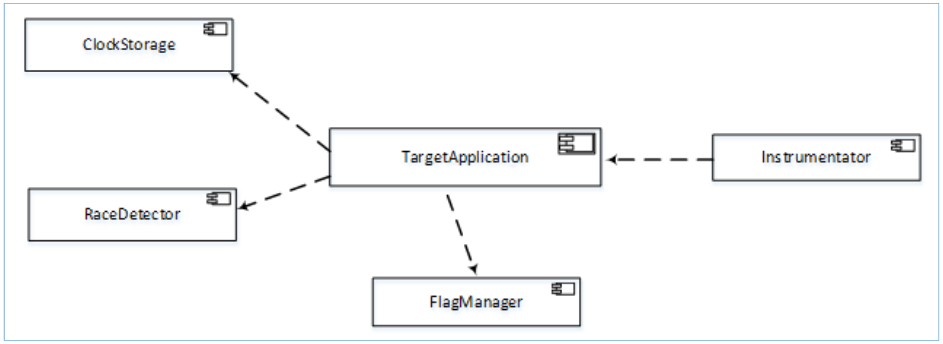

Рис.1. Архитектура модуля применения синхронизационных контрактов
Fig 1. Architecture of the contracts processing module.

5. Экспериментальное исследование

Предложенный подход был применён для анализа трёх приложений, имеющих разную специфику. В каждом содержалось 1000-2000 классов без учёта используемых библиотек, а также 10-30 потоков. Синхронизационные контракты для каждого приложения разрабатывались соответствующими командами программистов. Кроме того, использовался общий набор контрактов, разработанный ранее для стандартных средств синхронизации Java. Совокупный размер контрактов для каждого приложения не превышал долей процента от общего объёма кода приложения.

В табл. 1 представлены результаты экспериментов. В столбце «Количество контрактов» указано совокупное количество методов, для которых были составлены контракты. Столбец «количество ошибок» указывает число ошибок, обнаруженных при проверке корректности контрактов. Последний столбец содержит количество публичных методов, которые нуждались в составлении контракта, но были пропущены составителями.

Таблица 1. Результаты работы модуля обработки синхронизационных контрактов
Table 1. The contract processing module work results

Приложение	Кол-во контрактов	Кол-во ошибок	Пропущенные методы
A	80	0	75
B	135	3	95
C	120	1	80

Наборы контрактов в каждом случае описывали поведение порядка 500-1000 методов. Время работы модуля проверки занимает не более 10-20 секунд. Основная часть пропущенных методов приходилась на классы-структуры данных. Например, программист указывал метод get стандартной хеш-таблицы (HashMap) или списка (ArrayList) как немодифицирующий, но не указывал остальные немодифицирующие методы этого класса. Ещё в нескольких случаях

метод вспомогательного класса (utility class) указывался как потокобезопасный, в то время как таковым можно пометить весь класс.

Таким образом, внедрение модуля верификации контрактов оказалось полезным и позволило обнаружить несколько ошибок и ряд неточностей в составлении контрактов при достаточно малом времени работы.

Оценка производительности всего подхода синхронизационных контрактов на фазе динамического анализа является отдельной задачей, требующей, как минимум, разработки методики оценки. Дело в том, что, во-первых, динамический детектор jDRD совершает до ста тысяч операций в секунду[1]. Во-вторых, время, затрачиваемое на динамическую проверку контракта, может превосходить выигрыш от его применения.

Тем не менее, на основе проведенных экспериментов могут быть сделаны некоторые выводы по производительности. Во-первых, два из трёх приложений являются клиентскими, и пользователи показали повышение скорости работы приложений в сравнении с анализом при помощи старой версии jDRD. Во-вторых, общее число обрабатываемых операций синхронизации в секунду сократилось, поскольку количество добавившихся (обработка контрактов) на несколько порядков меньше количества устранённых (обработка операций синхронизации Java внутри контрактных методов). Как следствие, сократился и расход памяти на содержание векторных часов.

Полученные предварительные данные свидетельствуют о неухудшении производительности на анализируемых приложениях и о практической применимости предложенного подхода. Но требуются более детальные экспериментальные исследования.

6. Заключение

Динамический анализ является одним из основных подходов к автоматическому обнаружению гонок, но его практическая применимость ограничена вопросами производительности. Подход к снижению накладных расходов, основанный на описании синхронизационных контрактов сторонних компонент и замене их анализа применением этих контрактов, показал ранее высокую точность и практическую применимость. Однако как разбор контрактов, так и их применение на фазе динамического анализа связано с множеством сложностей. Возникают задачи как валидации и нормализации контрактов при загрузке, так и оптимального их хранения и использования во время анализа целевого приложения.

[1]Отметим, что задача тестирования производительности на микро-уровне (так называемый микробенчмаркинг) невероятно сложна. Так, команда разработчиков Core Java работала над утилитой, позволяющей надёжно измерять производительность операций на уровне микро- и нано-секунд, около 5 лет – см. саму утилиту (http://openjdk.java.net/projects/code-tools/jmh/) и выступление Алексея Шипилёва (https://www.youtube.com/watch?v=8pMfUopQ9Es).

Для решения этих задач предложен метод построения графа контрактов и проверки отсутствия циклов в нём. Далее в исходный код целевого приложения с помощью техники инструментирования байт-кода встраиваются соответствующие инструкции, проверяющие наличие контракта для метода. Хранение часов осуществляется в высокоскоростной хеш-таблице по генерируемым динамически синтетическим ключам. В рамках применения этого подхода для детектора jDRD разработана архитектура модуля контрактов и его техническая реализация.

Доработанный таким образом jDRD был применён на трёх индустриальных приложениях. Измерение основных метрик показало как практическую пользу от построения графа контрактов (был выявлен ряд ошибок и несоответствий в разработанных контрактах), так и сокращение количества обрабатываемых операций синхронизации за единицу времени.

Дальнейшие работы должны включать в себя постановку полноценного эксперимента по измерению времени работы детектора – от разработки методики эксперимента до его проведения. В качестве направления дальнейшего развития средств спецификации контрактов можно указать визуальное моделирование. Интерес представляет описание иерархии контрактов и соответствующих им Java-классов и Java-методов с помощью диаграмм классов UML, расширенных и настроенных подходящим образом [36-38]. Также возможны визуальные спецификации поведения контактов с помощью динамических моделей [39]. Интересно исследовать задачу автоматизированного извлечения описания контрактов из Java-кода и Java-документации и связь повторного использования контрактов с повторным использованием документации [40-41].

Список литературы

[1] Java Language Specification, Third Edition. Threads and Locks. Happens-before Order. http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5

[2] Netzer R., Miller B. What Are Race Conditions? Some Issues and Formalizations. ACM Letters on Programming Languages and Systems, 1(1), 1992, pp. 74–88.

[3] Blackout Final Report, August 14, 2003, http://www.ferc.gov/industries/electric/indus-act/reliability/blackout/ch5.pdf

[4] Leveson N., Turner C. S. An Investigation of the Therac-25 Accidents. In IEEE Computer, vol. 26, N 7, 1993, pp. 18–41.

[5] Engler D., Ashcraft K. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, 2003, pp. 237–252.

[6] Voung J., Jhala R., Lerner S. RELAY: Static Race Detection on Millions of Lines of Code. In ESEC/FSE, 2007, pp. 205–214.

[7] Herlihy M., Shavit N. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008, 528 p.

[8] Kahlon V., Sinha N., Kruus E., Zhang Y.: Static data race detection for concurrent programs with asynchronous calls. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Foundations of Software Engineering, 2009, pp. 13–22.

[9] 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, pp. 308–319.

[10] Radoi C., Dig D. Practical static race detection for java parallel loops. In Proc. of the 13th International Symposium on Software Testing and Analysis, ISSTA ’13, 2013. P.178–190.

[11] Xie X., Xue J., Zhang J. Acculock: Accurate and Efficient Detection of Data Races. Softw. Practice Experience, vol. 43, no. 5, May 2013, pp. 543–576.

[12] Burckhardt S., Musuvathi M. Effective program verification for relaxed memory models. In Proceedings of the 20th international conference on Computer Aided Verification, Berlin, Heidelberg, 2008. pp. 107–120.

[13] Huynh T., Roychoudhury A. Memory model sensitive bytecode verification. Form. Methods Syst. Des., 31(3), 2007, pp. 281–305.

[14] 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, pp. 211–230.

[15] Flanagan C., Freund S. Type inference against races. Sci. Comput. Program., Vol 64, January 2007, pp. 140–165.

[16] Rose J., Swamy N., Hicks M. Dynamic inference of polymorphic lock types. Science of Computer Programming, 58(3), 2005, pp. 366–383.

[17] Biswas S., Zhang M., Bond M., Lucia B. Valor: Efficient, Software-Only Region Conflict Exceptions. In OOPSLA, 2015, pp. 241–259.

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

[19] Kini D., Mathur U., Viswanathan M. Dynamic race prediction in linear time. SIGPLAN Not. 52(6), 2017, pp. 157–170.

[20] Qi Y., Das R., Luo Z., Trotter M. MulticoreSDK: a practical and efficient data race detector for real-world applications. Proceedings Software Testing, Verification and Validation (ICST), IEEE, 21-25 March 2011, pp. 309–318.

[21] Serebryany S., Iskhodzhanov T. ThreadSanitizer: Data race detection in practice. In Proceedings of the Workshop on Binary Instrumentation and Applications, 2009, pp. 62–71.

[22] Serebryany K., Potapenko A., Iskhodzhanov T., Vyukov D. Dynamic race detection with LLVM compiler - compile-time instrumentation for ThreadSanitizer. In RV, 2011, Lecture Notes in Computer Science, vol 7186, pp. 110–114.

[23] Yu M., Bae D., SimpleLock+: Fast and Accurate Hybrid Data Race Detection. Comput. J., vol. 59, no. 6, 2016, pp. 793–809.

[24] Zhang T., Jung C., Lee D. ProRace: Practical Data Race Detection for Production Use. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017, pp. 149–162.

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

[26] 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, pp. 134–143.
[27] Трифанов В.Ю. Обнаружение состояний гонки в Java-программах на основе синхронизационных контрактов. Компьютерные инструменты в образовании. №4, 2012, стр. 16-29.
[28] Трифанов В.Ю, Цителов Д.И. Динамический поиск гонок в Java-программах на основе синхронизационных контрактов. Материалы конференции "Инструменты и методы анализа программ (TMPA-2013)", Кострома, 2013, стр. 273–285.
[29] Трифанов В.Ю., Цителов Д.И. Язык описания синхронизационных контрактов для задачи поиска гонок в многопоточных приложениях. Программная инженерия. Т.8, N 6, 2017, стр. 250–257.
[30] Elmas T., Qadeer S., Tasiran S. Goldilocks: A Race and Transaction-Aware Java Runtime. Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07), 2007, pp. 245–255.
[31] Lamport L. Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM, Vol. 21, Issue 7, 1978, pp. 558–565.
[32] Intel Thread Checker, http://software.intel.com/en-us/intel-thread-checker/
[33] Трифанов В.Ю. Динамическое обнаружение состояний гонки в многопоточных Java-программах. Дисс. на соискание степени канд. техн. наук. СПбГУ, 2013.
[34] Liskov B., Wing J. A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16 (6). November 1994, pp.1811–1841.
[35] Click C. A lock-free wait-free hash table. https://web.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf
[36] Гаврилова Т.А., Лещева И.А., Кудрявцев Д.В. Использование моделей инженерии знаний для подготовки специалистов в области информационных технологий. Системное программирование. 2012. Т. 7. № 1, pp. 90–105.
[37] Кознов Д.В. Основы визуального моделирования. Интернет-Университет Информационных Технологий (ИНТУИТ). Москва, 2008.
[38] Ольхович Л.Б., Кознов Д.В. Метод автоматической валидации UML-спецификаций на основе языка OCL. Программирование. 2003. Т. 29. № 6, стр. 44–50.
[39] Иванов А., Кознов Д., Мурашева Т. Поведенческая модель RTST++. Записки семинара Кафедры системного программирования "Case-средства RTST++". 1998. № 1, стр. 37–52.
[40] Луцив Д.В., Кознов Д.В., Басит Х.А., Терехов А.Н. Задачи поиска нечётких повторов при организации повторного использования документации. Программирование. 2016. № 4, стр. 39–49.
[41] Кознов Д.В., Романовский К.Ю. Автоматизированный рефакторинг документации семейств программных продуктов. Системное программирование. 2009. Т. 4, стр. 128–150.

Applying synchronization contracts approach for dynamic detection of data races in industrial applications

V.Yu. Trifanov <vitaly.trifanov@gmail.com>
St. Petersburg State University,
Universitetski pr., 28, 198504 St. Petersburg, Russia

Abstract. Data race occurs in multithreaded program when several threads simultaneously access same shared data and at least of them writes. Two main approaches to automatic race detection – static and dynamic – have their pros and cons. Dynamic analysis can provide best precision on certain program execution but introduce enormous runtime overheads. Earlier we introduced high-performance approach that improves performance of dynamic race detection. The key idea is to define and exclude external trusted parts of code (e.g. libraries) from analysis and replace them with specifications of their behavior in multithreaded environment. Possible behavior was classified and corresponding language for describing contracts developed. Evaluation on lightweight applications confirmed performance boost but further industrial usage of detector revealed some problems. This article covers that problems, introduces method and architecture of contract processing module and some technical features that help to apply proposed approach on high load production systems.

Ключевые слова: multithreading; data race; dynamic analysis; automatic error detection.

DOI: 10.15514/ISPRAS-2018-30(3)-4

For citation: Trifanov V.Yu. Applying synchronization contracts approach for dynamic detection of data races in industrial applications. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018, pp. 47-62 (in Russian). DOI: 10.15514/ISPRAS-2018-30(3)-4

References

[1] Java Language Specification, Third Edition. Threads and Locks. Happens-before Order. http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5
[2] Netzer R., Miller B. What Are Race Conditions? Some Issues and Formalizations. ACM Letters on Programming Languages and Systems, 1(1), 1992, pp. 74–88.
[3] Blackout Final Report, August 14, 2003, http://www.ferc.gov/industries/electric/indus-act/reliability/blackout/ch5.pdf
[4] Leveson N., Turner C. S. An Investigation of the Therac-25 Accidents. In IEEE Computer, vol. 26, N 7, 1993, pp. 18–41.
[5] Engler D., Ashcraft K. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, 2003, pp. 237–252.
[6] Voung J., Jhala R., Lerner S. RELAY: Static Race Detection on Millions of Lines of Code. In ESEC/FSE, 2007, pp. 205–214.
[7] Herlihy M., Shavit N. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008, 528 p.
[8] Kahlon V., Sinha N., Kruus E., Zhang Y.: Static data race detection for concurrent programs with asynchronous calls. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Foundations of Software Engineering, 2009, pp. 13–22.

[9] 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, pp. 308–319.
[10] Radoi C., Dig D. Practical static race detection for java parallel loops. In Proc. of the 13th International Symposium on Software Testing and Analysis, ISSTA ’13, 2013. P.178–190.
[11] Xie X., Xue J., Zhang J. Acculock: Accurate and Efficient Detection of Data Races. Softw. Practice Experience, vol. 43, no. 5, May 2013, pp. 543–576.
[12] Burckhardt S., Musuvathi M. Effective program verification for relaxed memory models. In Proceedings of the 20th international conference on Computer Aided Verification, Berlin, Heidelberg, 2008. pp. 107–120.
[13] Huynh T., Roychoudhury A. Memory model sensitive bytecode verification. Form. Methods Syst. Des., 31(3), 2007, pp. 281–305.
[14] 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, pp. 211–230.
[15] Flanagan C., Freund S. Type inference against races. Sci. Comput. Program., Vol 64, January 2007, pp. 140–165.
[16] Rose J., Swamy N., Hicks M. Dynamic inference of polymorphic lock types. Science of Computer Programming, 58(3), 2005, pp. 366–383.
[17] Biswas S., Zhang M., Bond M., Lucia B. Valor: Efficient, Software-Only Region Conflict Exceptions. In OOPSLA, 2015, pp. 241–259.
[18] Flanagan C., Freund S. FastTrack: Efficient and Precise Dynamic Race Detection. In ACM Conference on Programming Language Design and Implementation, 2009, pp. 121–133.
[19] Kini D., Mathur U., Viswanathan M. Dynamic race prediction in linear time. SIGPLAN Not. 52(6), 2017, pp. 157–170.
[20] Qi Y., Das R., Luo Z., Trotter M. MulticoreSDK: a practical and efficient data race detector for real-world applications. Proceedings Software Testing, Verification and Validation (ICST), IEEE, 21-25 March 2011, pp. 309–318.
[21] Serebryany S., Iskhodzhanov T. ThreadSanitizer: Data race detection in practice. In Proceedings of the Workshop on Binary Instrumentation and Applications, 2009, pp. 62–71.
[22] Serebryany K., Potapenko A., Iskhodzhanov T., Vyukov D. Dynamic race detection with LLVM compiler - compile-time instrumentation for ThreadSanitizer. In RV, 2011, Lecture Notes in Computer Science, vol 7186, pp. 110–114.
[23] Yu M., Bae D., SimpleLock+: Fast and Accurate Hybrid Data Race Detection. Comput. J., vol. 59, no. 6, 2016, pp. 793–809.
[24] Zhang T., Jung C., Lee D. ProRace: Practical Data Race Detection for Production Use. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017, pp. 149–162.
[25] Bond M., Coons K., McKinley K. Pacer: Proportional Detection of Data Races. Proceedings of 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010), Toronto, June 2010, pp. 255–268.
[26] 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, pp. 134–143.

[27] Trifanov V.Yu. Detecting data races in Java programs with synchronization contracts. Komp'juternye instrumenty v obrazovanii [Computer Tools in Education]. №4, 2012, pp. 16-29. (in Russian)
[28] Trifanov V.Yu., Tsitelov D.I. Dynamic detection of data races in Java programs with synchronization contracts. Materialy konferencii "Instrumenty i metody analiza programm (TMPA-2013)" [Proc of Tools and Methods of Program Analysis conference TMPA-2013], Kostroma, 2013, pp. 273–285. (in Russian)
[29] Trifanov V.Yu., Tsitelov D.I. Language for synchronization contracts creation to detect races in multithreaded applications. Programmnaja inzhenerija [Software Engineering], vol. 8, N 6, 2017, pp. 250–257. (in Russian)
[30] Elmas T., Qadeer S., Tasiran S. Goldilocks: A Race and Transaction-Aware Java Runtime. Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07), 2007, pp. 245–255.
[31] Lamport L. Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM, Vol. 21, Issue 7, 1978, pp. 558–565.
[32] Intel Thread Checker, http://software.intel.com/en-us/intel-thread-checker/
[33] Trifanov V.Yu. Dynamic data race detection in multithreaded Java-programs. PhD thesis, SPbSU, 2013. (in Russian)
[34] Liskov B., Wing J. A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16 (6). November 1994, pp.1811–1841.
[35] Click C. A lock-free wait-free hash table. https://web.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf
[36] Gavrilova T.A., Leshheva I.A., Kudrjavcev D.V. Using models of knowledge engineering for growing specialists in information technologies area. Sistemnoe programmirovanie [System programming], vol. 7, № 1, 2012, pp.. 90–105. (in Russian)
[37] Koznov D.V. Basis of visual modeling. Internet-Universitet Informacionnyh Tehnologij (INTUIT) [Internet-University of Information Technologies], Moscow, 2008 (in Russian)
[38] Ol'hovich L.B., Koznov D.V. OCL-Based Automated Validation Method for UML Specifications. Programming and Computer Software, vol. 29, № 6, 2003, pp. 44–50. DOI: 10.1023/B:PACS.0000004132.42846.11
[39] Ivanov A., Koznov D., Murasheva T. Behavioral model RTST++, Zapiski seminara Kafedry sistemnogo programmirovanija "Case-sredstva RTST++" [Notes of seminar “Case-tools RTST++” of system engeneering department], 1998, № 1, pp. 37–52. (in Russian)
[40] Luciv D.V., Koznov D.V., Basit H.A., Terehov A.N. On fuzzy repetitions detection in documentation reuse. Programming and Computer Software, vol. 42, № 4, 2016, pp. 39–49. DOI: 10.1134/S0361768816040046
[41] Koznov D.V., Romanovskij K.Ju. Automated documentation refactoring for lines of program products. Sistemnoe programmirovanie [System programming], vol. 4, 2009, pp. 128–150. (in Russian)