На правах рукописи

Исходжанов Тимур Равилевич

АВТОМАТИЧЕСКИЙ ПОИСК ОШИБОК В КОМПЬЮТЕРНЫХ ПРОГРАММАХ С ПРИМЕНЕНИЕМ ДИНАМИЧЕСКОГО АНАЛИЗА

Специальность 05.13.11
Математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

АВТОРЕФЕРАТ

диссертации на соискание ученой степени
кандидата технических наук

Москва — 2013

Работа выполнена на кафедре информатики федерального государственного автономного образовательного учреждения высшего профессионального образования «Московский физико-технический институт (государственный университет)»

Научный руководитель: член-корреспондент РАН,
доктор физико-математических наук, профессор
Петров Игорь Борисович

Официальные оппоненты: доктор физико-математических наук, профессор
Петренко Александр Константинович.
Федеральное государственное бюджетное учреждение науки Институт системного программирования РАН, заведующий отделом Технологий программирования.

кандидат физико-математических наук
Потапов Антон Павлович.
ООО «Параллелз Рисерч»,
старший инженер-программист.

Ведущая организация: **Федеральное государственное бюджетное учреждение науки Научно-исследовательский институт системных исследований Российской академии наук (НИИСИ РАН)**

Защита состоится _____ 2013 г. в _____ часов на заседании диссертационного совета Д 002.017.02 при Федеральном государственном бюджетном учреждении науки «Вычислительный центр им. А.А. Дородницына Российской академии наук», расположенном по адресу: 119333, г.Москва, улица Вавилова, 40. С диссертацией можно ознакомиться в библиотеке ВЦ РАН.

Автореферат разослан _____ 2013 г.

Ученый секретарь диссертационного совета
Д 002.017.02, д.ф.-м.н., профессор  Рязанов В.В.

Общая характеристика работы

Актуальность работы

Компьютерные программы часто содержат ошибки. Ошибки в программном обеспечении могут приводить к разнообразным последствиям, в том числе к таким серьезным, как большие экономические потери и гибель людей; к неправильному или непредсказуемому поведению программ, замедлению их работы, аварийным завершениям исполнения, порче данных и т.п. Ошибки бывают детерминированные (воспроизводятся при одних и тех же начальных данных) и недетерминированные (приводят к различным последствиям от запуска к запуску).

Особый интерес представляют ошибки типа *«неопределенное поведение»* (англ. *undefined behavior*), которые происходят при нарушении программистом стандарта языка либо спецификаций используемых программных функций или библиотек. Примером такой ошибки является доступ к памяти за границами массива в программах на языках C/C++. Часто неопределенное поведение указывается в спецификации языка или интерфейса в целях ее упрощения и оптимизации, так как допускает большое количество различных корректных реализаций, из которых можно для каждого конкретного случая выбрать наиболее подходящую (например, наиболее эффективную). В случае нарушения спецификации, приводящего к возникновению неопределенного поведения, компилятор или библиотека вправе выполнять некорректные и неожиданные действия.

Особенностью таких ошибок является то, что *обычно* они не имеют наблюдаемых последствий, но *иногда* могут приводить к серьезным сбоям. При этом условия возникновения этих ошибок могут быть трудновоспроизводимыми, например, зависеть от версии компилятора, библиотеки или ОС, от частоты процессора, количества ядер или даже его температуры. Более того, многие такие ошибки приводят не ко мгновенным последствиям, а к порче данных, эффект от которой будет наблюдаться лишь через некоторое время, затрудняя понимание и обнаружение ошибки.

Многие такие ошибки не удается легко обнаружить при использовании обычных средств тестирования, даже при многократном запуске тестов. Про-

блема становится всё более острой, учитывая постоянный рост сложности и объемов исходных кодов современных программ, а также количества задач, которые решаются с помощью компьютеров. **Для эффективного нахождения ошибок в крупных программных проектах требуется использовать автоматические инструменты**. В данной работе такие инструменты будут называться *детекторами* ошибок.

Детекторы ошибок в основном разделяются на *статические* и *динамические*. Статический анализ ошибок выполняется над исходным кодом программы без необходимости запускать ее. В частности, к статическому анализу относится формальное доказательство корректности кода программы. Иногда статический анализ ошибок встроен непосредственно в компилятор и приводит к предупреждениям или даже сообщениям об ошибках, если корректность кода сомнительна. Динамический анализ заключается в запуске программы или отдельных ее частей в рамках набора тестов с наблюдением и анализом ее поведения. По сути, при использовании анализаторов происходит ужесточение правил языка программирования, когда часть «неопределенных поведений» имеет определенный результат, такой как ошибка компиляции, печать сообщения об ошибке или аварийное завершение работы программы. Следует отметить, что ни одно из этих поведений не противоречит стандарту языка.

К очевидным недостаткам динамического анализа относится то, что ошибки возможно находить только в том коде, который исполняется на наборе тестов («покрыт» тестами). Впрочем, если код крупного программного проекта недостаточное покрыт тестами, то это само по себе является проблемой. К недостаткам статического анализа относится его значительная вычислительная сложность (нередко — более чем линейно зависящая от размера кода) и обычно низкая точность. Часто поиск нетривиальных ошибок статическим анализом с достаточной точностью возможен только для отдельных изолированных модулей программы и требует особой организации ее исходного кода.

В рамках данной работы **был выбран динамический анализ**, так как стояла практическая задача нахождения ошибок в существующей большой базе кода (более 100 млн строк кода на языке C++), существенное редактирование и реорганизация которой за разумное время представлялись затруднительными.

Существует множество различных алгоритмов, детекторов и подходов к динамическому анализу компьютерных программ, однако в ходе их иссле-

дования было обнаружено, что они зачастую обладают недостаточной эффективностью, функциональностью и точностью.

Цель диссертационной работы — разработка новых методов динамического тестирования программ, позволяющих достичь большей эффективности и точности при поиске ошибок в крупных программных проектах, чем у известных.

Для достижения цели работы были поставлены следующие задачи:

1. Изучение и развитие существующих средств поиска ошибок работы с памятью и «состояний гонок» (data race).

2. Разработка нового алгоритма поиска ошибок типа «состояние гонки» и динамического детектора таких ошибок, применимого для тестирования кода с нестандартными средствами синхронизации. Сравнение эффективности и точности полученного детектора с аналогами.

3. Исследование методов компиляторного, статического и динамического инструментирования кода, а также развитие средств инструментирования для задач автоматического поиска ошибок.
   (Инструментирование — модификация или внедрение кода в программу, например с целью отслеживания происходящих в ней событий.)

4. Создание автоматической системы тестирования крупного программного проекта с применением модернизированных и новых разработанных детекторов ошибок.

Научная новизна. Благодаря применению динамических аннотаций удалось упростить задачу нахождения ошибок типа «состояние гонки» (которая является NP-трудной), что позволило применить уточненную модель одновременности событий в многопоточных программах. Используя эту модель, был разработан новый алгоритм поиска таких ошибок, обладающий большей точностью, чем аналоги.

Разработан новый подход к инструментированию программ с целью поиска ошибок, позволяющий достичь большей функциональности и эффективности, чем ранее известные.

Практическая значимость заключается в увеличении эффективности

и точности инструментальных средств поиска ошибок, доступных разработчикам программных продуктов.

Разработан и реализован ThreadSanitizer, новый детектор ошибок типа «состояние гонки». Благодаря использованию ThreadSanitizer для тестирования браузера Chromium, а также серверного программного обеспечения компании Google было обнаружено более тысячи ранее неизвестных ошибок, в том числе десятки критических.

Предложен новый подход к инструментированию, позволяющий разрабатывать более эффективные и точные детекторы.

Рассматриваются вопросы практического применения детекторов для тестирования крупных программных проектов. Были доработаны известные детекторы ошибок работы с памятью (в частности, Valgrind/Memcheck).

Основными **методами исследования**, используемыми в работе, являются методы верификации программного обеспечения, системного программирования, теории алгоритмов и сложности, аппарат теории множеств и теории порядка. Для оценки эффективности предлагаемых решений применяются тесты производительности.

На защиту выносятся следующие положения:

- Разработан новый алгоритм поиска ошибок типа «состояние гонки» (data race) с применением уточненной модели одновременности событий в многопоточных программах. Алгоритм был реализован в новом детекторе состояний гонок для программ на языках C/C++, позволяющем достичь большей точности нахождения ошибок, чем у аналогов.

- Разработан и реализован новый подход к инструментированию кода программ, основанный на комбинации компиляторной и динамической инструментации.

- Создана система регулярного автоматического тестирования программного обеспечения, использующая различные динамические детекторы ошибок, доработанные на основании их анализа. Система внедрена для тестирования веб-браузера Chromium.

Программные реализации, описываемые в данной работе, доступны в виде открытых исходных кодов.

Апробация работы. На разных этапах работы основные результаты по теме диссертации были представлены на следующих конференциях и семинарах:

1. Workshop on Binary Instrumentation and Applications, WBIA'09 (Нью-Йорк, 2009).
2. 52-ая научная конференция МФТИ (Долгопрудный, 2009).
3. Runtime Verification, RV 2011 (Сан-Франциско, 2011).
4. Семинар Института системного программирования РАН (Москва, 2013).
5. Научные семинары кафедры информатики МФТИ (Москва, Долгопрудный, 2010–2013).

По теме диссертации автором опубликовано 5 работ (из них 3 в изданиях из перечня ВАК).

Разработанные детекторы и подходы были успешно использованы для тестирования браузера с открытым исходным кодом Chromium, а также серверного программного обеспечения Google.

Структура и объем диссертации. Диссертационная работа состоит из введения, четырех глав, заключения, списка использованных источников и одного приложения. Список использованных источников включает 67 наименований. Общий объем работы составляет 133 страницы. Объем приложений составляет 16 страниц.

Содержание работы

Во **введении** обоснована актуальность диссертационной работы, сформулирована цель и аргументирована научная новизна исследований, показана практическая и теоретическая значимость полученных результатов, представлены выносимые на защиту результаты и положения.

В **первой главе** дан обзор известных методов и подходов к поиску ошибок.

Описываются основные сценарии тестирования, с которыми применяются детекторы ошибок: ручное, автоматизированное, а также рандомизированное.

Для дальнейшего анализа точности детекторов определяются понятия *ошибки первого рода* (англ. *false positive, ложное срабатывание*) и *ошибки второго рода* (англ. *false negative, пропуск ошибки*), а также обсуждаются проблемы, к которым может приводить неточность детектора, и способы борьбы с ними.

Инструментированием или *инструментацией* называется модификация или внедрение кода в программу, например с целью отслеживания происходящих в ней событий для дальнейшей обработки алгоритмом поиска ошибок. Рассматриваются основные подходы к инструментированию программ: динамическое (во время работы программы), статическое (перед запуском программы) и компиляторное (во время компиляции).

Динамическое инструментирование удобно тем, что его можно применять даже к программам, исходный код и отладочная информация которых недоступна; а также к самомодифицирующимся программам и программам, генерирующим исполняемый код на лету (JIT). К недостаткам этого подхода можно отнести ограниченные возможности по внедрению дополнительного кода, как например: невозможность изменения расположения локальных переменных функций на стеке, чувствительность алгоритмов анализа к оптимизациям тестируемого кода, а также низкую производительность, особенно в начале работы программы.

Для корректной работы систем, использующих динамическое инструментирование, необходимо, чтобы все косвенные вызовы в программе выполнялись через специальный диспетчер вызовов системы инструментирования. В общем случае для этого требуется транслировать весь исполняемый код, что влечет за собой дополнительные затраты порядка 1000 инструкций системы инструментирования на одну инструкцию тестируемой программы. Эти затраты становятся существенными при тестировании крупных программных проектов.

Статическое инструментирование позволяет минимизировать затраты на инструментацию во время исполнения, выполняя ее до запуска. Однако его возможности еще более ограничены: в частности, оно не может применяться к самомодифицирующемуся и динамически генерируемому коду, а также в общем случае требует наличия отладочной информации программы.

Компиляторное инструментирование выполняется компилятором. Благодаря работе на уровне исходного кода или его промежуточного представления такой подход позволяет изменять расположение глобальных переменных,

локальных переменных в стековой памяти, а также достигать бóльшей производительности. К недостаткам можно отнести необходимость наличия исходного кода программы, а также невозможность инструментирования самомодифицирующегося и динамически генерируемого кода.

Помимо инструментирования, для поиска ошибок чаще всего нужно также применять специальную *библиотеку времени исполнения* (англ. *runtime library*), добавляющую в программу новые функции, а также заменяющую реализации существующих функций. Это позволяет внедренному инструментацией коду, а также оригинальному коду программы нужным образом взаимодействовать с алгоритмом детектора.

Многим алгоритмам поиска ошибок требуется хранить некоторое вспомогательное «состояние» для каждой ячейки или блока памяти. Совокупность состояний переменных называют *теневой памятью* (англ. *shadow memory*) или просто *тенью*, она выделяется и обрабатывается библиотекой времени исполнения и самим анализатором. Описываются основные подходы к организации теневой памяти: в заголовках блоков динамической памяти, в хэш-таблице; линейное и двухуровневое линейное отображения адресного пространства.

Затем рассматривается несколько распространенных типов ошибок в программах на языках C/C++ и устройство детекторов таких ошибок. Приводится пример программы с ошибкой типа «переполнение знаковой целочисленной переменной», которая корректно работает при компиляции в отладочном режиме, но приводит к неверному результату и завершается аварийно при использовании оптимизаций.

Описывается детектор *ошибок работы с памятью* Memcheck. Приводятся примеры ошибок следующих типов: утечка динамической памяти, повторное освобождение блока динамической памяти, выход за границы динамической памяти, доступ к освобожденной памяти, использование неинициализированных данных — и описываются их возможные последствия. Рассматриваются алгоритмы, которыми детектор Memcheck находит такие ошибки. Из-за сложности применяемых алгоритмов и использования динамического инструментирования Memcheck существенно замедляет работу тестируемых программ — в среднем, в 20–30 раз для однопоточных программ и еще больше для многопоточных.

Следует отметить, что в детекторе Memcheck разделяются понятия утеч-

ки памяти и возможной утечки памяти. *Множеством начальной достижимости* (в заданный момент времени, например в конце работы программы) назовем совокупность регистров общего назначения всех работающих потоков, их стеков, а также глобальной памяти. *Достижимый блок памяти* — такой, на начало которого есть указатель во множестве начальной достижимости или в другом достижимом блоке. *Возможно достижимым блоком* назовем такой, на один из байтов которого есть указатель во множестве начальной достижимости или в другом возможно достижимом блоке. Утечками памяти обычно считаются те блоки, которые не являются возможно достижимыми, а возможными утечками считают те блоки, которые не являются достижимыми. В зависимости от специфики конкретного программного проекта бывает и так, что требуется освобождать все выделенные блоки динамической памяти, тогда даже блоки памяти, достижимые в конце работы программы, считаются утечками.

Описываются применяемые в детекторе Memcheck *правила подавления* (англ. *suppressions*), используя которые можно избавляться от ложных срабатываний детектора, а также автоматически игнорировать повторные сообщения об известных ошибках.

Детектор AddressSanitizer не находит ошибок типа «использование неинициализированных данных», зато благодаря применению компиляторного инструментирования и более простому алгоритму поиска ошибок работы с памятью он работает значительно эффективнее Memcheck. В среднем тестируемые программы замедляются примерно вдвое. Осуществление инструментации в компиляторе позволяет также находить ошибки работы со стековой памятью и глобальными переменными. При этом детектору AddressSanitizer удается находить ошибки без ложных срабатываний. Однако из-за ограничений компиляторного инструментирования этот детектор не может находить некорректные доступы к памяти, происходящие в системных и сторонних библиотеках.

Рассматриваются основные подходы к поиску ошибок типа *«состояние гонки»*, возникающие в многопоточных программах.

Определение. ***Состоянием гонки*** *(англ. data race) называется ситуация, когда два или более потока исполнения программы могут одновременно осуществить доступ к одной области памяти (например, переменной) и как*

минимум один из этих доступов является записью.

Состояния гонок — одни из самых опасных и труднонаходимых при тестировании и отладке ошибок. При этом переход развития аппаратного обеспечения от роста частот к росту количества ядер и процессоров делает задачу эффективного поиска таких ошибок всё более актуальной. Задача поиска состояний гонок является NP-трудной, однако существуют приближенные алгоритмы ее решения. Динамические детекторы без особого труда могут проверить все условия в определении состояния гонки, за исключением одновременности событий. Таким образом, именно моделью одновременности событий в основном отличаются алгоритмы поиска таких ошибок.

Алгоритм Lockset ставит каждой разделяемой переменной V в соответствие множество блокировок $LS(V)$, которые были захвачены при всех наблюдавшихся доступах к V. При каждом новом доступе к V алгоритм записывает в $LS(V)$ пересечение старого значения $LS(V)$ и множества захваченных текущим потоком блокировок. Если это множество становится пустым, то это значит, что нет ни одной блокировки, которая бы синхронизировала все доступы к V. С точки зрения алгоритма Lockset это значит, что возможны одновременные доступы к переменной V из нескольких потоков, то есть состояние гонки.

Алгоритм Lockset не пропускает состояний гонок, которые могут возникнуть в исполняемом при тестировании коде (отсутствие ошибок второго рода), а также очень эффективно реализуется в детекторах; однако существует множество примеров ложных срабатываний, в особенности при использовании таких средств синхронизации, как семафоры и очереди сообщений. Другими словами, модель одновременности, применяемая в алгоритме Lockset, неточна: он хорошо подходит только для тестирования подмножества многопоточных программ, в которых вся синхронизация осуществляется посредством блокировок.

Другой распространенный подход к поиску состояний гонок — установление между событиями в программе отношения частичного порядка «$\prec$» («предшествует»). Пользуясь гипотезой о том, что если при одном запуске программы событие A предшествовало B, то они не могут произойти одновременно, можно считать, что события A и B могут происходить одновременно тогда и только тогда, когда ни одно из них не предшествует другому. Использующий эту гипотезу алгоритм (далее — алгоритм Happens-before) проверяет доступы к каждой области памяти и, если ему не удается

установить отношение предшествования между ними, сообщает о наличии состояния гонки. Используя такой алгоритм, удается находить состояния гонок без ложных срабатываний при условии, что детектор наблюдает все события синхронизации в программе. Однако применение такой гипотезы приводит к большому количеству пропускаемых ошибок в программах, использующих для синхронизации блокировки, то есть такая модель одновременности событий также неточна. Кроме того, вычисление отношения предшествования требует бóльших временных затрат, чем у алгоритма Lockset.

Существуют и *гибридные алгоритмы поиска состояний гонок*, которые пытаются совместить Lockset и Happens-before. Обычно они отличаются своей сложностью, при этом, вместо теоретического обоснования, они строятся эмпирически. Так как задача NP-трудна, эти алгоритмы не могут быть точными, а их сложность затрудняет понимание отчетов об ошибках. При изучении статей по гибридным алгоритмам не было обнаружено решений, позволяющих эффективно бороться с ложными срабатываниями, что требуется для тестирования крупных программных проектов.

Во **второй главе** описывается алгоритм поиска состояний гонок, разработанный для детектора ThreadSanitizer.

Алгоритм может работать в двух режимах: гибридном и Happens-before. Вместо решения NP-трудной задачи в общем случае при использовании гибридного алгоритма в код тестируемой программы может потребоваться добавить вызовы специальных функций — *динамических аннотаций*. Аннотируя нестандартные средства синхронизации, применяемые в тестируемой программе, удается более точно устанавливать отношение предшествования между событиями, чем это делает алгоритм Happens-before. Аннотации можно также применять для создания вспомогательных событий уведомления и ожидания в коде, что позволяет детектору более точно обрабатывать такие паттерны многопоточного программирования, как очереди сообщений и подсчет ссылок.

Используя инструментацию, детектор наблюдает процесс работы исследуемой программы в виде последовательности событий *синхронизации* и *доступов к памяти*. События доступов к памяти бывают двух типов: чтение и запись. События синхронизации бывают двух основных типов: уведомление-ожидание (семафоры, POSIX condvar и подобные примитивы) и работа с блокировками

(захват/освобождение).

Введем несколько вспомогательных определений для описания алгоритма.

Определение. *Событие A* ***предшествует*** *событию B ($A \prec B$), если A наблюдалось до B и выполнено хотя бы одно из следующих утверждений:*

- *A и B были выполнены одним и тем же потоком исполнения;*
- *A является* ***уведомлением*** *(англ. signal) некоторого примитива P (семафора, элемента очереди сообщений и т.п.), при этом B является* ***ожиданием*** *(англ. wait) уведомления на том же примитиве;*
- *Существуют такие события A' и B', что выполняется $A \preceq A' \prec B' \preceq B$ (транзитивность), где $X \preceq Y \equiv (X \prec Y) \vee (X = Y)$.*

Следует отметить, что в отличие от алгоритма Happens-before, в гибридном алгоритме ThreadSanitizer не устанавливается отношения предшествования между событиями над блокировками; вместо этого применяются множества блокировок подобно алгоритму Lockset.

Определение. *Непрерывную подпоследовательность событий, произошедших в одном потоке, среди которых нет событий синхронизации, будем называть* ***сегментом****.*

Так как все события в сегменте произошли в одном потоке, они не могут участвовать в состоянии гонки друг с другом.

Естественным образом определяется отношение предшествования между сегментами: сегмент $S_a \prec S_b$, если для любых событий $A \in S_a$ и $B \in S_b$ выполняется $A \prec B$.

Пусть сегмент S произошел в потоке T. Из определения сегмента естественным образом следует, что во время выполнения всех его событий потоком T захвачена (locked) одна и та же пара наборов блокировок для записи $LS_{wr}(S)$ (writer-lockset) и для чтения $LS_{rd}(S)$ (reader-lockset).

Определение. *Пусть A — операция доступа к памяти, выполненная внутри сегмента S. Определим* ***эффективный набор взятых для операции*** *A* ***блокировок*** *LS_A как:*

$LS_A = LS_{wr}(S)$, *если* A — *операция записи и*
$LS_A = LS_{wr}(S) \cup LS_{rd}(S)$, *если* A — *операция чтения.*

Определение. *Будем считать, что события доступа к памяти* A *и* B ***могут произойти одновременно****, если они не предшествуют друг другу* ($A \not\prec B$, $B \not\prec A$) *и пересечение их эффективных наборов взятых блокировок пусто* ($LS_A \cap LS_B = \emptyset$).

Напомним, что *состоянием гонки* называется такая ситуация, когда два или более потока исполнения программы могут одновременно осуществить доступ к одной области памяти и как минимум один из этих доступов является записью.

Следует отметить, что такое определение одновременности учитывает и блокировки, и предшествование событий. Как показывается в приложении, такое определение более точно, чем определения, применяемые в алгоритмах Lockset и Happens-before,

Далее также пригодится следующее определение:

Определение. *Будем называть набор сегментов* $SS = \{S_1, ...S_N\}$ ***набором неупорядоченных сегментов*** *(англ. segment set), если для любых* S_i *и* S_j *из этого набора выполняется* $S_i \not\preceq S_j$.

Следует заметить, что по определению в наборе неупорядоченных сегментов не может быть нескольких сегментов из одного и того же потока.

При обработке каждого события синхронизации в текущем потоке начинается новый сегмент и для него вычисляются текущие наборы блокировок LS_{wr} и LS_{rd}. Также производятся вычисления, необходимые для эффективной проверки отношения предшествования между событиями.

Состояние области памяти описывается в ячейке теневой памяти двумя наборами сегментов: SS_{wr} и SS_{rd}, которые соответствуют ранее наблюдавшимся операциям записи и чтения в эту область памяти. SS_{wr} байта памяти — это набор неупорядоченных сегментов, в которых в него происходили запись и чтение, а его SS_{rd} — это набор неупорядоченных сегментов, в которых из него происходило чтение, но не запись. Начальные SS_{wr} и SS_{rd} для каждого байта памяти пусты. Далее каждый доступ к памяти обрабатывается алгоритмом 1.

Если в процессе обработки обнаруживается, что среди двух наборов неупорядоченных сегментов некоторого байта по адресу $Addr$ есть пара сег-

Алгоритм 1 Обработка доступа к памяти по адресу $Addr$ потоком T_{id}.

```
1:  function HANDLE-MEMORY-ACCESS(Addr, IsWrite, T_id)
2:      (SS_wr, SS_rd) ← GET-STATE-FOR-ADDRESS(Addr)
3:      Seg ← GET-CURRENT-SEGMENT(T_id)
4:      if IsWrite then
5:          // Обработка записи изменяет и SS_wr, и SS_rd.
6:          SS_rd ← {s : s ∈ SS_rd ∧ s ⋠ Seg}
7:          SS_wr ← {s : s ∈ SS_wr ∧ s ⋠ Seg} ∪ {Seg}
8:      else if Seg ∉ SS_wr then
9:          // Обработка чтения изменяет только SS_rd.
10:         SS_rd ← {s : s ∈ SS_rd ∧ s ⋠ Seg} ∪ {Seg}
11:     SET-STATE-FOR-ADDRESS(Addr, SS_wr, SS_rd)
12:     if CHECK-IF-RACE(SS_wr, SS_rd) then
13:         // Сообщить о состоянии гонки на памяти по адресу Addr.
14:         REPORT-RACE(IsWrite, T_id, Seg, Addr)
```

ментов, хотя бы один из которых соответствует записи, а пересечение их эффективных наборов взятых блокировок пусто, то это значит, что имеет место состояние гонки (см. алгоритм 2).

Алгоритм 2 Алгоритм проверки наличия состояния гонки

```
1:  function CHECK-IF-RACE(SS_wr, SS_rd)
2:      // Проверка наличия состояния гонки.
3:      N_W ← SEGMENT-SET-SIZE(SS_wr)
4:      // Перебор сегментов, в которых была запись.
5:      for i = 1 to N_W do
6:          W_1 ← SS_wr[i]
7:          LS_1 ← GET-WRITER-LOCK-SET(W_1)
8:          for j = i + 1 to N_W do
9:              // Проверка того, что не было одновременных записей.
10:             W_2 ← SS_wr[j]
11:             LS_2 ← GET-WRITER-LOCK-SET(W_2)
12:             // W_1 ⋠ W_2, W_2 ⋠ W_1 по построению.
13:             if LS_1 ∩ LS_2 = ∅ then
14:                 return true
15:         for R ∈ SS_rd do
16:             // Проверка одновременности пар чтение-запись.
17:             LS_R ← GET-READER-LOCK-SET(R) ∪ GET-WRITER-LOCK-SET(R)
18:             if W_1 ⋠ R and LS_1 ∩ LS_R = ∅ then
19:                 return true
20:     return false
```

При использовании алгоритма, вычисляющего отношение предшествования за $O(T)$, алгоритм 1 имеет сложность в худшем случае $O(S^2(T + L))$ (где S — максимальное количество сегментов в наборе, T — количество уникальных потоков за время жизни программы, а L — максимальное количество одновременно захваченных потоком блокировок). При ограничении максимального

количества сегментов в наборе, а также благодаря программным оптимизациям удается сделать сложность обработки доступов в среднем $O(T)$.

При использовании алгоритма вычисления отношения предшествования, имеющего сложность $O(1)$, удается достичь сложности в худшем случае $O(S^2L)$ и $O(1)$ в среднем. Однако на практике ускорения работы детектора от оптимизации процедуры вычисления отношения предшествования не наблюдалось.

Следует отметить важное свойство алгоритма ThreadSanitizer: он сводится к алгоритму Happens-before, если устанавливать отношение предшествования между событиями над блокировками, как это сделано в классическом алгоритме. При работе в гибридном режиме отношение предшествования устанавливается между событиями из разных потоков реже, чем в алгоритме Happens-before, благодаря чему уменьшается количество пропускаемых ошибок, а также уменьшаются накладные расходы на вычисление отношения предшествования.

Алгоритм ThreadSanitizer был разработан таким образом, чтобы его отчеты о найденных ошибках были понятны обычным разработчикам. Это потребовало как тщательного анализа типичных ошибок и примеров корректного многопоточного кода, так и специальной организации данных алгоритма, чтобы отчеты были достаточно информативными. В частности, в отличие от многих других детекторов состояний гонок, ThreadSanitizer печатает стеки вызовов (англ. *stack traces*), соответствующие всем доступам к памяти, участвующим в гонке, а не только последнему. Также при печати сообщений об ошибках в любом из двух режимов детектора сообщается информация о взятых каждым потоком блокировках, что не позволяет делать классический алгоритм Happens-before.

Далее описывается организация теневой памяти, а также приемы, благодаря которым удалось эффективно реализовать алгоритм в детекторе, как например применение упрощенных векторных часов для вычисления отношения предшествования и кэширование результатов часто повторяющихся бинарных операций.

Рассматриваются основные функциональные отличия гибридного алгоритма ThreadSanitizer от алгоритмов Lockset и Happens-before, а также сравнивается их работа на типичных конфигурациях взаимодействующих потоков (в приложении).

Далее описываются основные *динамические аннотации*, которые позво-

ляют детектору корректно обрабатывать нестандартные средства синхронизации, а также избавиться от ложных срабатываний гибридного алгоритма, при этом находя больше ошибок, чем алгоритм Happens-before. Типичные примеры многопоточного кода, в который необходимо вносить динамические аннотации, приведены в приложении.

Рассматриваются способы обработки стандартных средств синхронизации, таких как POSIX Threads, а также синхронизации с применением низкоуровневых атомарных инструкций. Приводятся рекомендации по применению детектора для поиска ошибок в программах, в том числе порядок выбора режимов работы и внесения в код аннотаций. Рассматривается трудоемкость и целесообразность внесения аннотаций.

Далее приводятся данные экспериментальной оценки производительности двух версий детектора ThreadSanitizer, использующих динамическое и компиляторное инструментирование. Показывается, что при использовании системы динамического инструментирования Valgrind скорость работы сравнима с другими детекторами, использующими эту систему (замедление на тестах Chromium в 20–30 раз), а при использовании компиляторного инструментирования производительность ThreadSanitizer существенно превосходит аналоги (замедление на тестах Chromium в 2–3 раза).

Благодаря использованию детектора ThreadSanitizer для тестирования браузера Chromium и серверного программного обеспечения Google удалось найти более тысячи ошибок типа «состояние гонки», десятки из которых были признаны критически опасными.

В **третьей главе** рассматривается новый подход к инструментированию — комбинация компиляторного инструментирования тестируемой программы с динамическим инструментированием сторонних и системных библиотек, а также сгенерированного и самомодифицирующегося кода. Это позволяет достичь бóльшей производительности, чем при применении только динамического инструментирования, при этом получая бóльшую функциональность, чем у каждого из этих подходов по отдельности.

В работе показывается практическая польза комбинированного инструментирования на примере детекторов DRASan (на базе AddressSanitizer) и MSanDR (на базе MemorySanitizer). Описывается, что в зависимости от спе-

цифики детектора можно применять комбинированное инструментирование как для нахождения ошибок в бóльшем объеме кода, чем при применении только компиляторного инструментирования (уменьшение количества ошибок второго рода), так и для уменьшения количества ошибок первого рода, если алгоритму детектора для корректной работы требуется наблюдение событий во всей программе.

Исследуется производительность систем инструментирования Valgrind, PIN и DynamoRIO в условиях, специфичных для комбинированного инструментирования — когда внедрение дополнительного кода требуется только для отдельных динамических библиотек. Показывается, что при тестировании крупных программ из-за неэффективности этих систем может происходить замедление в 50–700 раз. Это гораздо больше, чем среднее замедление от десятков процентов до 4 раз на тестах SPEC CPU2006, которые обычно используются авторами систем инструментирования для оценки производительности.

Производится анализ замедления DynamoRIO, самой быстрой из протестированных систем инструментирования, и описываются выполненные для ее дальнейшего ускорения оптимизации. В частности, показывается, что для частичного динамического инструментирования (и, как следствие, для комбинированного) оказывается полезным указывать список модулей программы, не требующих внедрения дополнительного кода, что позволяет сократить накладные расходы на транслирование кода в 2,5–3,5 раза. В результате предложенных оптимизаций удается достичь четырехкратного ускорения системы DynamoRIO на тестах производительности.

Описывается новый архитектурный подход к динамическому инструментированию. Идея подхода заключается в том, чтобы вообще не транслировать динамически те модули программы, которые уже были инструментированы компилятором. При использовании комбинированного инструментирования можно предполагать, что компиляторная инструментация выполнена таким образом, который позволяет избавиться от необходимости транслировать весь код. Например, при компиляции косвенных вызовов внедряются необходимые вызовы диспетчера переходов динамической системы инструментирования. Показывается, что благодаря использованию такого подхода удается ускорить короткие тесты больших программ более чем в 10 раз по сравнению с первоначальной версией DynamoRIO.

В **четвертой главе** описывается практическое применение динамических детекторов для регулярного автоматического тестирования браузера Chromium. Благодаря использованию этих детекторов за четыре года удалось обнаружить более 2500 ошибок, в том числе десятки критических и сотни опасных. Часть ошибок была найдена в используемых проектом сторонних и системных библиотеках. Найти столько ошибок вручную в проекте, размер исходного кода которого превышает 500 МБ, было бы весьма затруднительно и потребовало бы очень много трудозатрат.

Учитывая огромный размер кода проекта Chromium, а также его разнообразие, можно с достаточной уверенностью утверждать, что предложенный подход к тестированию проекта, а также внесенные в детекторы доработки подойдут для тестирования большого количества других проектов меньшего размера.

Рассматривается организация системы регулярного тестирования Chromium, состоящая из координирующих узлов и тестирующих компьютеров, а также типовые сценарии работы с ней. Описывается роль разработчиков-*шерифов*, которые по очереди следят за статусом тестирующей системы и при необходимости исправляют ошибки, вносимые в репозиторий проекта.

Описываются изменения, которые потребовалось внести в детекторы, чтобы добиться эффективной работы в рамках тестирующей системы. Для всех детекторов, у которых бывают ложные срабатывания, была добавлена поддержка правил подавления сообщений об ошибках, подобных применяемым в Memcheck. Были разработаны автоматические генераторы правил подавления и утилита, позволяющая шерифам проверять работоспособность новых написанных правил подавления без необходимости перезапускать тесты. Это позволило упростить и ускорить процесс работы шерифов.

Рассматриваются проблемы, возникшие при переходе от запуска тестов на одном тестирующем компьютере к параллельному запуску на N тестирующих компьютеров, в частности дублирование похожих сообщений об ошибках и увеличение частоты нестабильных от запуска к запуску сообщений об ошибках. Сообщения об ошибках обрабатываются таким образом, что из них удаляется лишняя информация, которая может зависеть от условий конкретного запуска (например, адреса переменных), а затем из них удаляются дубликаты. Если

какое-то сообщение об ошибке ранее не наблюдалось, то авторам последних изменений в коде, которые могли внести эту ошибку, отправляется уведомление.

Применение детекторов на таких крупных программных проектах, как Chromium, порой приводит к тому, что в самих детекторах обнаруживаются ошибки или недочеты. Демонстрируется неудобство стандартных типов правил подавления сообщений об ошибках детектора Memcheck при масштабном тестировании программ, работающих на разных ОС, а также при использовании различных компиляторов; вместо них предлагаются более простые и гибкие.

Показывается, что применение детекторов утечек памяти при тестировании больших 32-битных программ связано со значительным количеством пропусков ошибок, если возможные утечки памяти не считаются ошибками. Большое количество ошибок второго рода чревато также трудностью сопоставления новых отчетов об ошибках с недавними изменениями в репозитории проекта.

Показывается, что детектор Memcheck отличается большим количеством ложных срабатываний при поиске возможных утечек в коде на языке C++. Исправляются причины ложных сообщений о возможных утечках на объектах типов `std::string`, `std::vector` и других, память для которых выделена оператором `new[]`.

Также в результате доработки алгоритма поиска утечек памяти удалось ускорить тестирование крупных приложений с применением детектора Memcheck на 10 %.

В **заключении** сформулированы основные результаты диссертационной работы. Результаты и положения, выносимые на защиту, совпадают.

В **приложении** подробно иллюстрируется работа предложенного алгоритма поиска ошибок типа «состояние гонки». Результаты работы алгоритма сопоставляются с результатами работы основных ранее известных алгоритмов. Приводятся примеры наиболее распространенных ошибок этого типа, найденных при использовании детектора ThreadSanitizer. Рассматриваются основные причины ложных срабатываний алгоритма ThreadSanitizer и описывается как их удаётся избежать при использовании динамических аннотаций.

Список публикаций автора по теме диссертации

1. **Serebryany K., Iskhodzhanov T. ThreadSanitizer: data race detection in practice // ACM International Conference Proceeding Series — 2009 — P. 62–71.**

2. **Serebryany K., Potapenko A., Iskhodzhanov T., and Vyukov D. Dynamic race detection with LLVM compiler // Runtime Verification — Lecture Notes in Computer Science. V. 7186 — 2012 — P. 110–114.**

3. **Iskhodzhanov T., Kleckner R., Stepanov E. Combining compile-time and run-time instrumentation for testing tools // Programmnye produkty i sistemy — 2013 — no. 3 (103) — P. 224–231.**

4. Исходжанов Т. Р., Серебряный К. С. Эффективный динамический анализ корректности синхронизации многопоточного кода с применением гибридного алгоритма // Труды 52–й научной конференции МФТИ. Часть VII. Управление и прикладная математика. Т. 3 — М.–Долгопрудный, 2009 — С. 15–18.

5. Пименов М. Н., Исходжанов Т. Р., Вьюков Д. С. Определение состояний гонки в языке программирования Go // Труды 54–й научной конференции МФТИ. Управление и прикладная математика. Т. 2. — М.–Долгопрудный–Жуковский, 2011, — С. 67–68.

(Полужирным шрифтом отмечены публикации в журналах из перечня ВАК)

Личный вклад соискателя в работах заключается в следующем:
[1,2,4,5] — разработка и реализация алгоритма поиска состояний гонок на основании анализа распространенных состояний гонок;
[1] — анализ производительности детектора;
[3] — исследование возможностей комбинированного инструментирования и разработка инструментов тестирования, основанных на этом подходе; анализ производительности систем инструментирования и разработка предложений по повышению производительности системы инструментирования DynamoRIO.