На правах рукописи

Андрианов Павел Сергеевич

Анализ корректности синхронизации компонентов ядра операционных систем

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

Автореферат
диссертации на соискание учёной степени
кандидата физико-математических наук

Москва — 2021

Работа выполнена в Федеральном государственном бюджетном учреждении науки «Институт системного программирования им. В.П. Иванникова Российской академии наук» и в Федеральном государственном автономном образовательном учреждении высшего образования «Московский физико-технический институт (национальный исследовательский университет)».

Научный руководитель: **Хорошилов Алексей Владимирович**, кандидат физико-математических наук

Официальные оппоненты: **Галатенко Владимир Антонович,** доктор физ.-мат. наук, старший научный сотрудник, Федеральное государственное учреждение "Федеральный научный центр Научно-исследовательский институт системных исследований РАН", заведующий сектором

Климов Юрий Андреевич, кандидат физико-математических наук, Федеральное государственное учреждение "Федеральный исследовательский центр Институт прикладной математики им. М.В. Келдыша РАН", старший научный сотрудник

Ведущая организация: Федеральное государственное учреждение Федеральный исследовательский центр Информатика и управление Российской академии наук

Защита состоится 25 мая 2021 г. в 11 часов на заседании диссертационного совета Д 002.087.01 при Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В.П.Иванникова РАН по адресу: 109004, г. Москва,ул. А. Солженицына, дом 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Институте системногопрограммирования им. В. П. Иванникова РАН.

Автореферат разослан _____ 2021 года.

Ученый секретарь
диссертационного совета Д 002.087.01,
кандидат физ.-мат. наук Зеленов С. В.

Общая характеристика работы

Актуальность темы исследования. Информационные технологии являются важной составляющей инфраструктуры современного общества. Они позволяют автоматизировать различные процессы жизнедеятельности человека и обеспечить возможность коммуникации. Таким образом, задача повышения производительности компьютерных систем становится чрезвычайно важной. Развитие технологий многоядерности и многопоточности являются важнейшими направлениями решения данной задачи. Например, в ядре операционной системы может параллельно выполняться большое число (несколько десятков) совершенно различных активностей: обработчики прерываний, системные вызовы от пользовательских программ, драйверы внешних устройств, внутренние службы ядра, например, планировщик. Только за счет использования многопоточности современные системы могут обеспечивать необходимые показатели производительности.

В дополнение к ошибкам, которые встречаются во всех видах программного обеспечения, многопоточные программы могут содержать специфические ошибки, связанные с параллельным выполнением: например, состояния «гонки» и состояния взаимной блокировки. В общем случае состоянием гонки называют ситуацию, при которой поведение программы зависит от порядка или времени выполнения некоторых неконтролируемых событий. Важное уточнение заключается в том, что такое выполнение не всегда является ошибкой. Проблемы возникают тогда, когда разработчик не предусматривает некоторое из возможных поведений программы. Часто рассматривают более узкий класс – «состояния гонки по данным». Эта ситуация возникает при одновременном доступе к данным из разных потоков (процессов). Одновременное чтение данных из нескольких потоков не может привести к недетерминированным результатам, и состояния гонки по данным становятся опасными в случае, если имеет место хотя бы один доступ на запись в разделяемую область памяти. В этом случае результирующее значение переменной зависит от порядка выполнения инструкций, а в параллельно выполняемом коде, в общем случае, последовательность выполнения инструкций не определена.

Состояния взаимных блокировок являются вторым большим классом ошибок в многопоточных программах. Они возникают при некорректном использовании блокирующих механизмов синхронизации. В этом случае некоторые потоки системы находятся в ожидании некоторого разблокирующего действия от других потоков и не могут продолжить свое выполнение.

Оба класса ошибок так или иначе связаны с некорректной синхронизацией параллельно выполняющихся потоков. Поиск и исправление таких ошибок, связанных с параллельным исполнением кода, затруднен из-за необходимости анализировать все возможные сценарии взаимодействия потоков, а также из-за случайного характера их проявления, так как ошибка может проявляться очень редко. Это связано с тем, что для проявления ошибки необходимы некоторая конкретная последовательность и порядок действий различных потоков. В системном программном обеспечении могут использоваться не только обыкновенные примитивы синхронизации, но и некоторые специальные, например, запреты прерываний. Для упрощения поиска таких ошибок рарабатываются различные автоматизированные подходы. Но для анализа системного программного обеспечения, например, операционных систем, требуются специализированные инструменты, которые будут учитывать их особенности.

Обычно выделяют два класса подходов к анализу программ: статические, которые проводят анализ исходного кода программы, и динамические, которые анализируют поведение программы в процессе ее выполнения. Каждый имеет свои достоинства и недостатки, например, методы динамического анализа обычно характеризуются высоким уровнем истинных предупреждений, в то время как статические методы позволяют покрыть больший объем кода. На данный момент не существует универсального подхода для анализа программ.

В большинстве прикладных пользовательских программ может быть достаточно провести тщательное тестирование, возможно, с помощью инструментов динамического анализа, чтобы проверить основные сценарии поведения программы. В случае же критически важного ПО, в т.ч. системного программного обеспечения, цена пропущенной ошибки может быть

слишком велика. Кроме того, некоторые сценарии поведения программы слишком сложно воспроизвести при реальном выполнении, что затрудняет использование динамического анализа. Поэтому применения только динамического анализа недостаточно, необходимо применять и методы статического анализа.

Методы статической верификации способны обеспечить доказательство отсутствия ошибок в некоторых заранее заданных предположениях, но при этом традиционно испытывают сложности при масштабировании на большие объемы исходного кода. Как уже упоминалось ранее, к специфике системного ПО относятся большой объем сильно связанного кода, большое количество активностей, которые могут выполняться параллельно, использование низкоуровневых операций, например, адресной арифметики, специальные примитивы синхронизации и др. Таким образом, разработка масштабируемого метода статической верификации для поиска состояний гонки с учетом специфики системного ПО является актуальной задачей.

Цели и задачи. Целью данной работы является разработка метода поиска состояний гонок для анализа корректности синхронизации, который будет масштабироваться на большие объемы кода, будет обладать приемлемым уровнем ложных предупреждений и будет учитывать специфику ядра операционных систем.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Разработать метод поиска состояний гонки, на основе подхода с раздельным анализом потоков.
2. Разработать алгоритм построения окружения потока, который будет обладать конфигурируемостью и масштабироваться на большие объемы исходного кода, и доказать его корректность.
3. Модернизировать алгоритм адаптивного статического анализа (англ. Configurable Program Analysis, CPA) с целью обеспечения возможности выполнять верификацию программого обеспчения с раздельным анализом потоков и доказать его корректность.
4. Реализовать разработанные алгоритмы.

5. Провести эксперименты на приложениях и ядрах операционных систем и сравнить результаты с другими инструментами статической верификации.

Научная новизна:

1. Метод поиска состояний гонки на основе раздельного анализа потоков, использующий средства абстракции состояний и переходов для управления точностью и ресурсоемкостью верификации.
2. Алгоритм построения окружения потока, который позволяет гибко настраивать уровень абстракции над взаимодействием потоков, и доказательство его корректности.
3. Новый алгоритм, который является обобщением существующего алгоритма статической верификации программ при помощи метода CPA, расширяющий типовой набор CPA-анализаторов средствами верификации многопоточных программ с раздельным анализом потоков, и доказательство его корректности.

Теоретическая и практическая значимость работы. Развитие метода адаптивного статического анализа для обеспечения возможности его использования вместе с подходом раздельного анализа потоков является важным теоретичеким вкладом в развитие программной инженерии. Разработка нового алгоритма построения окружения потока является важным развитием подхода с раздельным анализом потоков для обеспечения возможности применения к большому объему исходного кода. Практическим результатом данного исследования является множество найденных ошибок как в операционных системах реального времени, так и в драйверах устройств операционной системы Linux.

Методология и методы исследования. При проведении работы были использованы как теоретические методы исследования (анализ, формализация, абстрагирование), так и практические (эксперименты, сравнение).

Положения, выносимые на защиту:

1. Метод поиска состояний гонки на основе раздельного анализа потоков, использующий средства абстракции состояний и переходов для управления точностью и ресурсоемкостью верификации.

2. Алгоритм построения окружения потока и верификации многопоточных программ с помощью подхода с раздельным анализом потоков и доказательство его корректности.
3. Обобщение алгоритма статической верификации программ при помощи метода CPA, расширяющий типовой набор CPA-анализаторов средствами верификации многопоточных программ с раздельным анализом потоков, и доказательство его корректности.

Апробация результатов. Основные результаты работы докладывались на:

- Международная научно-практическая конференция «Инструменты и методы анализа программ» (TMPA: Tools and Methods for Program Analysis), Кострома, 2014 г.
- Международный семинар разработчиков CPAchecker, Москва, 2015 г.
- Летняя научная школа компании Microsoft (Microsoft Summer School), Кэмбридж, Англия, 2015 г.
- Научно-практическая Открытая конференция ИСП РАН, Москва, 2016 г.
- Научно-исследовательский семинар лаборатории «Software and Computational Systems Lab» Университета Пассау, Германия, 2016 г.
- Международная научно-практическая конференция «Инструменты и методы анализа программ» (TMPA: Tools and Methods for Program Analysis), Москва, 2017 г.
- Международный семинар разработчиков CPAchecker, Падерборн, Германия, 2017 г.
- Международный семинар разработчиков CPAchecker, Москва, Россия, 2018 г.
- Соревнования по статической верификации SV-COMP, Прага, Чехия, 2019 г.
- Международный семинар разработчиков CPAchecker, Фрауенинзель, Германия, 2019 г.

– Семинар «Математические вопросы информатики» Мехмат МГУ, Москва, Россия, 2019 г.

Публикации. Основные результаты по теме диссертации изложены в 8 печатных изданиях [1–8], 6 из которых изданы в журналах, рекомендованных ВАК [1–6], из них 3 находится в базах Scopus и Web of Science [2; 3; 6], 2 — в тезисах докладов [7; 8].

В статье [4] автором описана основная идея метода (глава 3) и его реализация (глава 5). В статье [5] автором написаны разделы, посвященные общей идее метода (глава 3), его реализации (глава 4), процессу уточнения (глава 5) и анализу потоков (глава 6). В статье [6] автором написаны разделы, посвященные разработанному методу и его реализации (главы 3–6). В статье [7] автором написаны разделы, в которых описывается ключевые особенности метода (главы 3–5). В статье [8] автором написаны разделы, посвященные разработанному методу (главы 3–5). В статье [3] автором написаны разделы, описывающие теорию масштабируемого метода (главы 3–8).

Личный вклад автора. Все представленные в диссертации результаты получены лично автором.

Объем и структура работы. Диссертация состоит из введения, четырех глав, заключения и двух приложений. Полный объем диссертации 220 страниц текста с 23 рисунками и 28 таблицами. Список литературы содержит 98 наименований.

Содержание работы

Во **введении** обосновывается актуальность исследований, проводимых в рамках данной диссертационной работы, формулируется цель, ставятся задачи работы, излагается научная новизна и практическая значимость представляемой работы. В первой главе приводится обзор научной литературы по изучаемой проблеме, во второй главе описывается теоретические основы метода, доказывается его корректность в некоторых предположениях. В третьей главе описывается схема реализации и архитектура разработанного инструмента. В четвертой главе представлены результаты

апробации на частных примерах: как на множестве тестовых примеров, так и на исходном коде реальных операционных систем.

Первая глава посвящена обзору существующих методов поиска ошибок использования различных примитивов синхронизации в многопоточном программном обеспечении. В разделе 1.1 кратко определяются основные методы поиска ошибок: экспертиза кода, статические методы, динамические методы и формальные методы. Далее приводятся основные критерии инструментов верификации, которые являются важными для решения задачи проверки корректной синхронизации системного программного обеспечения: точность, масштабируемость и поддержка сложных конструкций языка Си.

В разделе 1.2 определяются общие термины, которые будут использованы при проведении обзора: многопоточная программа, доступ к данным, алгоритмы Lockset и Happens-Before для поиска состояний гонки и другие.

В разделе 1.3 рассматриваются методы динамического анализа. В разделе рассматриваются основные подклассы методов динамического анализа: методы на основе векторных часов, статико-динамические методы, узкоспециализированные методы. Методы динамического анализа на основе векторных часов применяются для поиска ошибок синхронизации во время работы программы, поэтому одна из основных характеристик таких методов – это замедление целевой программы. Статико-динамические методы используют предварительный анализ кода для получения первичного множество предупреждений, которое необходимо проверить при реальном выполнении программы. Узкоспециализированные методы обычно нацеливаются либо на ошибки специального типа, например, ad-hoc синхронизации, либо на программное обеспечение специального вида, например, MySQL сервер. Основной вывод заключается в том, что методы динамического анализа не являются полными, т.е. достаточными для проверки для проверки системного программного обеспечения.

В разделе 1.4 рассматриваются методы статического анализа. Основной задачей таких методов является нахождение ошибок при как можно меньшем числе ложных предупреждений и наименьших затратах ресурсов.

Многие из методов статического анализа являются корректными, то есть способны не пропускать ошибки, но в этом случае процент ложных срабатываний будет слишком большой. Большое количество ложных срабатываний сильно затрудняет ручной анализ предупреждений, поэтому зачастую применяются различные фильтры, которые сокращают число ложных сообщений об ошибках. Однако, такие фильтры являются лишь неточными эвристиками, которые снижают корректность метода в целом и способны привести к пропуску реальных ошибок. Таким образом, методы статического анализа также не являются полными, т.е. достаточными для проверки системного программного обеспечения.

В разделе 1.5 рассматриваются различные методы статической верификации (англ. software model checking). Такие методы основаны на том, что автоматически строится некоторая формальная модель программы, а затем эта модель проверяется на соответствие заданным свойствам. Такие методы являются достаточно точными при условии, что модель достаточно хорошо соответствует исходной программе. Одним из важных минусов таких методов являются высокие требования к ресурсам. Другим минусом статической верификации является то, что на реальном программном обеспечении достаточно сложно достигнуть высокого уровня соответствия модели и исходной программы, что приводит к большому количеству ложных срабатываний. К плюсам можно отнести возможность формального доказательства, что в программе отсутствуют дефекты определенного типа, опять же, при условии адекватности построенной модели.

В разделе 1.6 подводятся основные итоги обзора и делаются выводы. Результаты обзора позволяют заключить, что основные усилия сейчас сосредоточены на анализе пользовательских приложений. Применение методов динамического анализа к системному программному обеспечению осложняется тем, что требуется трудоемкая настройка тестового окружения. Кроме того, методы динамического анализа не способны обеспечить гарантию отсутствия ошибок. Методы статического анализа успешно применяются к любым объемам кода любой сложности. В случае применения таких методов к большому объему сложного кода будет получено огромное

количество предупреждений. Анализ этих предупреждений вручную может потребовать большого количества времени. Методы статической верификации способны дать гарантию отсутствия дефектов определенного типа, в некоторых разумных, заранее известных предположениях. Однако, такие методы не способны в настоящее время успешно применяться к большим объемам исходного кода.

Таким образом, можно заключить, что в настоящее время отсутствуют такие методы анализа больших объемов системного кода, в том числе, операционных систем, которые могут обеспечить высокий уровень надежности.

Вторая глава посвящена описанию разработанного метода поиска состояний гонки. Основная идея метода заключается в том, что каждый поток в программе анализируется независимо от остальных. В этом случае удастся избежать комбинаторного взрыва состояний, который бы неизбежно присутствовал, если бы анализ проводился с учетом всех взаимодействий между потоками. Для того, чтобы обеспечить достаточно высокую точность метода, поток анализируется в некотором окружении, которое формируется с учетом других потоков программы. Уровень точности окружения, который требуется для анализа, может гибко настраиваться.

Метод с раздельным анализом потоков может внутри себя использовать уже существующие техники и подходы, например, такие как CEGAR. В этом случае необходимо предложить обобщенный алгоритм анализа программ, частным случаем которого уже будут как классические подходы к анализу последовательных программ, так и различные подходы к анализу многопоточных программ, в частности, предложенный подход с раздельным анализом потоков.

Для того чтобы доказать корректность предложенного подхода, необходимо определить формальную семантику программы, то есть ее математическую модель. С помощью предложенного подхода строится некоторая абстракция этой математической модели. Адекватность такой абстракции определяется тем, что каждое из возможных поведений модели должен присутствовать и в абстракции. Таким образом, ошибка не может быть пропу-

щена. Именно в этом смысле далее используется термин корректность анализа (англ. soundness).

В разделе 2.1 описана семантика многопоточных программ. Формальная модель программы определяется, как множество конкретных состояний программы, которые описываются значениями переменных каждого из потоков, а также глобальных переменных, статусом примитивов синхронизации и информацией об активных потоках. Далее описывается семантика всех поддерживаемых операторов программы, то есть задаются правила преобразования конкретных состояний операторами программы. В число поддерживаемых операторов входят операторы условия (англ. assumption), присваивания (англ. assignment), кроме того, специальные операторы работы с примитивами синхронизации: захват блокировки и ее освобождение, а также создание потока (thread_create). В заключении даны определения ошибки в программе, в частности, определение состояния гонки.

В разделе 2.2 описан адаптивный статический анализ (англ. Configurable Program Analysis, CPA), который является формальной математической моделью некоторого статического анализа программы. CPA определяется доменом абстрактных переходов и операторами над этими абстрактными переходами: оператором перехода $transfer$, оператором объединения $merge$ и оператором останова $stop$. Каждый из этих операторов должен удовлетворять некоторым условиям для обеспечения корректности. Одним из важных отличий данного определения от классического варианта теории является использование абстрактных переходов вместо абстрактных состояний. Это необходимо для возможности построения абстракции не только над конкретными состояниями, но и над конкретными дугами, то есть операциями программы. Еще одним важным отличием данного определения является ослабление условия на оператор $transfer$, что позволяет определять более сложные CPA.

В разделе 2.3 описан обобщенный алгоритм вычисления множества достижимых переходов (алгоритм 1).

Одним из важных особенностей предложенного обобщенного алгоритма является то, что его абстрактные переходы являются частичными, то

Data: адаптивный статический анализ $\mathbb{D} = (D, \Pi, \rightsquigarrow, merge, stop, prec)$, начальный абстрактный переход e_0 с точностью $\pi_0 \in \Pi$, множество $reached$ элементов из $E \times \Pi$, множество $waitlist$ элементов из $2^{E \times \Pi}$

Result: множество достижимых состояний $reached$

$waitlist := \{(e_0, \pi_0)\}$;
$reached := \{(e_0, \pi_0)\}$;
while $waitlist \neq \emptyset$ **do**
    pop (e, π) from $waitlist$;
    for e' : $(e, \pi) \overset{reached}{\rightsquigarrow} e'$ **do**
        $(\widehat{e}, \widehat{\pi}) = prec(e', \pi, reached)$;
        for $(e'', \pi'') \in reached$ **do**
            $e_{new} = merge(\widehat{e}, e'', \widehat{\pi})$;
            if $e_{new} \neq e''$ **then**
                $waitlist := waitlist \setminus \{(e'', \pi'')\} \cup \{(e_{new}, \pi'')\}$;
                $reached := reached \setminus \{(e'', \pi'')\} \cup \{(e_{new}, \pi'')\}$;
            end
        end
        if $!stop(\widehat{e}, reached, \widehat{\pi})$ **then**
            $waitlist := waitlist \cup \{(\widehat{e}, \widehat{\pi})\}$;
            $reached := reached \cup \{(\widehat{e}, \widehat{\pi})\}$;
        end
    end
end

Algorithm 1: Алгоритм $CPA(\mathbb{D}, e_0, \pi_0)$

есть множество конкретных переходов программы, соответствующее некоторому частичному абстрактному переходу, может зависить не только от этого перехода, но и от других частичных переходов.

Сам алгоритм остается практически неизменным по сравнению с классическим вариантом, за исключением небольшого изменения оператора $transfer$, что является необходимым для обеспечения возможности рассмотрения переходов из любого множества абстрактных состояний. Основное отличие возникает при доказательстве корректности этого алгоритма, то есть, утверждения о том, что построенное множество абстрактных переходов аппроксимирует сверху множество конкретных переходов. Доказательство этой теоремы приведено в приложении.

В разделе 2.4 кратко описана схема использования и конфигурации CPA. Настройка, или конфигурация, CPA позволяет добиться требуемого баланса между скоростью и точностью анализа. Комбинация различных CPA между собой позволяет нивелировать различные недостаки одних CPA с помощью других, что позволяет повысить точность анализа.

В разделе 2.5 представлен алгоритм, реализующий функциональность подхода с раздельным анализом потоков, в том числе, построение окружения. Этот алгоритм не только рассматривает обычные переходы (т.н. *переходы в потоке*), но и строит специальные *переходы в окружении*, которые представляют собой влияние потоков друг на друга. Для описания этого алгоритма используется набор операторов: оператор проекции $\cdot|_p$, оператор составления перехода $compose$ и оператор проверки совместности $compatible$.

Переходы в окружении моделируют изменение состояния отдельного потока в результате действий другого потока. Для построения окружения используется оператор проекции, который представляет как выглядит тот или иной переход для другого потока. Таким образом, если переход не модифицирует разделяемые данные, его проекцией будет тождественный переход, означающий, что он не может повлиять на состояния других потоков. Каждая полученная проекция должна быть применена к каждому переходу потока, если это возможно. Возможность применения задает оператор

$compatible$, который, по сути, проверяет могут ли проекция и переход в потоке выполняться параллельно. Обычно, эта проверка означает то, что два частичных состояния могут соответствовать одному конкретному состоянию. В частности, отсюда следует, что все значения разделяемых данных должны совпадать.

Кроме того, в разделе рассмотрен предельный случай CPA, который является инвариантным к переходам в окружении. Такое свойство позволяет значительно повысить скорость анализа за счет применения более эффективных оптимизаций.

В разделе 2.6 показано, как классический вариант подхода с раздельным рассмотрением потоков без абстракции может быть выражен с использованием предложенной теории. Это демонстрирует ее выразительность.

В разделе 2.7 представлено формальное определение CompositeCPA, который обеспечивает композицию различных CPA между собой. Его домен представляет собой декартово произведение доменов внутренних CPA, а все операторы используют параллельную композицию соответствующих операторов внутренних CPA. Кроме того, CompositeCPA может усиливать переходы одних CPA за счет информации из других. Для этого служит специальный оператор $strengthen$. В данном случае, если некоторый CPA не может определить единственную CFA дугу для следующего перехода, данный CPA может подсказать ему, используя информацию о том, по какой дуге будут переходить следующие CPA. Такая реализация оператора $strengthen$ является достаточно тривиальной и в дальнейшем может быть усилена.

Разделы 2.8, 2.9, 2.10 посвящены описанию различных вариантов ThreadCPA, который отслеживает множество активных потоков. Простой его вариант может лишь отличить один поток от другого, но не может определить, может ли он выполняться параллельно, что приводит к снижению точности анализа. Вариант с использованием эффектов окружения позволяет учитывать зависимости по созданию потоков, а значит, позволяет вычислять совместные переходы более точно, однако не является инвариантным к переходам в окружении. Наконец, расширенный вариант ThreadCPA поз-

воляет обеспечить необходимую точность анализа при сохранении инвариантности к переходам в окружении.

Раздел 2.11 представляет формальную модель LocationCPA, который отвечает за синтаксическую достижимость точек программы. Данный CPA обеспечивает связь с исходным кодом программы, а также за определение следующих переходов, которые задаются CFA дугами.

Раздел 2.12 посвящен описанию PredicateCPA, который реализует анализ предикатов в случае подхода с раздельным рассмотрением потоков. Кроме ожидаемых изменений, связанных с дополнительными операторами, и поддержки переходов в окружении, было необходимо обеспечить возможность анализа нескольких потоков, исполняющих одну функцию. Это значит, что при переходах в окружении необходимо переименовывать все встречающиеся локальные переменные для избежания коллизии имен.

Раздел 2.13 представляет описание LockCPA, который отслеживает множество испльзуемых примитивов синхронизации. Данный CPA является инвариантным к переходам в окружении, и его проекции служат лишь для повышения точности оператора $compatible$, так как два перехода не могут выполняться параллельно друг с другом, если был захвачен одна и та же блокировка.

Раздел 2.14 описывает анализ явных значений ValueCPA, который может отслеживать лишь присваивания явных значений в переменные.

В разделе 2.15 представлен основной метод поиска состояний гонки. Он основан на алгоритме Lockset, который определяет состояние гонки, как ситуацию, при которой происходит параллельный доступ к разделяемой памяти с непересекающимся множеством блокировок. Предложенный метод использует оператор $compatible$ и определяет состояние гонки, как пару *совместных* переходов, которые производят параллельный доступ к разделяемой памяти. Такой метод сводится к алгоритму Lockset при использовании только LockCPA, однако при анализе реальных программ обычно применяются несколько различных CPA совместно, что позволяет повысить точность определения состояний гонки по сравнению с классическим алгоритмом Lockset.

Третья глава посвящена описанию реализации разработанного метода поиска состояний гонки. Далее будем называть инструмент, реализующий предложенный метод, CPALockator. В этой же главе представлены решения, которые используются для эффективного анализа программного обеспечения.

В разделе 3.1 представлена общее устройство инфраструктуры CPAchecker: используемый парсер, набор различных алгоритмов и CPA, а также автоматный способ задания ошибки.

В разделе 3.2 описана конфигурация и типовой набор CPA, которые включаются в инструмент CPALockator. Обычно используются следующие CPA: ThreadModularCPA, ARGCPA, CompositeCPA, LocationCPA, CallstackCPA, LockCPA, ThreadCPA, PredicateCPA. Однако, такая конфигурация не является единственно возможной, и, в зависимости от целевой задачи, возможно исключение или добавление других CPA.

Раздел 3.3 представляет основные оптимизации, сделанные при реализации ThreadModularCPA. Основная оптимизация заключается в переходе от операций над переходами в окружении к операциям над проекциями, которых значительно меньше. А уже после операций $merge$ и $stop$ над проекциями вычисляются переходы в окружении.

Раздел 3.4 описывает одну из важных оптимизаций BAMCPA, которая основана на кэшировании результатов проведенного анализа некоторого абстрактного блока. Такая оптимизация существенно позволяет ускорить анализ программы, однако в данный момент она не позволяет использовать CPA, которые не являются инвариантными к переходам в окружении. Данное ограничение является техническим, и в дальнейшем возможна реализация данной оптимизации в общем варианте.

Раздел 3.5 описывает служебный ARG CPA, который обеспечивает построение абстрактного графа достижимости (англ. Abstract Reachability Graph, ARG). Абстрактные переходы данного CPA содержат в себе связи, соответсвующие операторам. Например, связи parent-child означает, что дочерний переход был получен из родительского с помощью оператора $transfer$.

Раздел 3.6 представляет основные отличия реализации LockCPA от формально описанной модели. Так как в реальных программах блокировки часто используются по указателю, то одним из основных отличий реализации от теории является поддержка возможности работы с указателями. Использование анализа алиасов является неоправданным для сложного программного обеспечения, поэтому делается предположение, что работа с блокировками ведется одинаковым образом, и если одна блокировка была захвачена по одному указателю, то она будет освобождаться с использованием этого же указателя. Еще одной возможностью стала поддержка расширенного множества операций над блокировками, в том числе рекурсивный захват. Также в этом разделе описаны особенности реализации, связанные с оптимизацией BAM.

В разделе 3.7 описаны основные особенности реализации LockationCPA, которые заключаются, в основном, в представлении различных служебных дуг, необходимость в которых появилась только для реализации подхода с раздельным рассмотрением потоков.

Раздел 3.8 представляет описание реализации ThreadCPA. Основным отличием реализации является отсутствие идентификаторов потока на практике. Одним из вариантов решения данной проблемы является использование в качестве идентификаторов имя переменной-описателя потока, однако она может быть переприсвоена даже при активном потоке, что является нетипичной ситуацией, однако не является ошибкой. Другое важное ограничение заключается в ограниченной поддержке операций типа $thread\_join$, которые не были описаны в теории.

Раздел 3.9 посвящен реализации PredicateCPA. Анализ предикатов традиционно является самым медленным, то есть именно его операторы тратят большую часть времени анализа программы по сравнению с операторами других CPA. Поэтому задача повышения его эффективности становится очень актуальной. В разделе описаны используемые оптимизации, которые используются для повышения скорости анализа различными способами.

Одной из важных оптимизаций является настраиваемое кодирование блоков, которое позволяет сократить количество пересчетов абстракции. Однако, подход с раздельным рассмотрением потоков накладывает определенные ограничения на применение этой оптимизации. Другим важным вопросом является представление эффектов окружения таким образом, чтобы можно было эффективно строить логические формулы, соответствующие примененному эффекту. Основной проблемой при этом является вычисление корректных SSA-индексов, которые должны соответствовать индексам в состоянии потока.

Уточнение абстракции по контрпримерам также имеет некоторые особенности из-за подхода с раздельным анализом потоков. В первую очередь возникают проблемы при восстановлении глобального пути, а также при его представлении, так как эффект окружения может не описываться ни одной CFA дугой. Многие эти проблемы требуют радикального переосмысления механизма уточнения в инфраструктуре CPAchecker, поэтому не все они были решены в рамках данной работы.

В разделе 3.10 описаны основные особенности реализации CompositeCPA, которые заключаются, в основном, в удобной комбинации CPA, которые реализуют подход с раздельным рассмотрением потоков, и тех, которые являются инвариантными к переходам в окружении, а значит, могут не предоставлять расширенное множество операторов. Кроме того, в разделе описаны отличия при реализации оператора $strengthen$.

В разделе 3.11 описан UsageCPA, который предоставляет дополнительные возможности для настройки инструмента на целевой код, а также для поиска высокоуровневых гонок с помощью аннотаций.

Раздел 3.12 посвящен еще одной важной оптимизации – анализу разделяемых данных, который применяется непосредственно перед основным анализом для поиска разделяемых данных. Так, если какая-то область памяти не является разделяемой, то для нее не будут выданы предупреждения в дальнейшем. Анализ является консервативным, то есть если не доказано, что какая-то область памяти является локальной, то считается, что она может быть разделяемой.

В разделе 3.13 представлены основные особенности при вычислении состояний гонки при анализе реальных программ. Важным отличием является активное использование указателей, однако использование анализа алиасов является очень неэффективным. Кроме того, анализ алиасов требует явной инициализации каждого указателя. Поэтому для решения данной проблемы была использована специализированная модель памяти BnB, основанная на разделении памяти на непересекающиеся регионы по типам. Кроме того, каждое поле структуры, от которого не брался адрес, выделяется в отдельный регион. Такая модель памяти позволяет достаточно точно вычислять потенциальные состояния гонки.

Однако сам процесс вычисления устроен сложным образом, так как необходимо сначала восстановить всю информацию о доступах к данным, потерянную из-за применения оптимизации BAM, а затем обеспечить эффективное хранение, поиск и модификацию этой информации. Для этого применяются различные оптимизации, связанные с выделением приоритетной информации и упорядочивании ее.

Раздел 3.14 содержит описание процесса уточнения при поиске состояния гонки. Так как полученная абстракция программы может является аппроксимацией сверху множества состояний программы, возможны ситуации, при которых некоторые важные детали не будут учтены. Это приведет к тому, что будет выдано ложное сообщение об ошибке. Процесс уточнения позволяет повысить степень уверенности в том, что найденная ошибка является истинной. Например, может быть проверена локальная достижимость каждого из путей, участвующих в состоянии гонки. Этот процесс требует достаточно большого количества ресурсов, однако может быть прерван в любой момент.

Основным отличием от классического варианта является то, состояние гонки определяется парой доступов, соответственно, возникает необходимость уточнения пары путей. Кроме того, так как процесс уточнения запускается после полного построения абстракции, имеется большой выбор состояний для уточнения, а уточнение всех путей подряд приведет к большому количеству однообразных предикатов. Для решения этой проблемы

используются различные эвристики для определения значимости уточнения каждого конкретного пути.

В разделе 3.15 представлено описание формата визуализации результатов. Для визуализации используется формат GraphML, который уже применяется в других инструментах статической верификации. В этом разделе описан механизм построения трассы, приводящей к состоянию гонки, а также показана возможность их визуализации.

В **четвертой главе** приведены результаты экспериментальной оценке предложенного метода. Сравнение проводилось на множестве задач SV-COMP, множестве задач, подготовленных на основе подсистемы *drivers/net* ОС Linux, а также двух ядрах закрытых операционных систем реального времени.

В разделе 4.1 описана общая схема проведения экспериментов, описаны использованные задачи, конфигурации и машины, на которых производился запуск.

В разделе 4.2 представлены результаты сравнения различных инструментов верификации: CPALockator и участников соревнования SV-COMP. Threading является реализацией подхода с чередованиями на базе CPAchecker. Yogar-CBMC и Lazy-CSeq реализуют подходы ограничиваемой проверки моделей и секвенциализации соответственно.

Таблица 1 — Запуск на наборе задач SV-COMP

	Другие инструменты			
Подход	Yogar-CBMC	Lazy-CSeq	Threading	CPALockator
Вердикт «ошибка»	773	811	727	1028
из них корректных	773	811	727	805
из них некорректных	0	0	0	223
Вердикт «нет ошибки»	284	256	165	21
из них корректных	284	256	165	21
из них некорректных	0	0	0	0
Анализ не завершен	25	15	190	33
Время CPU (с)	7 000	29 000	111 000	23 700

Основные выводы заключаются в том, что CPALockator демонстрирует эффективную работу, однако его точность значительно ниже, чем у других инструментов. При этом общее время работы одного из инструментов меньше, чем у CPALockator, за счет быстрого решения небольших задач. Тем не менее, никто из инструментов-участников соревнования SV-COMP не решил те несколько примеров, которые основаны на драйверах ОС Linux, кроме CPALockator. Таким образом, можно заключить, что CPALockator экспериментально подтверждает то, что он не пропускает ошибок, а также то, что он эффективно решает сложные задачи.

В разделе 4.3 представлены результаты сравнения различных вариантов реализации PredicateCPA: различные варианты оператора $merge$, а также использование оптимизаций. Результаты показывают, что использование оператора $merge_{Join}$ является предпочтительным для большинства задач. Среди оптимизаций стоит выделить оптимизацию ABE, которая улучшает эффективность анализа любого исходного кода. Некоторые другие оптимизации имеет смысл применять при анализе системного программного кода, в то время как на искусственных тестах SV-COMP они значительно ухудшают показатели. Другие оптимизации позволяют значительно ускорить анализ, однако приводят к появлению новых ложных сообщений об ошибках.

В разделе 4.4 представлены результаты сравнения различных вариантов реализации ThreadCPA. Эксперименты подтверждают, что расширенный вариант ThreadCPA, инвариантный к переходам в окружении, способен обеспечить сопоставимую точность с вариантом с эффектами окружения. Тем не менее, простой анализ потоков является достаточным для тех случаев, в которых создание потоков производится искусственным образом в модели окружения.

Различные варианты обработки повторно создаваемого потока также демонстрируют различные результаты в зависимости от целевого исходного кода. Например, при анализе исходного кода ОС РВ все три способа показывают похожие результаты, в то время как на наборе SV-COMP они проявляют себя совершенно по-разному.

Общий вывод по результатам заключается в том, что по возможности нужно использовать простой вариант ThreadCPA, который является корректным при некоторых дополнительных предположениях. Однако, если есть сомнения в выполнении данных предположений для конкретного исходного кода, тогда следует использовать тот вариант, который гарантирует корректность.

В разделе 4.5 приведено сравнение различных вариантов реализации LockCPA. Сравнение различных вариантов реализации оператора $merge$ не выявило существенных отличий ни на каком наборе задач. Это демонстрирует невысокую актуальность данного варианта. Различные варианты оптимизации BAM показывают свою эффективность только на большом объеме исходного кода, однако в этом случае становятся принципиально важными. Использование уточнения, наоборот, снижает эффективность анализа на любом тестовом наборе, что подтверждает предположение, что затраты на уточнение не перекрывают выгоду от менее точной начальной абстракции.

В разделе 4.6 рассмотрено сравнение вклада в точность и эффективность анализа дополнительных CPA. Оптимизация BAM в данный момент ограничена только CPA, инвариантными к переходам в окружении. Результаты демонстрируют ее высокую эффективность, кроме того большие объемы исходного кода могут быть проанализированы только с ее помощью.

Применение анализа разделяемых данных в данной реализации имеет смысл только для решения задачи поиска состояния гонки. При этом даже для задач на основе драйверов ОС Linux анализ разделяемых данных не показывает существенного улучшения из-за анализа потоков. Но для ОС РВ анализ разделяемых данных способен существенно снизить количество ложных предупреждений об ошибке.

Анализ предикатов обеспечивает высокую точность, однако для получения легковесного варианта инструмента его можно исключить из конфигурации, это приведет к значительному повышению скорости анализа за счет снижения точности.

В разделе 4.7 представлены результаты поиска известных ошибок в драйверах ОС Linux. Для этого был проведен анализ исправлений в стабильных версиях ядра Linux, и были найдены 8 коммитов, которые исправляют ошибки, связанные с состоянием гонки.

Таблица 2 — Результаты запуска инструмента на известных ошибках в стабильных версиях ядра ОС Linux

Коммит	Результат	Комментарий
0e2400e	+	
7357404	±	Спец. конфигурация
f1a8a3f	∓	Спец. конфигурация; найдена нецелевая ошибка
1a81087	±	Найдена нецелевая ошибка
f0c626f	∓	Спец. конфигурация; найдена нецелевая ошибка
aea9dd5	–	
10ef175	–	Таймаут
4036523	–	

В двух случаях инструмент находит именно те ошибки, которые исправляются в данных коммитах. В трех случаях инструмент находит нецелевые ошибки, то есть реальные, но не те, которые исправляются в коммитах. Еще в трех случаях ошибка не была найдена из-за исчерпания лимита времени. При этом для трех задач потребовалось использовать специальную конфигурацию инструмента. Таким образом, подтверждается, что инструмент CPALockator позволяет не пропустить ошибку при некоторых предположениях.

В разделе 4.8 приведены результаты анализа основных причин ложных срабатываний инструмента.

Таблица 3 — Анализ предупреждений в модулях ядра ОС Linux

Количество	Процент	Описание
41	48%	Истинных предупреждений
25	29%	Неточность подготовки задачи
19	22%	Неточность метода
8	9%	Неточность модели памяти
7	8%	Специфика ядра ОС (прерывания)
4	5%	Неточность отдельных типов анализа

Как и следовало ожидать, основные причины ложных срабатываний связаны с некорректной подготовкой верификационной задачи сторонним компонентом и с неточной моделью памяти. Менее значимые причины включают в себя неточности различных компонентов инструмента: анализ примитивов синхронизации, анализ предикатов, анализ разделяемых данных и т. д.

В разделе 4.9 сформулированы основные выводы по результатам всех экспериментов: основные сценарии использования различных конфигураций и оптимизаций в зависимости от целевого исходного кода. Кроме того, показаны основные проблемные места на основе анализа причин ложных срабатываний и результатов поиска известных ошибок.

В **заключении** приведены основные результаты работы, которые заключаются в следующем:

1. Был разработан метод поиска состояний гонки на основе раздельного анализа потоков, использующий средства абстракции состояний и переходов для управления точностью и ресурсоемкостью верификации.
2. Был разработан алгоритм построения окружения потока, который позволяет гибко настраивать уровень абстракции над взаимодействием потоков, и доказана его корректность.
3. Был разработан новый алгоритм, который является обобщением существующего алгоритма статической верификации программ при помощи метода CPA, расширяющий типовой набор CPA-анализаторов средствами верификации многопоточных программ с раздельным анализом потоков, и доказана его корректность.

Эксперименты показали преимущества подхода на больших верификационных задачах перед существующими техниками статической верификации. Небольшие задачи со сложным взаимодействием потоков лучше решаются другими инструментами, так как предложенный подход абстрагируется от такого взаимодействия, что приводит к потере точности, существенной для небольших искусственных задач. Однако разработанный под-

ход также не пропускает ошибок (при некоторых предположениях) и может быть развит в будущем.

Таким образом, можно заключить, что основные требования к новому инструменту для анализа корректности синхронизации были выполнены, так как он успешно применяется к различным программным системам, в том числе, к драйверам ОС Linux и ядрам ОС реального времени.

Публикации автора по теме диссертации

1. *П.С. Андрианов*. Анализ корректности синхронизации компонентов ядра операционных систем // *Труды Института системного программирования РАН*. — 2019. — Т. 5, № 31. — С. 203–232.

2. *Andrianov Pavel*. Analysis of Correct Synchronization of Operating System Components // *Programming and Computer Software*. — 2020. — Vol. 46, no. 8. — P. 712–730.

3. *Andrianov Pavel, Mutilin Vadim*. Scalable Thread-Modular Approach for Data Race Detection // *Frontiers in Software Engineering Education*. — 2020. — Pp. 371–385.

4. *П.С. Андрианов, В.С. Мутилин, А.В. Хорошилов*. Метод легковесного статического анализа для поиска состояний гонок // *Труды Института системного программирования РАН*. — 2015. — Т. 5, № 27. — С. 87–116.

5. *П.С. Андрианов, В.С. Мутилин, А.В. Хорошилов*. Конфигурируемый метод поиска состояний гонок в операционных системах с использованием предикатных абстракций // *Труды Института системного программирования РАН*. — 2016. — Т. 6, № 28. — С. 65–86.

6. *Andrianov Pavel, Mutilin Vadim, Khoroshilov Alexey*. Predicate Abstraction Based Configurable Method for Data Race Detection in Linux Kernel // Tools and Methods of Program Analysis: 4th International Conference, TMPA 2017, Moscow, Russia, March 3-4, 2017, Revised Selected Papers / Ed. by

Vladimir Itsykson, Andre Scedrov, Victor Zakharov. — Cham: Springer International Publishing, 2018. — Pp. 11–23. — URL: https://doi.org/10.1007/978-3-319-71734-0_2.

7. *P.S. Andrianov, V.S. Mutilin, A.V. Khoroshilov*. An approach to lightweight static data race detection // Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering. — 2014. — Pp. 27–33.

8. *P.S. Andrianov, V.S. Mutilin, A.V. Khoroshilov*. Lightweight Static Analysis for Data Race Detection in Operating System Kernel // Proceedings of the international conference "Tools and Methods of Programs Analysis". — 2014. — Pp. 128–135.

Андрианов Павел Сергеевич

Анализ корректности синхронизации компонентов ядра операционных систем

Автореф. дис. на соискание ученой степени к. ф.-м. н.

Подписано в печать ___.___._____. Заказ № _____
Формат 60×90/16. Усл. печ. л. 1. Тираж 100 экз.
Типография _____