Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(государственный университет)»
Кафедра информатики

На правах рукописи

БИТНЕР Вильгельм Александрович

ИССЛЕДОВАНИЕ И РЕАЛИЗАЦИЯ МОДЕЛИ СТАТИЧЕСКОГО АНАЛИЗА НАХОЖДЕНИЯ СОСТОЯНИЯ ГОНКИ В МНОГОПОТОЧНЫХ АЛГОРИТМАХ С ИСПОЛЬЗОВАНИЕМ ЛИНЕАРИЗОВАННОГО ГРАФА ПОТОКА УПРАВЛЕНИЯ

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

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
доктор физико-математических наук, профессор
Тормасов Александр Геннадьевич

МОСКВА – 2014

Оглавление

Введение

Современная тенденция развития микропроцессоров и вычислительного оборудования обуславливает создание новых подходов к промышленной разработке программного обеспечения, отвечая все более высоким требованиям потребителя и современным условиям конкуренции. В частности, повышение количества ядер в микропроцессорах заставляет программных разработчиков использовать многопоточные алгоритмы и применять технологии многопоточной программной разработки не только для высокопроизводительных систем, но и для персональных компьютеров и мобильных устройств [2, 30].

Повышение количества ядер в микропроцессорах зачастую сопровождается понижением тактовой частоты каждого ядра микропроцессора, что усугубляет положение ранее написанных однопоточных программ. Далеко не редкость в современной IT-индустрии ухудшение производительности промышленных комплексов программ при переходе на более современные вычислительные системы с новыми поколениями микропроцессоров. Данный факт дополнительно стимулирует разработку многопоточных программ для достижения наилучших показателей производительности.

Разработка многопоточных программ с наилучшими показателями производительности не имеет общего подхода. Каждый случай индивидуален в зависимости от архитектуры вычислительной системы и сложности комплексов программ. Такое положение приводит к появлению трудно выявляемых ошибок в коде программ, особенно связанных с параллельным исполнением потоков. Таким образом, всегда остается актуальным разработка новых и совершенствование старых средств верификации и контроля качества программ во время разработки программного обеспечения (ПО).

Актуальность темы

Среди наиболее сложных для обнаружения ошибок в многопоточных программах являются состояния конкурентного доступа к памяти или состояния гонки (race condition, RC). *Race condition* – ситуация, когда несколько потоков одновременно обращаются к одному и тому же ресурсу, причем хотя бы один из потоков выполняет операцию записи, и порядок этих обращений точно не определен [5]. Наличие состояний гонок приводит к недетерминированному поведению программы.

Методики поиска состояний гонок обычно разделяют на *статический анализ*, *динамический анализ*, *проверку на основе моделей и аналитического доказательства корректности программ*. Динамический анализ посвящен анализу эмпирических запусков программы на определенном наборе тестов и входных данных. Существуют довольно много коммерческих и бесплатных программных утилит, которые позволяют проводить динамический анализ на разных платформах и архитектурах: Intel Thread Checker, CHESS и т.д. Недостатком данного вида анализа является невозможность анализа больших программных систем, т.к. количество необходимых тестов экспоненциально растет в зависимости от количества входных параметров.

Статический анализ – это анализ без исполнения программы. Преимуществом данного вида анализа является возможность проверки и устранения ошибок в коде во время разработки, глубина анализа как правило выше, что позволяет гарантировать более высокую надёжность конечного программного продукта. В качестве недостатка статического анализа можно выделить сложность применения в промышленных масштабах и не самую лучшую надежность методов анализа, которые не давали бы ложные срабатывания: обнаружение состояний гонок, когда их на самом деле нет, либо упущение реально присутствующих состояний гонок.

В рамках диссертационной работы за основу взята математическая модель статического анализа многопоточных программ, описанных в исследованиях А.Г. Тормасова, М.Ю. Кудрина, А.С. Прокопенко, Н.В. Заборовского [13-15]. Математическая модель строится на основе анализа графа совместного исполнения потоков, построенного по определённым участкам кода программы. Результаты исследований зарекомендовали себя как перспективный и эффективный метод поиска потенциальных угроз состояния гонки. Однако представленный метод хорошо работает на линейных участках и имеет ряд ограничений при анализе более сложных конструкций кода, таких как циклы, условные конструкции, алиасы. Реализация и исследование математической модели для применения в промышленных комплексах программ остается актуальной задачей, в рамках которой необходимо разрешить ограничения представленного метода и, как следствие, расширить класс применимости метода среди промышленного ПО.

Цель работы и задачи исследования

В рамках работы [13] уже была предпринята попытка приблизить математическую модель к промышленному использованию метода анализа. Однако реализация и соответствующая математическая модель, выполненная с использованием высокоуровневого промежуточного представления не поддерживаемой системы и не имеющая широкого использования, не позволяет применять метод в промышленной разработке ПО. С другой стороны, современная промышленная разработка ПО не обходится без использования оптимизирующих компиляторов, которые используют промежуточное представление программы (intermediate representation – IR) для проведения оптимизаций над кодом программы. Таким образом, использование IR промышленного компилятора в качестве основы для проведения анализа позволило бы значительно улучшить применимость представленного метода.

Существенную роль в методе может сыграть оптимизации компиляторов, которые могли бы создать наилучший контекст IR перед применением метода анализа многопоточных программ.

Целью диссертационной работы является исследование, адаптация и реализация математической модели статического анализа многопоточных программ.

Для достижения поставленной цели в работе решаются следующие основные задачи:

- Исследование и анализ современных оптимизирующих компиляторов и их оптимизаций. Выявление наиболее подходящих оптимизаций для создания такого контекста IR, который содержал бы наименьшее количество изменений потока управления, т.е. циклов и условных конструкций.
- Разработка дополнительных оптимизаций над графом потока управления, не влияющих на анализ контекста программы в исследуемом методе статического анализа.
- Адаптация математической модели и метода анализа под новый контекст IR.
- Реализация математической модели в контексте промышленного IR в виде автоматизированной системы обнаружения состояний гонок в программе.

Объект исследования

В работе объектом исследования является процедура поиска состояний гонок в многопоточных алгоритмах, а также оптимизация анализируемого контекста на промежуточном представлении с помощью математической модели, основанная на графах совместного исполнения потоков.

Предмет исследования

В работе предметом исследования является математическая модель, основанная на графах совместного исполнения потоков, построение которых осуществляется по коду программ на языках Си и Си++.

Метод исследования

В работе применяются и используются методы теории множеств, теории графов, теории компиляции, методы программирования на языках высокого уровня.

Научная новизна

В процессе исследования получены следующие новые научные результаты, выносимые на защиту:

- Программная реализация системы обнаружения состояния гонок в многопоточных алгоритмах на основе IR промышленного компилятора, которая позволяет получать IR программ различных языков высокого уровня, разных платформ и архитектур. Использование общепризнанного и популярного компилятора гарантирует применимость метода к широкому классу задач на долгие годы вперед. Возможность получения унифицированного IR программ, ориентированных на такие архитектуры, как Intel, ARM, Power, позволяет значительно расширить класс применимости метода в среде коммерческого ПО. Появилась возможность анализировать код программы на предмет состояния гонок на разных этапах компиляции – до применения оптимизаций и после применения оптимизаций, что дает дополнительную возможность проверки.
- Математическая модель, адаптированная под контекст нового IR.

- Метод линеаризации графа потока управления для представленного анализа. Особенность представленного метода статического анализа позволил создать дополнительную оптимизацию графа потока управления, с помощью которой удалось в некоторых случаях избавиться от лишних узлов графа потока управления и, как следствие, уменьшить количество ветвления управления на графе.
- Комплекс программ и результаты вычислительных экспериментов для выявления наличия гонок в некоторых многопоточных алгоритмах, в том числе неблокирующих.

Работа имеет принципиальную новизну, как в постановке задачи, так и в выборе методов решения поставленных задач.

Практическая ценность

Усовершенствована методика нахождения состояний гонок в многопоточных алгоритмах, реализация которых используется в коммерческом ПО. Разработанные программные модули позволили сделать методику нахождения состояний гонок в промышленном ПО более автоматизированной, надежной и универсальной, что было достигнуто благодаря использованию промежуточного представления оптимизирующего компилятора LLVM.

Выявленные и предложенные оптимизации LLVM позволили получать наиболее подходящий контекст для применения анализа с точки зрения производительности поиска состояний гонок в алгоритме.

Предложенная методика линеаризации графа потока управления, которая используется после оптимизаций LLVM и перед основным анализом, позволила расширить класс задач, пригодных для анализа на предмет наличия состояний гонок исследуемого метода.

Таким образом, программная реализация новшеств и улучшений позволила расширить класс задач, для которых применим метод статического анализа на

основе графа совместного исполнения потоков. Как следствие, данный метод анализа состояний гонок можно более широко использовать для верификации промышленного ПО.

Публикации и апробация результатов

По теме диссертации опубликовано 9 печатных работ, в том числе 3 в рецензируемых изданиях, рекомендованных ВАК РФ для опубликования основных научных результатов диссертаций.

Публикации в ведущих рецензируемых журналах, входящие в перечень ВАК:

1. Битнер В.А., Тормасов А.Г. Анализ и сокращение промежуточного представления программы в модели статического анализа нахождения состояния гонки в многопоточных алгоритмах // Вестник КГТУ им. А.Н. Туполева, 2014. –№3. – С. 203-212.
2. Битнер В.А., Тимербаев Н.Ф. Контекстно зависимая линеаризация графа потока управления в статическом анализе состояний гонок в многопоточных алгоритмах // Вестник Казанского технологического университета, 2014. – Т. 17, №15. – С. 187-192.
3. Битнер В.А., Заборовский Н.В. Построение универсального линеаризованного графа потока управления для использования в статическом анализе кода алгоритмов // Моделирование и анализ информационных систем, 2013. – Т. 20, №2. – С. 166-177.

Результаты работ докладывались автором на научных конференциях и семинарах:

1. Труды 57-й научной конференции МФТИ «Актуальные проблемы фундаментальных и прикладных наук в современном информационном обществе» (Москва, 2014).

2. Научная конференция «Математическое моделирование и информатика» при ФГБОУ ВПО МГТУ «СТАНКИН» (Москва, 2014).
3. The 8th Congress of the International Society for Analysis, its Applications, and Computation (Москва, 2011).
4. Международной заочной научно-практической конференции «Актуальные проблемы науки» (Тамбов, 2011).
5. Труды 52-й научной конференции МФТИ «Современные проблемы фундаментальных и прикладных наук» (Москва, 2009).
6. XXXVI международная молодежная научная конференция «Гагаринские чтения» (Москва, 2009).

Структура и объем диссертации

Диссертационная работа состоит из введения, трех глав, заключения, одного приложения и списка использованных источников. Список использованных источников включает 37 наименований. Общий объем работы составляет 103 страницы. Объем приложений составляет 12 страниц.

Во **введении** обоснована актуальность темы, сформулирована цель работы и определен перечень решаемых задач, указана новизна научных изысканий, отмечены особенности подхода, раскрываемого в диссертационной работе, практическая ценность полученных решений и разработок, а также дан краткий обзор содержания по главам.

В **первой главе** даются необходимые определения и понятия, которыми оперируют в работе. В данной главе приводится аналитический обзор программных разработок и литературы по теме диссертации. Проведен анализ оптимизирующих компиляторов и типов промежуточных представлений, который позволил сформулировать подход к разработке программных модулей для имплементации метода анализа состояний гонок на основе графа совместного исполнения потоков.

В первой главе при анализе оптимизирующих компиляторов делается вывод о возможности использовать и адаптировать некоторые оптимизации компилятора с целью получения наиболее удобного и эффективного представления программ, написанных на языке высокого уровня, что позволит решить задачу линеаризации CFG для повышения эффективности и расширения применимости исследуемого метода статического анализа многопоточных алгоритмов на предмет состояния гонки.

Помимо анализа оптимизирующих компиляторов в первой главе приводится обзор существующих средств поиска состояний гонок в программах. Уделяется внимание слабым и сильным качествам тех или иных подходов и технологий. Делается вывод о текущем положении статических анализаторов, что на практике они дают неточный ответ о наличии того или иного свойства в программе. Основной принцип статического анализа: либо точность, либо время. Выбор оптимального сочетания времени работы анализатора и точности является практической задачей, которая разрешается в каждом случае отдельно.

В первой главе приводится описание исследуемой математической модели поиска состояний гонок в многопоточных алгоритмах, а также вводятся необходимые определения и понятия, связанные с исследуемой моделью.

Во **второй главе** описывается реализация модели статического анализа нахождения состояния гонки в многопоточных алгоритмах с использованием линеаризованного графа потока управления. В данной главе проводится анализ оптимизаций компилятора LLVM. LLVM содержит более 100 уникальных оптимизаций, которые были проанализированы с точки зрения влияния на CFG. Также в главе рассматривается подход использования выявленных оптимизаций LLVM на практике с целью линеаризации графа потока управления.

Во второй главе проводится анализ инструкций LLVM IR, не влияющих на математическую модель поиска состояний гонок, и соответствующее доказательство их корректного удаления из анализируемого контекста программы, что приводится в утверждениях и теоремах 1-3. Как следствие, в

данной главе приводится описание метода получения сокращенного промежуточного представления программы, удаляя ненужные инструкции из контекста.

В последующем в данной главе на основе доказанных теорем и утверждений приводится описание алгоритма удаления пустых CFG-узлов, полученных на сокращенном промежуточном представлении. Описанный алгоритм используется в общей модели получения контекстно-зависимого линеаризованного графа потока управления. На основе приведенной модели приводится описание организации поиска состояния гонок на основе исследуемого метода статического анализа и линеаризованного графа потока управления.

В заключительной части второй главы приводится описание программного комплекса, разработанного в рамках данной работы, для организации с высокой степенью автоматизации систему поиска неразрешимых состояний гонок. Приводится описание архитектуры системы поиска и описание составных частей разработанного программного комплекса.

В **третьей главе** описывается практическое применения реализованного комплекса программ на различных алгоритмах. Показывается, что на модельных задачах получаются верные результаты и демонстрируется корректность подхода, используя промежуточное представление программы и его дальнейшее упрощение.

В третьей главе приводятся программный код, подробные результаты процесса получения Reduced CFG, а также результаты анализа Reduced CFG в Wolfram Mathematica. По результатам применения прототипа комплекса программ делается заключение, что подход работает корректно на модельных задачах и может быть использован в других реальных задачах с двумя потоками, компилируемых оптимизирующим компилятором CLANG&LLVM.

В **приложении** подробно иллюстрируется работа предложенной модели организации поиска состояния гонок. Приводятся промежуточные

анализируемые графы, полученные анализатором на Wolfram Mathematica, такие как классы эквивалентности на графе совместного исполнения потоков, анализ путей на расчетном графе. Приведенные данные в приложении являются дополнением к третьей главе, где приводятся практические результаты анализа различных многопоточных алгоритмов

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

Глава 1. Общие понятия и существующие методы анализа многопоточных алгоритмов для обнаружения состояний гонок

1.1. Понятие об оптимизации и промежуточном коде

Введем понятие оптимизации, которое будет использоваться в дальнейшем. *Оптимизация* – это эквивалентное преобразование над промежуточным представлением компилятора.

Промежуточное представление – структура данных, фиксирующая состояние(я) программы в процессе компиляции от исходной записи на входном языке до выходного состояния – целевого исходного кода программы, исполняемой на заданной платформе. Основные функции промежуточного представления:

- отображение и сохранение инвариантной семантики исходной программы;
- базис для проведения анализа и оптимизирующих преобразований программы;
- интерфейс взаимодействия со всеми фазами компиляции, позволяющий фиксировать и передавать изменения программы.

В процессе трансляции компилятор часто использует промежуточное представление исходной программы, предназначенное, прежде всего, для удобства генерации кода и/или проведения различных оптимизаций. Сама форма промежуточного представления зависит от целей его использования. Наиболее часто используемыми формами промежуточного представления являются ориентированный граф (в частности, абстрактное синтаксическое дерево, в том числе атрибутированное), трехадресный код (в виде троек или четверок), префиксная и постфиксная запись.

Линейным участком в промежуточном представлении будем называть совокупность операций с выделенными входной и выходной операциями,

передача управления в который может происходить только через входную операцию.

Граф потока управления [16, 25] – аналитическая структура, которую логически можно представить, как управляющую надстройку над промежуточным представлением. В реализации наиболее распространенной схемой является представление управляющего графа как отдельного объекта, имеющего взаимно однозначное соответствие с операционной семантикой промежуточного представления, в которой выражена вся полнота семантики передачи управления. Представляет собой направленный связный граф, каждый узел (вершина) которого соответствует линейному участку, а дугам – управляющие связи между ними, отображающие передачу управления. В графе есть два выделенных узла: узел START, не имеющий входных дуг, и узел STOP, не имеющий выходных дуг. Для простоты обозначения граф управления называют CFG (Control Flow Graph), а узлы и дуги графа управления – CFG-узлы и CFG-дуги, соответственно. Далее в тексте будут синонимично употребляться оба обозначения графа управления и его составляющих.

Как правило, для любой оптимизации можно выделить объект применения. *Объект применения* – это некоторое подмножество промежуточного представления, с которым работает оптимизация. Например, операция, пара операций, узел управляющего графа, цикл, процедура. Оптимизации на уровне операций принято называть низкоуровневыми оптимизациями, оптимизации, работающие на уровне узлов управляющего графа – макрооптимизациями.

Цели оптимизации можно подразделить на две категории. Первая цель состоит в том, чтобы избавиться от лишних вычислений, вторая – наиболее оптимально адаптировать программу к имеющимся архитектурным особенностям и аппаратным ресурсам. При этом обе цели не являются полностью независимыми. Иногда адаптация к аппаратным особенностям требует увеличить число выполняемых команд, чтобы сократить общее время их

выполнения, но это возможно только при наличии параллельных аппаратных ресурсов.

Программа, поступающая на вход оптимизирующего компилятора, предварительно преобразуется в вид, удобный для анализа и оптимизаций. Далее для нее строятся специальные вспомогательные структуры данных, облегчающие оба процесса. При анализе компилятор должен найти и устранить ложные зависимости, для чего зачастую используется техника переименования регистров, а при оптимизации должен избавиться от лишних вычислений, найти наиболее часто выполняемые последовательности команд, опираясь на профильную информацию, и построить для них оптимальный код, включая планирование для архитектур с параллельными исполняющими устройствами.

В данной работе будет уделено большое внимание именно к CFG и связанным с его преобразованием оптимизациям, хотя подобные аналитические структуры, как CFG, далеко не единственны в оптимизирующих компиляторах. При этом промежуточное представление исходной программы можно рассматривать как программу для абстрактной машины, поскольку оно основано на ассемблере целевой архитектуры, но при этом не является программой на языке ассемблера [6]. Данная особенность позволяет наиболее точно и оптимально по сложности использовать промежуточное представление в методе статического анализа состояний гонок в многопоточных алгоритмах, описанном в работах [13-14].

1.2. Оптимизирующие компиляторы

Оптимизирующие компиляторы занимают важную роль в современной промышленной разработке, позволяя максимально эффективно использовать аппаратные особенности платформ и архитектур, на которых предполагается исполнение программ. Разработкой собственных оптимизирующих компиляторов занимаются все ведущие IT-корпорации и компании, которые

связаны не только с инженерным развитием платформ и архитектур, но и программной разработкой. Основная мотивация к разработке собственного оптимизирующего компилятора – собственными силами эффективнее разрешить проблемы генерации высокопроизводительного и надежного целевого кода отдельно взятой архитектуры. Таким образом, на сегодняшний день существует достаточной обширный пласт коммерческих оптимизирующих компиляторов таких компаний, как Intel, Sun Microsystems, Transmeta, Microsoft, IBM, HP, Elbrus [9].

Помимо коммерческих оптимизирующих компиляторов существуют также бесплатные компиляторы с открытым кодом, которые развиваются группами разработчиков повсеместно во всем мире. Наиболее популярные и зарекомендовавшие себя оптимизирующие компиляторы – это GCC и CLANG&LLVM.

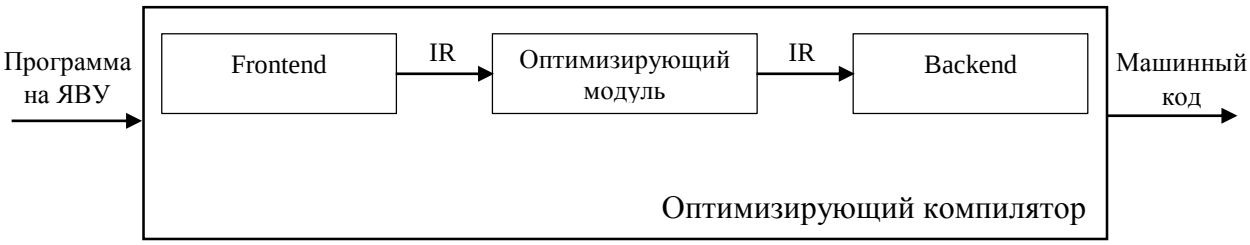


Рис. 1. Структура оптимизирующего компилятора.

Несмотря на богатое разнообразие оптимизирующих компиляторов, их структура состоит из трех абстрактных модулей (см. рис.1): «frontend» – преобразование программы, написанной на языке высокого уровня (ЯВУ), в промежуточное представление (IR – intermediate representation); «оптимизирующий модуль» – процесс анализа и оптимизации промежуточного представления программы; «backend» – генерирование оптимизированного машинного кода.

Помимо общей структуры оптимизирующих компиляторов всегда можно выявить базовые оптимизации, которые имплементированы во всех компиляторах, особенно связанные с анализом и изменением CFG, такие как

оптимизация переходов, оптимизации вызовов оптимизация циклов и цепочек переходов и т.д. [16]. Таким образом, задача линеаризации CFG для организации более эффективного статического анализа алгоритмов в рамках данной работы имеет пересечения с задачами оптимизирующих компиляторов – получения эффективного кода целевой архитектуры. При этом под *линеаризацией CFG* будем понимать процесс уменьшения количества условных переходов в графе потока управления, которые приводят к параллельным веткам исполнения кода в зависимости от условий.

Особенно факт пересечения задач компилятора и линеаризации CFG проявляется для оптимизирующих компиляторов архитектурных платформ, использующих статический подход к распараллеливанию на уровне инструкций – EPIC-архитектуры (Explicitly Parallel Instruction Computing), а также архитектурных платформ с явной поддержкой параллелизма на уровне отдельных инструкций с широким командным словом – VLIW-архитектуры (Very Long Instruction Word).

EPIC- и VLIW-архитектуры, как правило, имеют длинный конвейер, из-за чего остро стоит проблема условных переходов, т.к. в момент исполнения операций условного перехода процессору неизвестно до конца, какая из веток условного перехода будет исполняться. В зависимости от успешности предсказаний условного перехода процессор либо исполняет ранее подкаченные инструкции из конвейера, либо освобождает конвейер и загружает нужные инструкции, что, очевидно, негативно сказывается на производительности исполнения программ. Для решения данной проблемы немалый вклад вносит оптимизирующий компилятор той или иной платформы, который осуществляет различные оптимизации на промежуточном представлении программ в процессе их компиляции.

Процесс оптимизации промежуточного представления компилятора разделен на фазы оптимизации, где каждая фаза реализует отдельную чаще независимую оптимизацию над IR [16]. Все фазы оптимизации формируют

линейку оптимизаций, которая в зависимости от реализации компилятора может меняться по контексту IR либо по требованию пользователя. Особенность реализации компиляторов позволяет получать IR практически после каждой фазы оптимизации, а также соответствующие абстрактные представления этого IR, такие как CFG.

Таким образом, есть возможность использовать и адаптировать некоторые оптимизации компилятора с целью получения наиболее удобного и эффективного представления программ, написанных на языке высокого уровня, что позволит решить задачу линеаризации CFG для повышения эффективности и расширения применимости исследуемого метода статического анализа многопоточных алгоритмов на предмет состояния гонки.

1.3. Средства статического анализа

На сегодняшний день для применения в промышленной разработке программного обеспечения существуют как коммерческие, так и бесплатные статические анализаторы. Фундаментальные подходы в реализации статических анализаторов многопоточных алгоритмов могут быть классифицированы следующим образом [13, 14]:

1. Идентификация ресурсов.

В анализе требуется определить, какая память разделяется между потоками. В общем случае задачу решить нельзя, потому что адресация памяти может происходить неявно. В случае создания тредов в Unix-системах необходимо добавлять в анализ все глобальные переменные, т.к. автоматически начинают разделяться после создания. Помимо прочего переменные могут передаваться в качестве параметра в поток, причем по указателю или по ссылке, что создает угрозу появления состояний гонок, поэтому такие переменные должны быть также добавлены в анализ.

2. Идентификация потоков.

Анализ требует идентификации конкурирующих потоков. Такая задача сводится к определению момента создания потоков для каждого языка высокого уровня. Методы создания могут быть как явными – системный вызов (fork() в Linux), так и неявными – сигналы и сообщения, при этом создание потока определяется языком программирования, на котором написан анализируемый код программы. Идентифицировать создание потоков можно на стадии компиляции программы.

3. Использование псевдонимов.

Статический анализ программ, которые используют указатели, будет практически невозможен без анализа псевдонимов [19]. Статические анализаторы стали пригодны на практике только с появлением методов по контролю над псевдонимами: внешнепроцедурный анализ Дойча [18] и весьма эффективный чувствительный к потокам анализ Стринсгарда [28]. Наличие точного и быстрого анализа псевдонимов является обязательным элементом статических анализаторов. В большинстве инструментов производится поверхностный анализ, в том числе с использованием нескольких эвристик, и часто пропускаются псевдонимы [19]. Надо отметить, что в статических анализаторах это делается для того, чтобы уменьшить число ложных срабатываний. В результате появления процессоров с технологией HyperThreading анализаторы, использующие поверхностный анализ, стали демонстрировать совсем плохие результаты. Без учета псевдонимов могут быть пропущены заведомо присутствующие состояния гонки – операции с тем же участком памяти, но названным по-другому или переданным по адресу.

4. Вычисления при помощи набора состояний.

На самом деле, статический анализ построен на исполнении программы, но под исполнением здесь понимают немного другое: выполнение программы на некоторой абстрактной машине, с набором абстрактных нестандартных состояний, подменяющих обычные. Основное понятие – состояние, что есть

набор переменных программы с ассоциированными с ними значениями. В случае с внутри-процедурным анализом в состояние входят локальные переменные, а внешнепроцурный анализ вносит в состояние еще и глобальные переменные, стек. Целью статических анализаторов является связь множества всевозможных состояний со всевозможными моментами исполнения программ. Обычно число таких состояний бесконечно или, по крайней мере, очень велико, поэтому статические анализаторы должны сводить состояния к упрощенному описанию, показывая отношения только между основными переменными, но в статических анализаторах это сводится всего лишь к отслеживанию только возможных значений.

5. Анализ участков кода.

Некоторые методы статического анализа (проверки на основе моделей или формальное математическое доказательство корректности) трудно применить в больших программах. Тогда статический анализ может быть применен к части программы, однако, такой метод не гарантирует корректного результата. Несмотря на то, что статический анализ может быть применен к части программы (к отдельному файлу и/или процедуре), такой подход дает плохие результаты – много ложных срабатываний [19]. Но есть и положительные моменты: действительно, часто можно не проверять полностью код на недостающие функции и процедуры.

Статические анализаторы разделяются в свою очередь по типу анализа. Всего разделяют 3 типа анализа:

1. Чувствительные к путям.

Средства, использующие анализ, чувствительный к путям, анализируют только действующие ветви исполнения программ. Они учитывают переменные и условные конструкции, исключая из обработки недосягаемые ветви исполнения.

2. Нечувствительные к путям.

Средства анализа, не чувствительные к путям, не производят отсев и рассматривают все варианты исполнения. Этот подход дает более точные результаты, но работает дольше по времени.

3. Чувствительные к контексту.

а) внешнепроцедурный анализ.

Средства, отслеживающие значения глобальных переменных и параметры функций. В общем случае внешнепроцедурный анализ работает довольно быстро, однако не очень точен. На данный момент не известно чувствительного к контексту средства, позволяющего находить все дефекты кода без ложных срабатываний.

б) внутрипроцедурный анализ.

Средства, анализирующие каждую функцию отдельно, вне контекста. Целью большинства разработок статических анализаторов является создание средства с минимальным количеством ложных срабатываний, но на практике анализаторы часто выдают результат, что дефект «возможен».

Одним из первых анализаторов был Lint. Он появился в составе операционной системы Unix 7 и использовался в качестве основного средства контроля качества кода. Со времен средства Lint появилось множество других средств проверки корректности исходного кода. Важно рассмотреть обзор наиболее известных средств статического анализа [13]:

1. Slint.

Slint – средство статической проверки программ на языке C. Представляет собой более современный вариант анализатора lint. Особенность этого средства состоит в том, что оно дает возможность добавить в код специальные аннотации, позволяющие «усилить» проверки качества кода.

2. Cppcheck.

Статический анализатор для проверки кода на языках C и C++. Использует множество проверок, которые не делаются компилятором. В своей основе

использует набор эвристик. Может служить в качестве дополнения к проверкам компилятора и компоновщика. Обнаруживает ограниченное число ошибок и обладает небольшой точностью.

3. Clang.

Clang – компилятор, позволяющий осуществлять доступ сторонним приложениям к результатам компиляции. Работает с языками C, C++, Objective-C, Objective-C++. Помимо компилятора содержит встроенный синтаксический анализатор. Аналогично средству cppcheck может служить дополнительным средством проверки написанного кода вслед за компилятором и компоновщиком.

4. CHESS.

Chess – средство анализа, использующее вместе проверку на основе моделей и динамический анализ. Выявляет ошибки распараллеливания путем систематического анализа планирования и чередования потоков. Способно обнаруживать такие проблемы, как состояние гонки, взаимоблокировки, зависания, активные блокировками и т.д. Проблемы в исходном коде после обнаружения легко исправить, так как этот анализатор дает полностью воспроизводимые результаты. Как и в случае любой другой проверки на основе моделей, систематические исследования обеспечивают полный охват кода тестами – для этого требуется формализовать искомые проблемы в терминах модели, по которой происходит верификация. Помимо проверки на основе моделей, анализатор CHESS работает и в динамическом режиме, т.е. выполняет тест циклически с помощью особого встроенного планировщика. В каждый проход теста выбирается новый вариант распределения процессорного времени между потоками. С другой стороны, это средство проверки на основе моделей, и оно управляет специализированным планировщиком, способным создавать специальное чередование потоков. Для уменьшения вариантов анализа в CHESS применяется частичная редукция и ограничение числа новых контекстов

итераций. Используя ограничение числа контекстов итераций, CHESS не уменьшает глубину расширения пространства состояний, а уменьшает количество переключений потоков в данном сеансе работы приложения.

5. PVS-Studio.

PVS-Studio представляет собой коммерческий продукт для выявления ошибок в статическом режиме в программах на языках C, C++ и C++0x. Это средство анализа разработано на основе библиотеки VivaCore с открытым исходным кодом. Данное средство анализа ошибок состоит из нескольких утилит, среди которых VivaMP отвечает за проверку ошибок при многопоточном исполнении кода. Само средство достаточно тяжеловесно и использует много вычислительных ресурсов.

6. KISS.

KISS – еще один инструмент для проверки исходного кода, исполняемого в нескольких потоках, на основе моделей. Программы должны быть написаны на языке C. Принцип работы анализатора KISS заключается в том, что он преобразует параллельное приложение на C в последовательное, при этом выбирая способ чередования. После этого выполняется собственно анализ с применением последовательного средства проверки на основе моделей. Далее анализом занимается другая утилита – SLAM.

7. SLAM.

SLAM – средство анализа анализирует последовательные программы на основе моделей. Ошибки, выявленные SLAM в последовательной программе, транслируются в ошибки для исходной программы [33]. В худшем случае сложность SLAM оценивается как $O(P \cdot (G \cdot L)^3)$, где P – размер программы, G – число глобальных состояний в конечном автомате, L – максимальное число локальных состояний функций. KISS не дает ложных срабатываний.

Статические анализаторы применяются для верификации определенных свойств в программах и проверки соответствия определённым критериям, основываясь только на исходном коде программы. На практике статические анализаторы дают неточный ответ о наличии того или иного свойства в программе, что является проблемой подобных средств анализа программ. В контексте обнаружения состояний гонок в программе статические анализаторы возвращают ответы следующего формата:

a) если состояние гонки есть, то ответ: гонка должна произойти.
b) если состояния гонки нет, то ответ: гонки не должно быть.

Основной принцип статического анализа: либо точность, либо время [19]. Отсюда возникает тенденция ложных срабатываний статического анализатора при попытке ускорить анализ, что может привести к бесполезности результата анализа. Таким образом, есть два типа ложного ответа статического анализатора:

a) false positive: гонки нет, а ответ – гонка должна быть.
b) false negative: гонка есть, а ответ – гонки не должно быть.

Выбор оптимального сочетания времени работы анализатора и точности является практической задачей, которая разрешается в каждом случае отдельно.

1.4. Метод статического анализа на основе графа совместного исполнения потоков

При моделировании параллельной программы используется предпосылка – чередование (мультипрограммность) [15], что позволяет упростить анализ реального параллельного вычисления, выполняемого микропроцессорами, и при этом в полной мере представить контекст программы. Чередование позволяет представить выполнение параллельной программы как последовательность дискретных шагов. Для того, чтобы в таком представлении были учтены всевозможные сценарии исполнения программы, и при этом требования к

полноте представления контекста программы не нарушались, необходимо, чтобы инструкции были максимально атомарные [15].

В работе [13-15] была предложена математическая модель обнаружения состояния гонки в многопоточных алгоритмах посредством построения модели взаимного исполнения атомарных инструкций двумя потоками на разделяемой памяти. Общая идея метода сводится к следующим действиям:

1. Для каждого потока выделяются операции, связанные с разделяемыми переменными.
2. Строится граф совместного исполнения потоков на основании операций с разделяемыми переменными в каждом из потоков.
3. Находятся классы эквивалентности полных путей на графе совместного исполнения потоков.
4. Анализируются полные пути на графе совместного исполнения потоков, которые принадлежат разным классам эквивалентности. На основе анализа выбираются пути, для которых возможно состояние гонки.
5. Выделенные пути анализируются, например, с помощью метода неопределенных коэффициентов.
6. По состоянию в финальной вершине делается вывод о наличии гонок.

Граф совместного исполнения потоков – ориентированный граф, представляющий всевозможные варианты совместного исполнения потоков в многопоточном алгоритме, где каждая дуга ассоциирована с атомарной операцией одного из потока, а вершины – с множеством состояний общей памяти после выполнения очередной атомарной операции. В случае двух потоков исполнения граф можно описать следующим образом:

$$G \coloneqq (V, A):\ V = \bigcup_{\substack{i=1,k+1 \\ j=1,n+1}} v_j^i\ ,\quad A = \bigcup_{\substack{i=1,k \\ j=1,n+1}} (v_j^i, v_j^{i+1}) \cup \bigcup_{\substack{i=1,k+1 \\ j=1,n}} (v_j^i, v_{j+1}^i)\ , \qquad (1)$$

где k, n – количество операций первого и второго потока соответственно, i, j – номер атомарной операции первого и второго потока соответственно, V –

множество вершин графа, A – множество дуг графа. Пример подобного графа представлен на рис. 2.

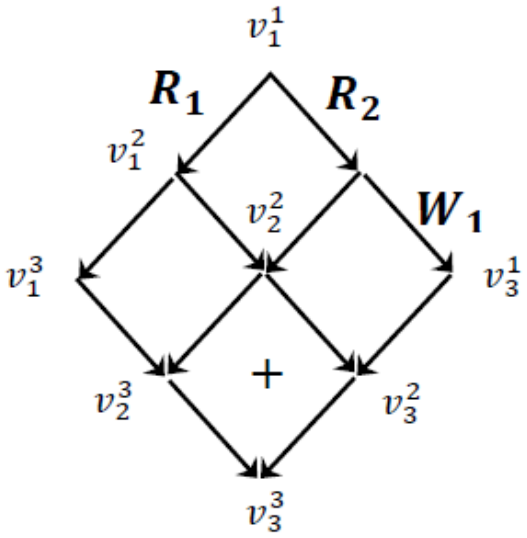


Рис. 2. Граф совместного исполнения двух потоков на разделяемой памяти.

Путям графа G соответствуют всевозможные варианты исполнения многопоточного алгоритма. Вершинам графа V соответствует множество состояний общей памяти после выполнения очередной инструкции, дугам A – атомарные операции над разделяемыми переменными. Каждому ребру поставлена в соответствие операция (R – чтение, W – запись) и ячейка памяти, над которой производится операция.

Граф G имеет вид ромба, в котором начальная вершина находится сверху, а конечная – снизу, а все ребра имеют направление сверху вниз. Начальная вершина определяет вектор исходного состояния взаимодействующих потоков. Начальной вершине сопоставлены значения всех разделяемых переменных после инициализации. Каждое ребро в таком графе определяет единичную атомарную операцию. После совершения потоком очередной операции происходит переход системы из одного состояния в другое по одному из ребер. Конечная вершина характеризует финальное (результирующее) состояние исполнения потоков.

Класс эквивалентности – набор полных путей на графе G, для которых состояние ячейки памяти в конечном узле одинаково.

Для нахождения классов эквивалентности необходимо владеть понятием *некоммутирующих операций*. Если существует хотя бы одна общая ячейка памяти, такая что состояние исполнения потоков относительно её после выполнения двух операций может зависеть от их порядка, то будем называть эти операции некоммутирующими, в противном случае скажем, что они коммутируют между собой [14]. Алгоритм поиска классов эквивалентности базируется на понятии некоммутирующих операций. Пример некоммутирующих операций изображен на рис. 3 и обозначен знаком «+».

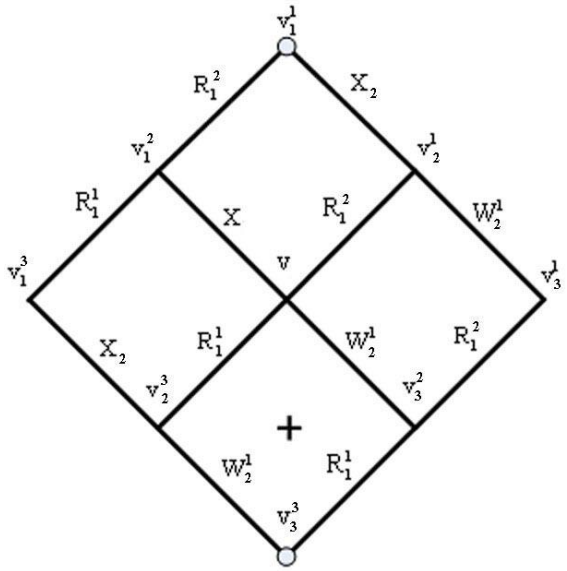


Рис. 3. Пример указания некоммутирующих операций на графе.

В работе [14] доказывается, что число классов эквивалентности полных путей может быть вычислено за $O(k \cdot n)$, где k и n – количество атомарных операций каждого из потоков. Данный факт является сильной стороной метода, указывающий на потенциальную высокую производительность анализа при реализации.

В работе [13] была улучшена математическая модель исследуемого анализа, что позволило расширить применимость метода. Улучшение было достигнуто за счет добавления нового абстрактного представления – расчетного графа,

который строится на основе графа совместного исполнения потоков. С помощью расчетного графа стало возможным учитывать значения всех разделяемых переменных, а не просто общую память.

Расчетный граф – конструктивная дискретная модель исходного кода программы, представляющая собой направленный граф, в котором каждому ребру соответствует атомарная операция, а вершинам ставится в соответствие множество значений всех разделяемых переменных.

Помимо добавления расчетного графа и анализа значений разделяемых переменных в работе [13] уделено внимание ветвлениям потока управления и циклам. Предложенное расширение позволяет дополнительно параметризировать некоторые ветвления и анализировать несложные циклы. На рис. 4 показан пример построения расчетного графа на примере алгоритма Петерсона для двух потоков [13].

Поток 1:	Поток 2:
A1: $y = 1$	B1: $x = 1$
A2: $z = 0$	B2: $z = 1$
A3: while(x==1&&z==0);	B3: while(y==1&&z==1);
A4: // critical section	B4: // critical section
A5: $y = 0$	B5: $x = 0$

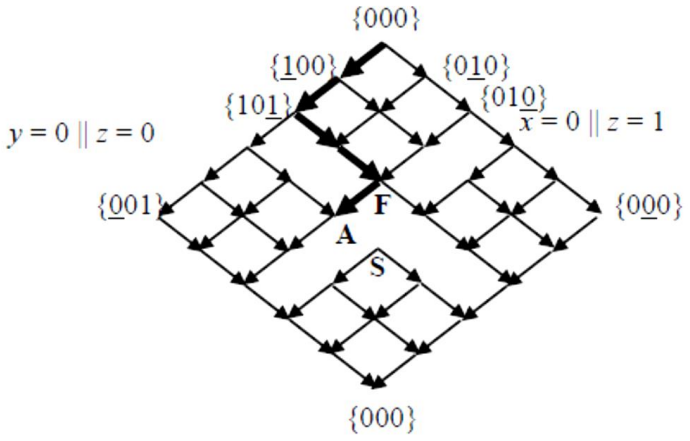


Рис. 4. Расчетный граф для алгоритма Петерсона.

Модель с расчетным графом является логичным продолжением развития математической модели анализа многопоточных алгоритмов, описанной в работе [14]. Поэтому модель с расчетным графом берется за основу в данной диссертационной работе с целью ее адаптации и реализации для промышленного использования.

В рамках данной работы принимается попытка адаптации математической модели при анализе контекста программы, полученного вследствие линеаризации графа потока управления. Поэтому интересно также привести описание гонки в терминах математической модели исследуемого метода статического анализа.

Гонка (в терминах графа G, описанный в (1)) – существование двух разных полных путей, для которых в финальном состоянии состояния хотя бы одной из ячеек памяти различны. С учётом начальных значений ячеек и того, как их значения меняются, используем метод неопределенных коэффициентов. На каждом ветвлении добавляем коэффициент $\alpha \in \{0, 1\}$ к изменению в левой ветви и (1-α) – правой. Множество неопределенных коэффициентов D полностью описывает путь на графе. В финальной вершине имеем множество значений, где каждая ячейка памяти в общем случае имеет вид [13]:

$$x_i = f_i(\vec{x}_0, D), \tag{2}$$

где $\vec{x}_0$ – состояние всех ячеек памяти в начальной вершине, D – значения неопределенных коэффициентов, f_i – некоторая функция, определенная для каждой из ячеек. Исходя из определения понятия гонки и принципов построения и анализа графа, изложенных выше, получаем, что гонка возможна тогда и только тогда, когда [13]:

$$\exists i, D_1, D_2 : f_i(\vec{x}_0, D_1) \neq f_i(\vec{x}_0, D_2), \tag{3}$$

где D_1 и D_2 – множества бинарных коэффициентов.

Глава 2. Реализации модели статического анализа нахождения состояния гонки в многопоточных алгоритмах с использованием линеаризованного графа потока управления

2.1. Анализ оптимизаций на промежуточном представлении

В работе был взят за основу компилятор CLANG&LLVM как наиболее активно развивающийся и обладающий рядом преимуществ с точки зрения гибкости реализации и удобства адаптирования под конкретные случаи и задачи. CLANG&LLVM для проведения оптимизаций использует платформонезависимый IR в SSA (Static Single Assignment) форме, причем CLANG выступает front-end'ом, транслируя язык высокого уровня в IR нужного вида (LLVM IR), а LLVM – оптимизирующим компилятором, который проводит необходимые оптимизации, трансформации или программный анализ на LLVM IR и затем транслирует финальный LLVM IR в бинарный код целевой архитектуры. Общая структура использования компилятора представлена на рис. 5.

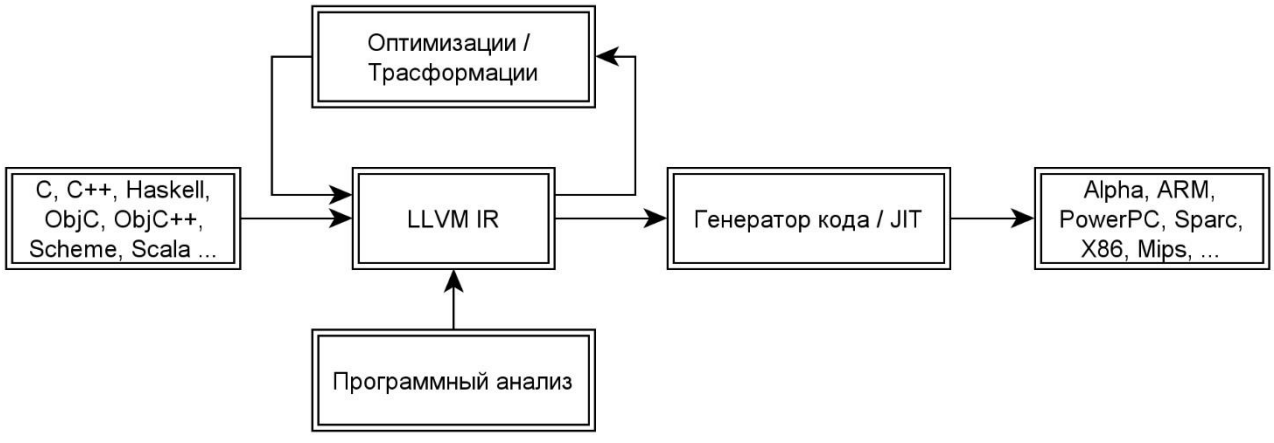


Рис. 5. Структура компиляции с помощью CLANG&LLVM.

Форма статического единственного присваивания (SSA) является одной из самых распространенных форм представления потока данных программы и

активно используется в большинстве современных оптимизирующих компиляторов [26]. В SSA-форме любая переменная может быть определена только один раз, вследствие чего для соблюдения данного ограничения должны выполняться следующие условия:

- Наличие φ-узлов в точках схождения потока управления. *φ-узел для переменной* – это операция, выбирающая среди множества значений переменной нужное.
- Переименование всех переменных так, чтобы каждому определению соответствовала своя уникальная переменная.

LLVM содержит более 100 уникальных оптимизаций, которые были проанализированы с точки зрения влияния на CFG. Предлагается рассмотреть и оценить влияние следующих оптимизаций, которые часто имеют свою реализацию в оптимизирующих компиляторах:

- If-conversions – перевод в предикатное представление [12].

  Современные EPIC-архитектуры поддерживают предикатное вычисление, что позволяет выполнять операции раньше перехода, ведущего на соответствующий участок. Операции, вынесенные выше перехода, исполняются под условием (предикатом) этого перехода. Используя поддержку предикатных вычислений, можно преобразовать ациклический регион программы, состоящий из нескольких линейных участков, в один линейный участок. При этом действии будут удалены все внутренние переходы ациклического региона. По сути, if-conversion преобразует зависимости по управлению с операциями переходов в зависимости по данным с предикатами. В рамках выбранного оптимизирующего компилятора CLANG&LLVM преобразование промежуточного представления в предикатную форму не реализовано, но предоставлен соответствующий API [22], который позволяет оперировать с нужными структурами IR и его представлениями. Таким образом, основываясь на

известных алгоритмах преобразования и общепринятых практиках [11, 26], можем сделать вывод, что возможна реализация if-conversion.

- Трансформация циклов:

  В современных оптимизирующих компиляторах реализованы десятки или даже сотни различных трансформаций циклов, некоторые из них можно отнести к оптимизациям, применение которых неявно приводит к линеаризации CFG. Эти и множество других трансформаций циклов подробно описаны в обзорной статье [17]. В рамках данной работы рассмотрены такие оптимизации CLANG&LLVM, применение которых с практической точки зрения наилучшим образом отвечают задаче линеаризации CFG. Согласно официальной документации оптимизирующего компилятора LLVM [24] возможно применить следующие оптимизации к циклам на IR, полученный в данном случае с помощью CLANG:

  - *-lcssa (Loop-Closed SSA Form Pass)*

    Оптимизация изменяет циклы путем переноса фи-узлов в конец циклов для всех значений, которые остаются живыми за границами цикла. Например:

```
for (...)                    for (...)
    if (c)                       if (c)
        X1 = ...                     X1 = ...
    else                 ->      else
        X2 = ...                     X2 = ...
    X3 = phi(X1, X2)             X3 = phi(X1, X2)
... = X3 + 4                  X4 = phi(X3)
                                 ... = X4 + 4
```

    Дополнительные фи-узлы являются излишними, но они легко удалятся после оптимизации InstCombine. Основная выгода данной трансформации – это сделать применение других цикловых оптимизаций, таких как LoopUnswitch, более простыми.

- *-licm (Loop Invariant Code Motion)*

  Оптимизация выполняет перемещение инвариантного кода в цикле, пытаясь удалить как можно больше кода из тела цикла.
- *-loop-deletion (Delete dead loops)*

  Оптимизация отвечает за удаление циклов, которые не вносят вклад в возвращаемое значение функции.
- *-loop-reduce (Loop Strength Reduction)*

  Оптимизация проводит понижение стоимости выражений с индуктивными переменными, заменяя дорогие операции на более дешевые. В частности, выражения с массивами изменяют так, чтобы была возможность воспользоваться преимуществами расширенного режима адресации индекса (scaled-index addressing modes), упрощающее обращение к элементам массивов.
- *-loop-rotate (Rotate Loops)*

  Оптимизация выполняет простое вращение цикла.
- *-loop-simplify (Canonicalize natural loops)*

  Оптимизация выполняет несколько трансформаций над естественными циклами, преобразуя их в более простую форму, которая позволяет делать последующий анализ и трансформации проще и эффективнее, например, для LICM (Loop Invariant Code Motion). Данная оптимизация создает дополнительный узел (предцикл) перед головой цикла, что гарантирует наличие только единственной входной дуги в цикл. Также гарантируется, что после применения оптимизации измененный цикл будет иметь только одну обратную дугу.
- *-loop-unroll (Unroll loops)*

  Оптимизация реализует простую раскрутку цикла, которая заключается в создании несколько копий итераций цикла (число копий n называет фактором развертки) и в увеличении шага цикла в это же число раз. Например, развертка с фактором 2:

```
for (i = 0; i < n; i++)          for (i = 0; i < n-1; i+=2) {
```

```
a[i] = b[i] + c[i];              a[i] = b[i] + c[i];
                          ->     a[i+1] = b[i+1] + c[i+1];
                               }
                               for (; i < n; i++)
                                   a[i] = b[i] + c[i];
```

Оптимизацию стоит применять после исполнения оптимизации *-indvars* (Canonicalize Induction Variables), когда циклы приведены к каноническому виду, что позволяет легко определять число итераций циклов.

- *-loop-unswitch (Unswitch loops)*

  Оптимизация изменяет циклы, которые содержат переходы на инвариантные для цикла условные операции для того, чтобы создать несколько циклов. Например:

```
for (...)                        if (lic)
    A                                for (...)
    if (lic)       ->                    A; B; C
        B                        else
    C                                for (...)
                                         A; C
```

  Ожидается, что оптимизация LICM исполняется раньше, что делает возможность применение данной оптимизации очевиднее.

- *-loop-idiom (Recognize loop idioms)*

  Оптимизация реализует распознаватель идиом, который трансформирует простые циклы в бесцикловую форму.

- *-loop-instsimplify (Simplify instructions in loops)*

  Оптимизация проводит легкое упрощение инструкций в теле цикла.

Среди представленных оптимизаций LLVM можно выбрать те, которые могут привести к существенному изменению CFG, а также прочих представлений IR, что в конечном итоге повышает линеаризацию CFG программ и упрощает различные виды статического анализа кода программ на предмет

состояния гонок. Стоит заметить, что для успешной линеаризации CFG программ важно не только применить необходимые оптимизации компилятора LLVM, но и правильно подобрать последовательность запусков оптимизаций, так как оптимизации могут быть конфликтующими – применимость одной приводит к неприменимости другой, и наоборот, связанными – неприменимость одной влечёт неприменимость другой [10].

Таким образом, наибольший интерес с точки зрения влияния на CFG вызывают следующие оптимизации:

- *-lcssa,*
- *-loop-simplify,*
- *-licm,*
- *-loop-reduce,*
- *-loop-unroll,*
- *-loop-unswitch.*

2.2. Линеаризация графа потока управления через линейку оптимизаций LLVM

Компилятор позволяет создавать свою собственную линейку оптимизаций либо использовать стандартную. Стандартных линеек оптимизаций у LLVM три: O1, O2, O3. Их также называют уровнями оптимизаций, где O1 – минимальный набор оптимизаций, а O3 – максимальный. Согласно документации LLVM интересуемый набор оптимизаций, выявленный в п.2.1, содержится во втором уровне оптимизации LLVM – O2. Таким образом, чтобы получить минимальный эффект от данных оптимизаций во время компиляции для оценки достаточно использовать второй уровень оптимизации O2.

В качестве примера интересно рассмотреть CFG-графы взаимоисключающего алгоритма Петерсона для 2 потоков, одна из реализации которого на языке Си выглядит следующим образом:

```
/* Algorithm of Peterson */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int volatile turn = 0;
int volatile flag0 = 0;
int volatile flag1 = 0;

static void *thread1_func ( void * args_ptr )
{
    turn = 1;
    flag0 = 1;
    while ( turn == 1 && flag1 == 1 );
    /**** Critical section ****/
    flag0 = 0;
    return NULL;
}

static void *thread2_func ( void * args_ptr )
{
    turn = 0;
    flag1 = 1;
    while ( turn == 0 && flag0 == 1 );
    /**** Critical section ****/
    flag1 = 0;
    return NULL;
}

int main () {
    pthread_t thread1,thread2;

    if (pthread_create(&thread1, NULL, thread1_func, NULL) != 0)
    {
        return EXIT_FAILURE;
    }

    if (pthread_create(&thread2, NULL, thread2_func, NULL) != 0)
    {
        return EXIT_FAILURE;
    }

    if (pthread_join(thread1, NULL) != 0)
    {
        return EXIT_FAILURE;
    }

    if (pthread_join(thread2, NULL) != 0)
    {
        return EXIT_FAILURE;
    }

    return 0;
}
```

Важным нюансом в представленной реализации алгоритма Петерсона является использование примитива «volatile». При применении большого количества оптимизаций компилятор, как правило, не знает о разделяемых переменных в коде, что может привести к неправильной генерации кода целевой архитектуры. Например, разделяемые переменные могут и вовсе быть удалены из процедур, исполняемых потоками. Таким образом, разработчику необходимо следить за расстановкой барьеров записи в память в случае активного использования оптимизаций компиляторов.

Поскольку программный код обоих тредов симметричен, то рассмотрим CFG одного из них на промежуточном представлении LLVM до оптимизаций и после применения интересуемых нас линейки оптимизаций. На рис. 6 и рис. 7 изображен CFG «thread1_func» до и после применения оптимизаций соответственно. Функционал LLVM позволяет получать промежуточное представление с различных фаз линейки компилятора в DOT-представлении, которые легко визуализируется различными утилитами для построения графов. Таким образом, в процессе применения метода статического анализа кода на предмет состояния гонок в многопоточном алгоритме дополнительно появляется удобное, быстрое и ясное средство визуализации, что повышает глубину анализа и упрощает применимость метода.

На рис. 6 видно, что на оптимизированном LLVM IR уменьшилось не только количество инструкций, но и ветвлений потока управления. Эффект от применения оптимизаций на реальных задачах будет больше, что сделает применение статического анализа более доступным [5].

Как видно на рис. 6, появились новые аналитические структуры – φ-узлы, которые ранее с точки зрения математической модели не анализировались. Докажем в следующем параграфе, что φ-узел в LLVM IR не влияет на анализ кода и на выводы о наличии состояний гонок в программном коде многопоточного алгоритма.

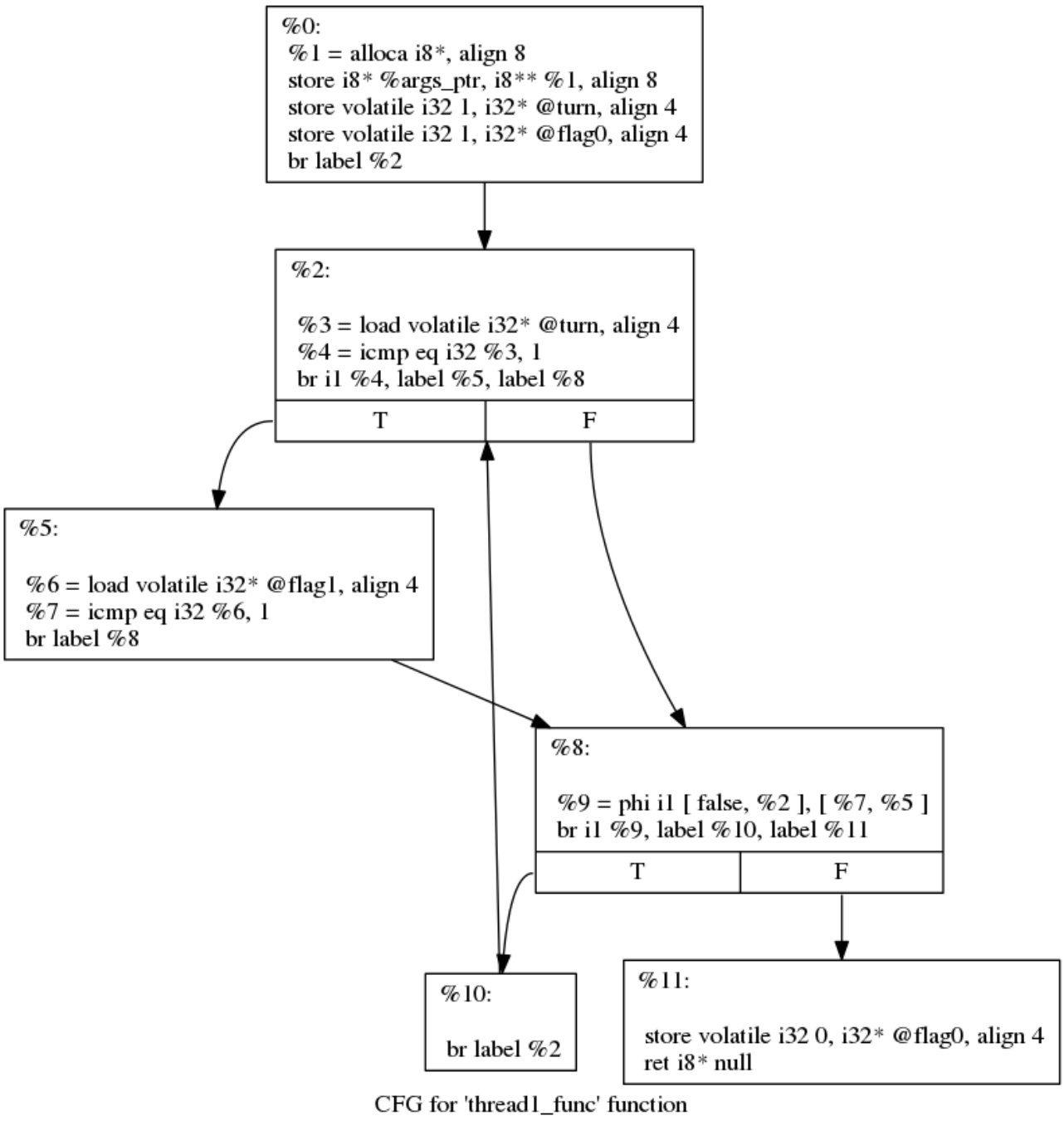


Рис. 6. CFG треда в алгоритме Петерсона до применения оптимизаций.

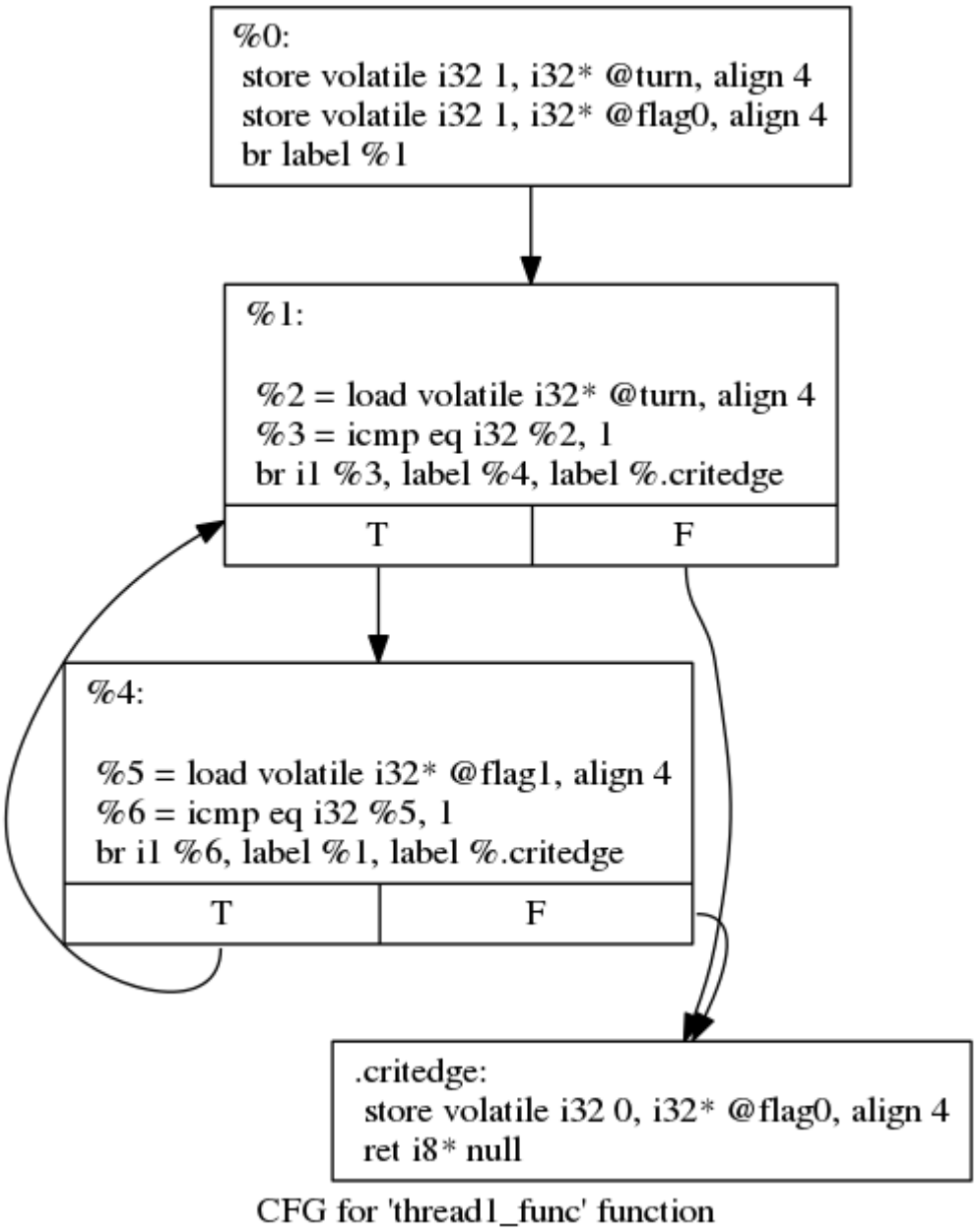


Рис. 7. CFG треда в алгоритме Петерсона после применения оптимизаций.

2.3. Анализ инструкций SSA-формы LLVM IR, не влияющих на математическую модель поиска RC

Исследуемый метод статического анализа многопоточных алгоритмов основывается на графе совместного исполнения потоков, узлы которого – атомарные операции. Данная особенность метода позволяет иначе взглянуть на

промежуточное представление программы. Можно смело утверждать, что некоторые инструкции IR программы могут быть удалены из анализируемого контекста программы в силу их не влияния на процесс анализа в рамках исследуемого метода. Поставим перед собой задачу выявления тех инструкций IR, которые не влияют на метод статического анализа, а также задачу доказательства не влияния обнаруженных инструкций [1, 4].

На промежуточном представлении атомарными операциями являются инструкции LLVM IR в SSA-форме. В свою очередь из определения о RC следует, что конкурентный доступ к памяти осуществляется через атомарные операции чтения/записи, которые в LLVM IR представляют собой инструкции load и store. Рассмотрим семантику данных инструкций [23]:

```
<result> = load [volatile] <ty>* <pointer>[, align <alignment>][,
!nontemporal !<index>][, !invariant.load !<index>]
<result> = load atomic [volatile] <ty>* <pointer> [singlethread]
<ordering>, align <alignment>
!<index> = !{ i32 1 }
```

```
store [volatile] <ty> <value>, <ty>* <pointer>[, align
<alignment>][, !nontemporal !<index>]
store atomic [volatile] <ty> <value>, <ty>* <pointer> [singlethread]
<ordering>, align <alignment>
```

Как видно из семантики инструкций load и store операция чтения/записи всегда происходит через указатель на память, а результат сохраняется в уникальную переменную согласно SSA-формы. Таким образом, данные инструкции оперируют и с регистрами, и с памятью. Отсюда **можно сделать вывод, что инструкции, оперирующие только с регистрами (виртуальными регистрами или временными переменными в SSA-форме), не представляют интереса в методе статического анализа, т.к. не влияют на возникновения RC**.

Анализируя многопоточные алгоритмы, нельзя не упомянуть класс инструкций, которые атомарно делают несколько операций, в том числе и

операции с памятью. Такие инструкции поддерживаются как на уровне микропроцессоров, так и на уровне ядра операционной системы. Поэтому в зависимости от вида операционной системы и архитектуры микропроцессора многообразие специальных инструкций для работы с памятью может отличаться, но, как правило, наблюдается общая концепция – выполнение нескольких операций за единый неделимый период времени: чтение из памяти, сравнение и изменение состояния памяти в зависимости от результатов сравнения. Ярким примером такой инструкции является CAS (Compare and Swap/Set), которая применяется в реализации неблокирующих алгоритмов. В LLVM IR инструкциями, проводящие атомарно несколько операций над памятью, являются следующие [23]:

1. «CMPXCHG» – читает из памяти значение и сравнивает его с переданным аргументом. Если значения равны, то в память сохраняется новое значение, переданное очередным аргументом.

   ```
   cmpxchg [weak] [volatile] <ty>* <pointer>, <ty> <cmp>, <ty> <new> [singlethread] <success ordering> <failure ordering> ; yields { ty, i1 }
   ```

2. «ATOMICRMW» – осуществляет различные операции над значением в памяти, где первый аргумент – тип операции (сложение, вычитание, сравнение и т.д.), второй аргумент – адрес в памяти, третий аргумент – значение-аргумент к осуществляемой операции.

   ```
   atomicrmw [volatile] <operation> <ty>* <pointer>, <ty> <value> [singlethread] <ordering> ; yields ty
   ```

Таким образом, заключаем, что класс атомарных операций типа CAS нельзя удалять из контекста программы, так как они непосредственно могут создать RC в параллельной программе.

SSA форма промежуточного представления обуславливает наличие φ-узлов. Докажем, что φ-узел не влияет на исследуемый метод статического анализа [4, 8]. В LLVM IR φ-узел называется φ-инструкцией. Особенность SSA-формы LLVM IR [6] включает в себя то, что все φ-инструкции в узлах графа потока

управления располагаются в начале, и до них не может быть никаких других инструкций. При этом φ-инструкция имеет следующую форму:
phi тип, [значение_1, label метка_1], ..., [значение_N, label метка_N], где значения – это временные переменные SSA-формы, хранящие результат исполнения каких-либо инструкций, а метки – узлы CFG, с которых пришло данное значение. Результат исполнения инструкции в SSA располагается на новую уникальную переменную – виртуальный регистр. Таким образом, входные параметры φ-инструкции всегда являются регистрами, т.е. из памяти не читает. Из определения φ-узла и того факта, что φ-инструкции располагаются в начале узла графа потока управления, следует, что результат сохраняется на виртуальном регистре. **Отсюда заключаем, что φ-инструкция оперирует только с регистрами и не влияет на исследуемый метод статического анализа многопоточных алгоритмов.**

Отдельно стоит уделить внимание инструкциям, которые передают управление в потоке исполнения программы, так называемые инструкции-перехода или branch-инструкции. В LLVM IR каждый линейный участок, что есть CFG-узел в графе потока управления, заканчивается соответствующей branch-инструкцией. В LLVM IR существуют следующие branch-инструкции [23]:

- *ret* – инструкция, которая используется для возраста управления из функции в точку вызова функции.

```
ret <type> <value>  ; возврат значения из не void функции
ret void            ; возврат из void функции
```

- *br* – инструкция, которая используется для передачи управления в другой линейный участок в текущей функции.

```
br i1 <cond>, label <iftrue>, label <iffalse>
br label <dest>      ; безусловный переход
```

- *switch* – инструкция, которая используется для передачи управления в один из нескольких линейных участков в зависимости от условия.

```
switch <intty> <value>, label <defaultdest> [ <intty> <val>,
```

```
label <dest> ... ]
```

- *indirectbr* – инструкция, которая реализует непрямой переход к метке внутри текущей функции, причем указан адрес метки, а не имя метки.

```
indirectbr <somety>* <address>, [ label <dest1>, label <dest2>,
... ]
```

- *invoke* – инструкция, которая передает управление указанной функции

```
<result> = invoke [cconv] [ret attrs] <ptr to function ty>
<function ptr val>(<function args>) [fn attrs]
```

Поскольку происходит, как правило, попроцедурный анализ кода в процессе поиска RC, то invoke или ret инструкции не представляют интереса для метода статического анализа, т.к. не приводят к ветвлению управления в CFG процедуры. Также семантика обозначенных выше инструкций показывает, что все инструкции оперируют с метками и адресами на код программы, адреса которых расположены на виртуальных регистрах/переменных SSA-формы. **Отсюда заключаем, что класс branch-инструкций не влияет на создание неразрешимого состояния гонки, т.к. не оперирует с памятью**. Однако удаление branch-инструкций из анализируемого контекста программы дает основание трансформировать ветвление управления, созданное данными branch-инструкциями, в CFG программы. Здесь возникает отдельная проблема корректного изменения ветвления управления в CFG программы. Данная проблема будет рассмотрена ниже.

Таким образом, анализ показывает, что существует возможность разделять инструкции LLVM IR на те, которые влияют на статический анализ, и те, которые могут быть опущены. Принцип отбора инструкции LLVM IR в зависимости от контекста программы до применения статического анализа позволяет значительно упростить представление программы и, как следствие, упростить граф потока управления.

2.4. Получения сокращенного оптимизированного промежуточного представления LLVM – Reduced Opt LLVM IR

В рамках данной работы был разработан программный комплекс по анализу LLVM IR и получения сокращенного промежуточного представления программы. В качестве технического средства для проведения анализа было выбрано представление LLVM IR в виде CFG в DOT-формате. В данном формате генерируется CFG средствами оптимизирующего компилятора, что является удобным, т.к. такой формат легко визуализировать множеством утилит, которые представляют DOT-формат в графическом виде.

Разработанный программный комплекс для анализа LLVM IR – анализатор IR – состоит из различных этапов преобразования LLVM IR, как в оригинальном виде, так и в DOT-формате. Общая архитектура анализатора изображена на рис. 8. На вход анализатору поступает код программы, написанной на языке высокого уровня (ЯВУ), который средствами front-end компилятора CLANG транслируется в LLVM IR. В качестве front-end системы может выступить любой другой транслятор, который создает корректное плаформонезависимое промежуточное представление LLVM IR [8].

Согласно п. 2.1-2.2 выявленные оптимизации LLVM применяются с целью первичной линеаризации CFG и сокращения общего количества инструкций в программе. В конечном итоге получаем новый вид промежуточного представления LLVM IR – Opt LLVM IR. Далее средствами оптимизирующего компилятора LLVM получаем CFG программы, построенное на LLVM IR, в DOT-формате. Выбор формата представления CFG не является принципиальным и может быть использован любой другой пригодный.

Заключающим этапом в анализаторе является удаление классов инструкций, не влияющих на статический анализ. Вначале удаляются основные классы инструкций, выявленные в рамках данной работы, затем удаляются дополнительные классы инструкций, указанные пользователем.

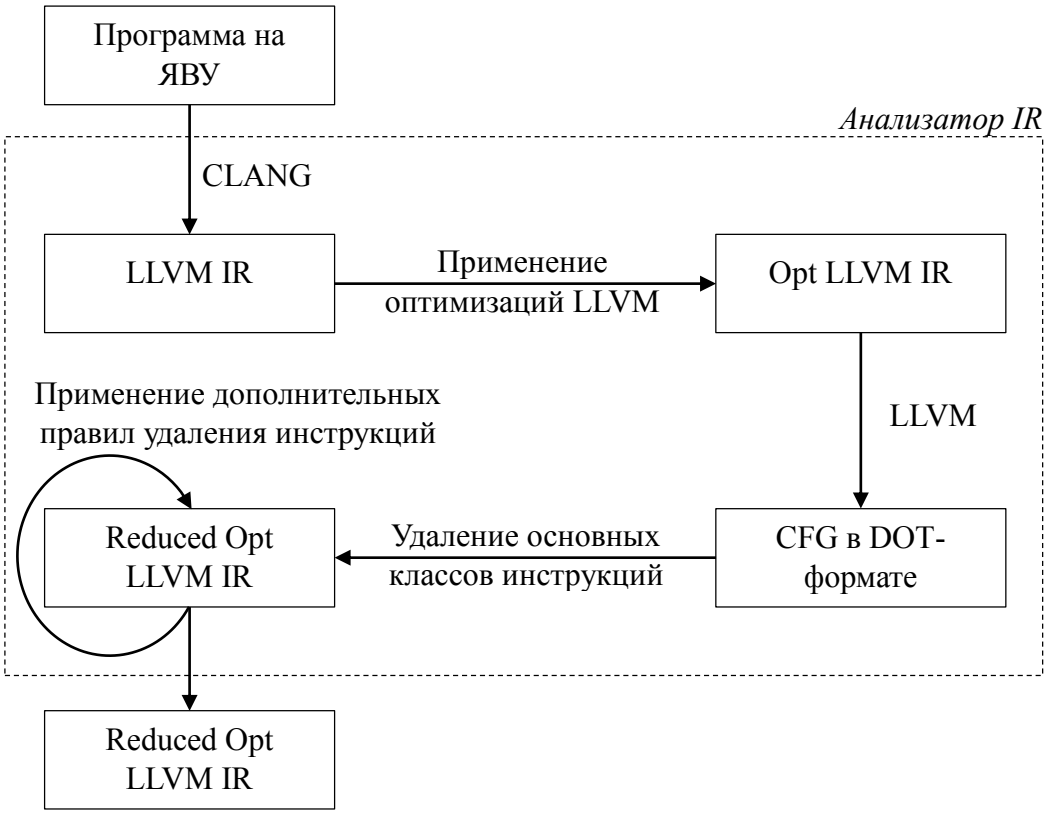


Рис. 8. Архитектура программного анализатора IR.

Дополнительные классы инструкций необходимы на тот случай, если основных классов инструкций недостаточно, и обнаружены специфические инструкции в LLVM IR, которые не влияют на статический анализ кода.

Согласно п.2.3 было показано, что на статический анализ не влияют инструкции, оперирующие только с виртуальными регистрами или переменными, в том числе branch-инструкции. Таким образом, на линейных участках (CFG-узлах) удаляются инструкции, которые не обращаются к памяти. После этого удаляются дополнительные инструкции из блока дополнительных правил удаления инструкций. В свою очередь к удалению могут быть отнесены инструкции работы с памятью, которая не разделяется между потоками [7].

Надо отметить, что данный этап при удалении branch-инструкций не влечет изменение в структуре линейных участков CFG, т.е. линейные участки не объединяются и не удаляются. Таким образом, структура ветвления управления в CFG сохраняется для последующего этапа анализа графа потока управления.

Этап сокращения промежуточного представления очень эффективен в промышленных комплексах программ, где логика каждого потока более нагружена локальным функционалом, не связанным с разделяемыми переменными. Поэтому для наглядности рассмотрим пример получения Reduced Opt IR в алгоритме Петерсона без этапа применения оптимизаций LLVM. Представим, что промежуточное представление первого потока (см. рис. 6) сразу попал на этап получения Reduced Opt IR. Рассмотрим CFG первого потока согласно рис. 6:

```
digraph "CFG for 'thread1_func' function" {
    label="CFG for 'thread1_func' function";

    Node0x1d27ff0 [shape=record,label="{%0:\l  %1 = alloca i8*,
                  align 8\l  store i8* %args_ptr, i8** %1, align
                  8\l  store volatile i32 1, i32* @turn, align 4\l
                  store volatile i32 1, i32* @flag0, align 4\l  br
                  label %2\l}"];
    Node0x1d27ff0 -> Node0x1d28280;
    Node0x1d28280 [shape=record,label="{%2:\l\l  %3 = load
                  volatile i32* @turn, align 4\l  %4 = icmp eq i32
                  %3, 1\l  br i1 %4, label %5, label
                  %8\l|{<s0>T|<s1>F}}"];
    Node0x1d28280:s0 -> Node0x1d284c0;
    Node0x1d28280:s1 -> Node0x1d28520;
    Node0x1d284c0 [shape=record,label="{%5:\l\l  %6 = load
                  volatile i32* @flag1, align 4\l  %7 = icmp eq
                  i32 %6, 1\l  br label %8\l}"];
    Node0x1d284c0 -> Node0x1d28520;
    Node0x1d28520 [shape=record,label="{%8:\l\l  %9 = phi i1 [
                  false, %2 ], [ %7, %5 ]\l  br i1 %9, label %10,
                  label %11\l|{<s0>T|<s1>F}}"];
    Node0x1d28520:s0 -> Node0x1d28910;
    Node0x1d28520:s1 -> Node0x1d28970;
```

```
    Node0x1d28910 [shape=record,label="{%10:\l\l  br label
                  %2\l}"];
    Node0x1d28910 -> Node0x1d28280;
    Node0x1d28970 [shape=record,label="{%11:\l\l  store volatile
                  i32 0, i32* @flag0, align 4\l  ret i8*
                  null\l}"];
}
```

В данном примере Анализатор IR удалит все инструкции кроме load и store, после чего Reduced Opt IR примет следующий вид:

```
digraph "CFG for 'thread1_func' function" {
    label="CFG for 'thread1_func' function";

    Node0x1d27ff0 [shape=record,label="{%0:\lstore i8* %args_ptr,
                  i8** %1, align 8\lstore volatile i32 1, i32*
                  @turn, align 4\lstore volatile i32 1, i32*
                  @flag0, align 4\l}"];
    Node0x1d27ff0 -> Node0x1d28280;
    Node0x1d28280 [shape=record,label="{%2:\l%3 = load volatile
                  i32* @turn, align 4\l|{<s0>T|<s1>F}}"];
    Node0x1d28280:s0 -> Node0x1d284c0;
    Node0x1d28280:s1 -> Node0x1d28520;
    Node0x1d284c0 [shape=record,label="{%5:\l%6 = load volatile
                  i32* @flag1, align 4\l}"];
    Node0x1d284c0 -> Node0x1d28520;
    Node0x1d28520 [shape=record,label="{%8:\l|{<s0>T|<s1>F}}"];
    Node0x1d28520:s0 -> Node0x1d28910;
    Node0x1d28520:s1 -> Node0x1d28970;
    Node0x1d28910 [shape=record,label="{%10:\l}"];
    Node0x1d28910 -> Node0x1d28280;
    Node0x1d28970 [shape=record,label="{%11:\lstore volatile i32
                  0, i32* @flag0, align 4\l}"];
}
```

В графическом представлении Reduced Opt IR для алгоритма Петерсона можно увидеть на рис. 9, который получается напрямую из полученного представления после Анализатора IR.

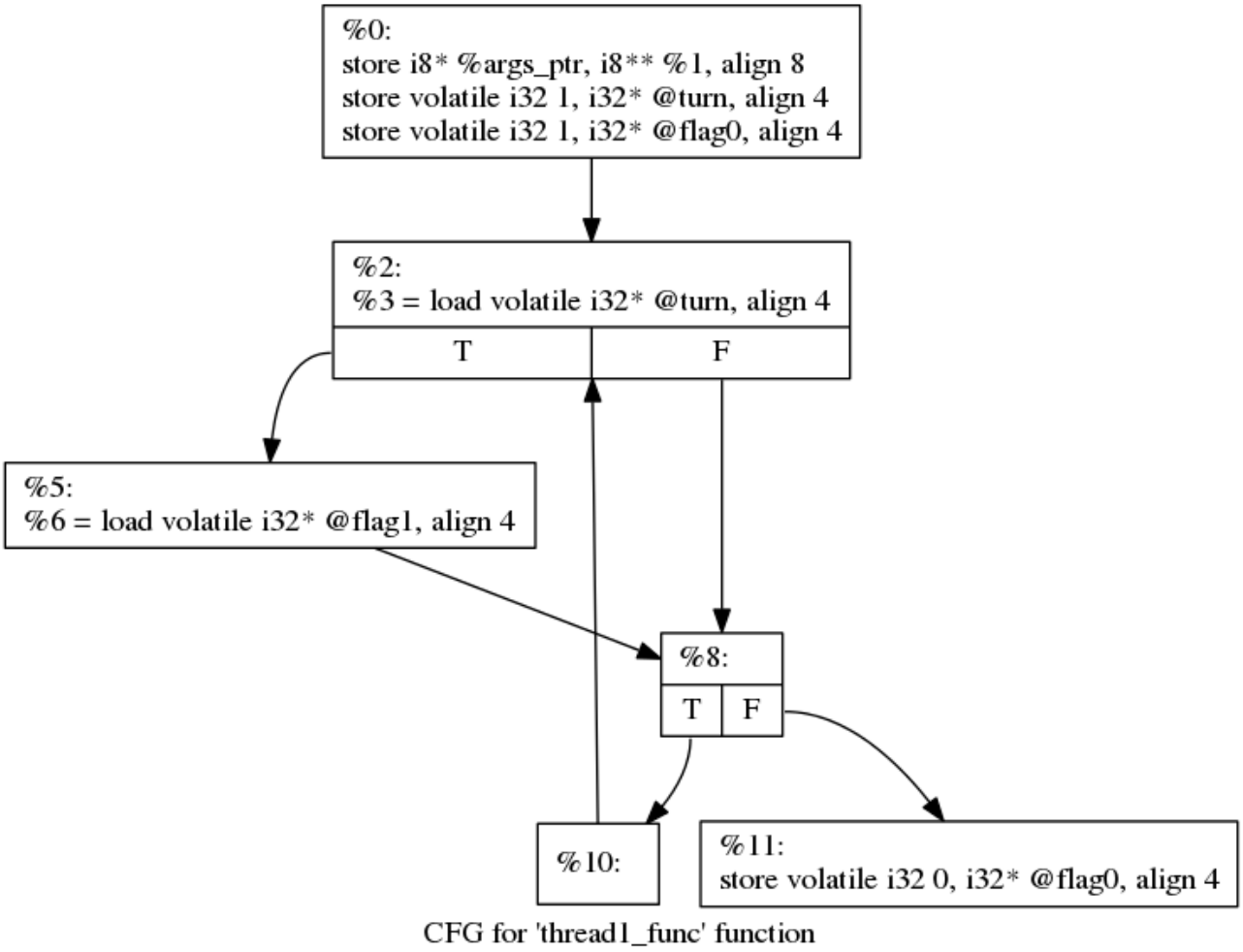


Рис. 9. Визуализация Reduced Opt IR для алгоритма Петерсона.

На рис. 9 видно, что контекст программы существенно упростился. Наблюдается не только сокращение количества инструкций, которые надо было бы анализировать в статическом анализе, но и появляются пустые CFG-узлы, которые дают основание для их удаления и, как следствие, проведения более глубокой линеаризации CFG программы. Процесс линеаризации CFG на подобном контексте программы с учетом соблюдения корректности для статического анализа будет рассмотрен ниже.

2.5. Контекстно-зависимая линеаризация графа потока управления на сокращенном промежуточном представлении программы

Принцип сокращения промежуточного представления программы посредством удаления инструкций, работающих только с регистрами (или виртуальными регистрами в SSA-форме), существенно влияет на представление CFG программы, которое в дальнейшем используется в статическом анализе кода. Инструкции перехода оперируют только с регистрами, поэтому возможна ситуация, когда некоторые узлы CFG остаются пустыми, т.к. в них изначально не было значимых для статического анализа инструкций, таких как load/store инструкции или атомарные инструкции типа CAS. Отсюда заключаем, что **узлы CFG, не содержащие значимых инструкций, являются также незначимые и могут быть удалены из CFG.** При удалении CFG-узла возникает нетривиальная задача – как поступить с входящими и выходящими дугами удаляемого узла. Возникают следующие нетривиальные случаи:

1. Выходящая дуга удаляемого CFG-узла является обратной дугой цикла.
2. Из удаляемого CFG-узла выходят несколько дуг в разные непустые CFG-узлы.

Теорема 1. Обратные дуги у пустого CFG могут быть удалены без влияния на математическую модель метода статического анализа.

Доказательство. В работе [13] была доказана теорема: «*для описания цикла в анализирующем графе достаточно одного повторения тела цикла*». В CFG тело цикла – это все CFG-узлы начиная от головы цикла и до CFG-узла, откуда выходит обратная дуга в голову цикла. Головой цикла называют CFG-узел, куда входит обратная дуга цикла. Отсюда заключаем, что обратную дугу можно удалять без ущерба статическому анализу, что и требовалось доказать.

Теорема 2. При удалении пустого CFG-узла с несколько выходящими дугами нельзя однозначно согласовать изменение потока управления без ущерба математической модели статического анализа.

Доказательство. Проставим каждому ветвлению управления в CFG неопределенный коэффициент β_i^j, где i – номер узла в CFG, j – порядковый номер выходящей дуги, причем $j \in [1, N-1]$, где N – число выходящих дуг. Если из CFG-узла выходит 2 дуги, то неопределённый коэффициент первой дуги – β_i^1, а второй – $(1-\beta_i^1)$. Если из CFG-узла выходит 3 дуги, то неопределённый коэффициент первой дуги – β_i^1, второй – β_i^2, третьей – $(1-\beta_i^1-\beta_i^2)$. Применяя данный принцип присваивания неопределённых коэффициентов к ветвлениям, получаем следующий вид состояния разделяемой ячейки памяти, который изначально был описан в формуле (2):

$$x_i = f_i(\vec{x}_0, D, B),$$

где B – множество коэффициентов β_i^j. К данному описанию ячейки памяти применима доказанная в работе [13] теорема: «*наличие гонки в предлагаемой модели с ветвлениями эквивалентно наличию гонки в задаче*». Отсюда получаем, что состояние гонки на CFG с неопределенными коэффициентами согласно формуле (3) описывается следующим образом:

$$\exists i, D_1, D_2, B: f_i(\vec{x}_0, D_1, B) \neq f_i(\vec{x}_0, D_2, B). \quad (4)$$

Таким образом, в случае наличия гонки в задаче должен существовать фиксированный набор неопределённых коэффициентов β_i^j, который может быть нарушен в случае удалении пустого CFG-узла с несколько выходящими дугами, т.к. любое изменение этих дуг (объединение, удаление и т.д.) влечет изменение соответствующих коэффициентов β_i^j. Отсюда делаем вывод, что удаление данного CFG-узла неоднозначно влияет на математическую модель исследуемого статического анализа, что и требовалось доказать.

На основе теоремы 1 и теоремы 2 был разработан и реализован алгоритм удаления пустого CFG-узла, который описывается следующим образом [3]:

Алгоритм 1. Удаление пустого CFG-узла.

1. Удаляются сначала пустые CFG-узлы, у которых одна выходящая дуга:
   а. Входящие дуги удаляемого CFG-узла переносятся в CFG-узел,

следующий за удаляемым CFG-узлом.

b. Выходящая дуга из удаляемого CFG-узла просто удаляется.

2. Повторяется п.1 до тех пор, пока представление CFG не перестанет меняться, т.е. все пустые CFG-узлы с одной выходящей дугой будут удалены.
3. Пустые узлы, у которых две и более исходящих дуги, остаются в CFG без удаления.

После применения алгоритма 1 мы получаем контекстно-зависимый линеаризованный граф потока управления – Reduced CFG, который является основном контекстном для анализа в исследуемом методе статического анализа. Прежде, чем проводить статический анализ исследуемым методом докажем, что анализируемый контекст является корректным для математической модели метода в следующей теореме.

Теорема 3. Наличие неразрешимого состояния гонки на редуцированном графе потока управления эквивалентно наличию гонки в задаче.

Доказательство. Из описания алгоритма 1 и доказательства теоремы 2 следует, что формулы (4) и (3) эквивалентно описывают состояние гонки в алгоритме. Отсюда следует доказательство теоремы.

Приведем пример контекстно-независимой линеаризации CFG. Рассмотрим Opt LLVM IR алгоритма Петерсона для 2 потоков, который был получен в п. 2.4. На рис. 9 можно увидеть два пустых CFG-узла: 8 и 10. CFG-узел 8 имеет две выходящие дуги, поэтому согласно алгоритму 1 данный узел не должен подвергаться удалению. CFG-узел имеет одну обратно выходящую дугу, поэтому согласно алгоритму 1 будет успешно удален, причем входная дуга будет просто удалена. Результат применения алгоритма представлен на рис. 10. Как видно из рис. 10, удалось избавиться от цикла и некоторых переходов в CFG представленной алгоритма. Эффект от применения контекстно зависимой линеаризации CFG на реальных задачах будет больше, что сделает применение статического анализа, описанного в работе [13-15], более доступным.

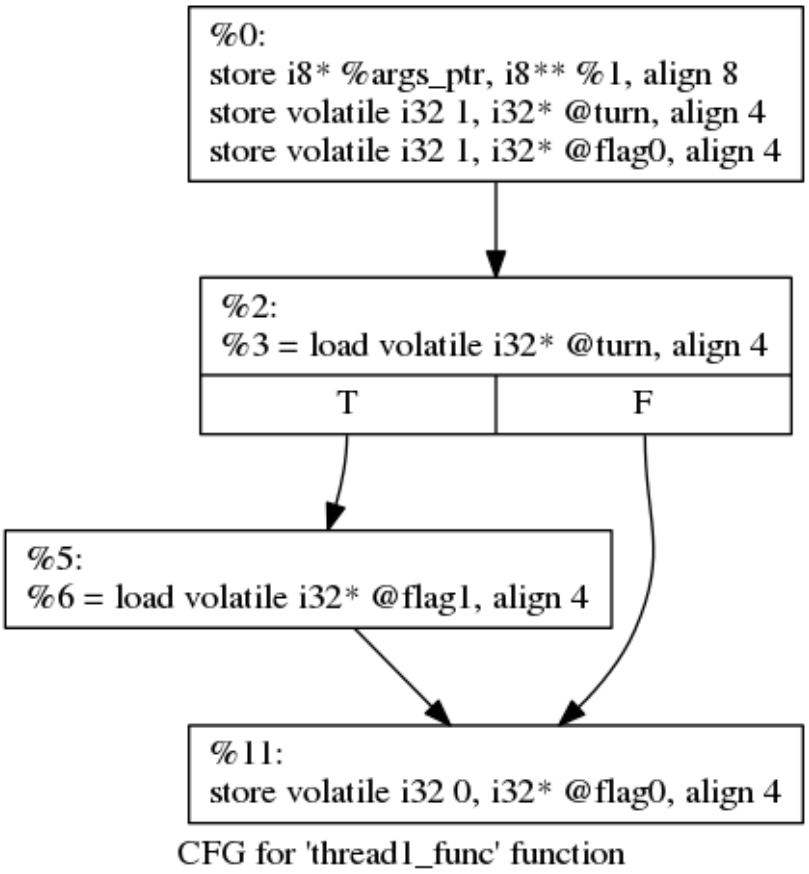


Рис. 10. CFG потока в алгоритме Петерсона после контекстно зависимой линеаризации.

На рис. 10 осталось ветвление управления, но оно значительно упрощено относительно начального представления. Подобному ветвлению можно снова применить принцип неопределенных коэффициентов для построения расчетного графа, что будет рассмотрено ниже в данной работе.

2.6. Поиск состояний гонок в исследуемом методе на Reduced CFG

Математическая модель исследуемого метода статического анализа согласно работам [13-15] предполагает построение графа потока управления, где единицей исследования является атомарная операция. В данной работе предлагается построение графа совместного потока на промежуточном представлении LLVM IR, которое имеет вид Reduced CFG. Причем ветвления управления в Reduced CFG не учитываются при построении графа потока управления. В данном случае используются только инструкции из Reduced CFG.

Рассмотрим процесс построения графа потока управления на примере алгоритма Петерсона, реализация которого представлена в п. 2.2. Согласно п.2.3. Анализатор IR создаст сокращенное промежуточное представление первого и второго потока, которое можно увидеть на рис. 11.

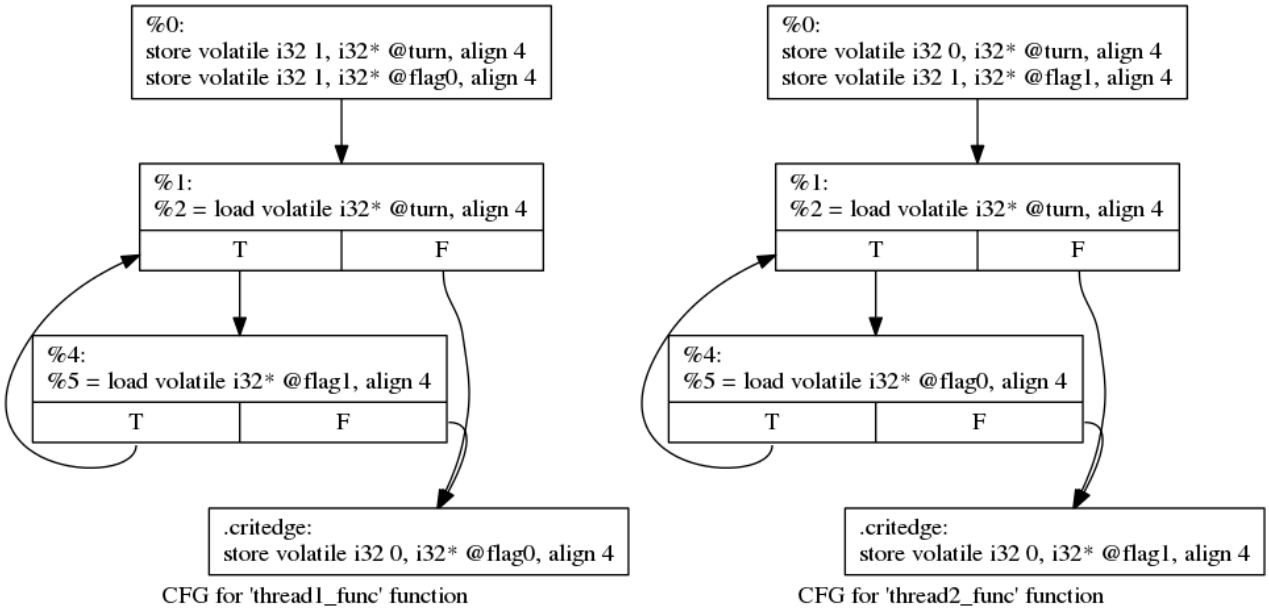


Рис. 11. Reduced Opt IR для обоих потоков в алгоритме Петерсона.

Далее контекстно зависимая линеаризация графа потока управления создаст Reduced CFG для каждого из потоков, которые можно увидеть на рис. 12.

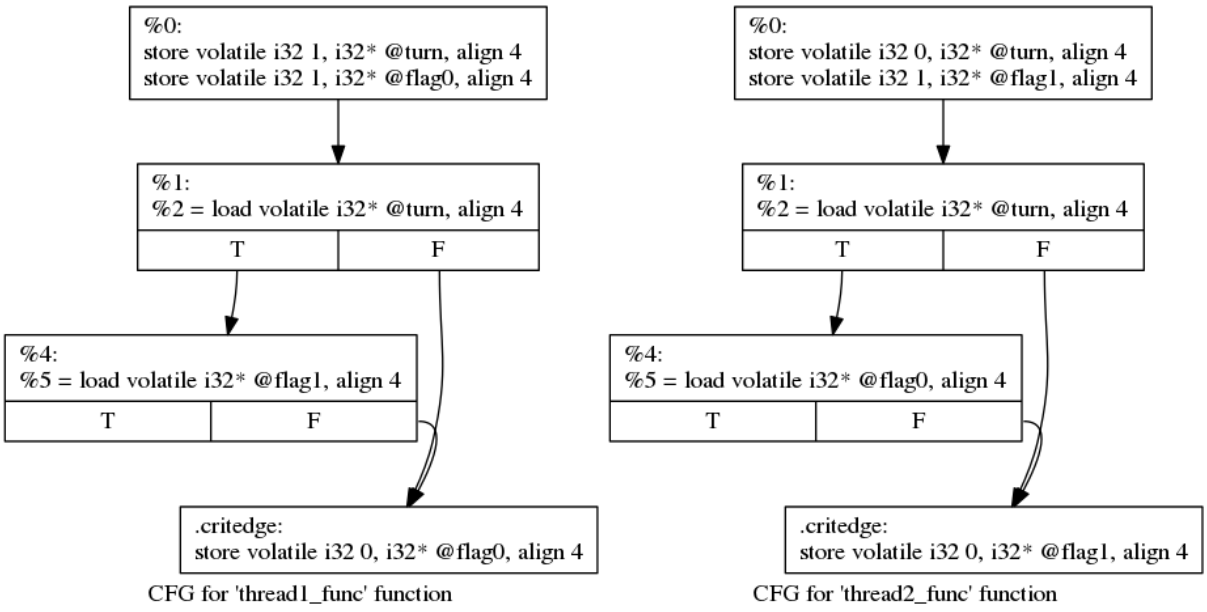


Рис. 12. Reduced CFG для обоих потоков в алгоритме Петерсона.

На рис. 12 видно, что в Reduced CFG убрана только обратная дуга в узле №4, но переход по условию остался. Поскольку пустых узлов нет, то линеаризация CFG оказалась минимальной. Построенный граф совместного исполнения потоков можно увидеть на рис. 13, где отображены знаком «+» некоммутирующие инструкции и точка нахождения обоих потоков в критической секции (точка «S»).

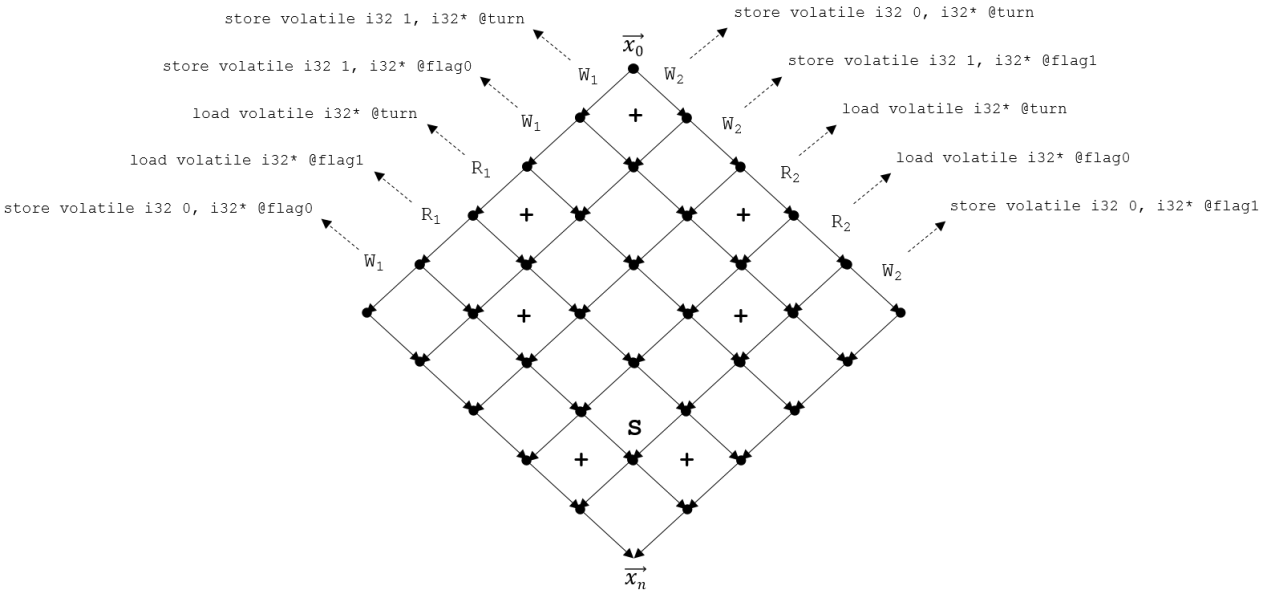


Рис. 13. Граф совместного исполнения потоков для алгоритма Петерсона.

Точка S на графе совместного исполнения потоков обозначает неразрешимое состояние гонки. Если будут обнаружены допустимые пути исполнения потоков, которые проходят через эту точку, то алгоритм не реализует критическую секцию.

Реализованный алгоритм поиска классов эквивалентности [14] показывает в данном примере 24 класса эквивалентности на Reduced CFG, которые приведены в приложении 1. В дальнейшем процесс поиска состояний гонок связан с анализом расчетного графа, который строится также на Reduced CFG, но учитывает простые ветвления управления. В CFG-узле, в котором более одной исходящей дуги, дугам расставляются неопределённые коэффициенты, также как это проделывается в доказательстве теоремы 2. При анализе путей в графе

совместного исполнения потоков дополнительно учитываются данные неопределённые коэффициенты.

При построении расчетного графа вводится дополнительная функция над вектором состояния системы, атрибуты которого являются значения разделяемых переменных – это функция условного перехода [13]:

$$M: \left(v_j^i, v_j^{i+1}\right) \to P(\vec{x}) = 0, \quad (5)$$

где P – функция-предикат, определенная на множестве значений вектора разделяемых переменных, определяет возможность передвижения системы из текущей вершины по рассматриваемой дуге в расчетном графе. Таким образом, в расчетном графе добавляются дуги-условия, которых в Reduced CFG нет.

Утверждение 1. В промежуточном представлении LLVM IR инструкциями, которые ассоциируются с дугами-условиями, являются инструкции, подготавливающие аргумент branch-инструкции.

Доказательство утверждения 1 следует из принципа построения CFG и определения branch-инструкций. Если имеется ветвление управления, то переход осуществляется по условию (branch по условию). В зависимости от условия branch-инструкция осуществляет передачу управления на нужную ветку потока управления.

Утверждение 2. Для ассоциации дуги-условия достаточно одной инструкции, вычисляющей аргумент для branch-инструкции по условию.

Доказательство утверждения 2 следует из особенностей промежуточного представления LLVM IR. Поскольку LLVM IR приближен к ассемблерному листингу, то и финальный аргумент передачи управления создается одной инструкцией. В LLVM IR такой инструкцией, например, является fCMP, iCMP.

Таким образом, дуга-условие в расчетном графе будет инструкция, являющаяся предком для соответствующей branch-инструкции. Анализатор IR при проведении анализа и трансформации сохраняет всех предков branch-инструкций. Список таких предков будет доступен в момент построения расчетного графа.

Построим расчетный граф для алгоритма Петерсона, который показан на рис. 14.

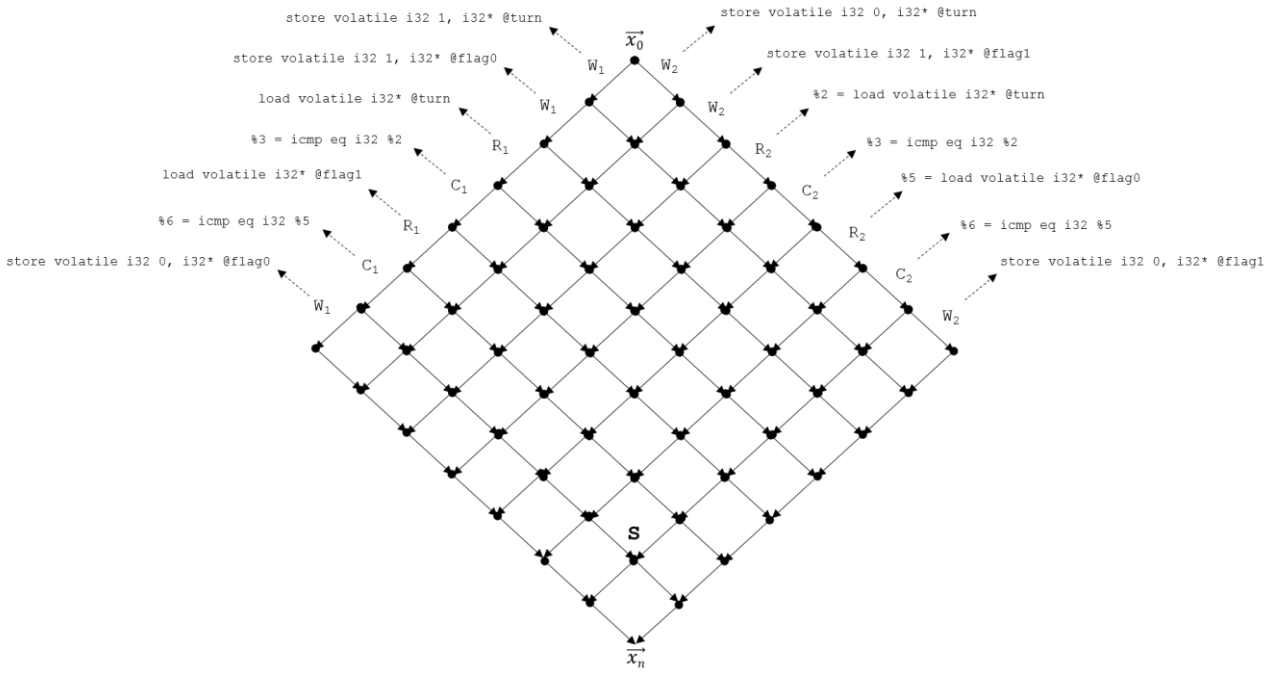


Рис. 14. Расчетный граф для двух потоков в алгоритме Петерсона.

В более ранних работах [13-15] были описаны алгоритмы анализа путей, взятые из классов эквивалентности. С учетом возможных значений разделяемых данных и дуг-условий выясняются допустимые пути на расчетном графе и, как следствие, на графе совместного исполнения потоков. Допустимые пути на расчетном графе обозначены синимы цветом на рис. 15.

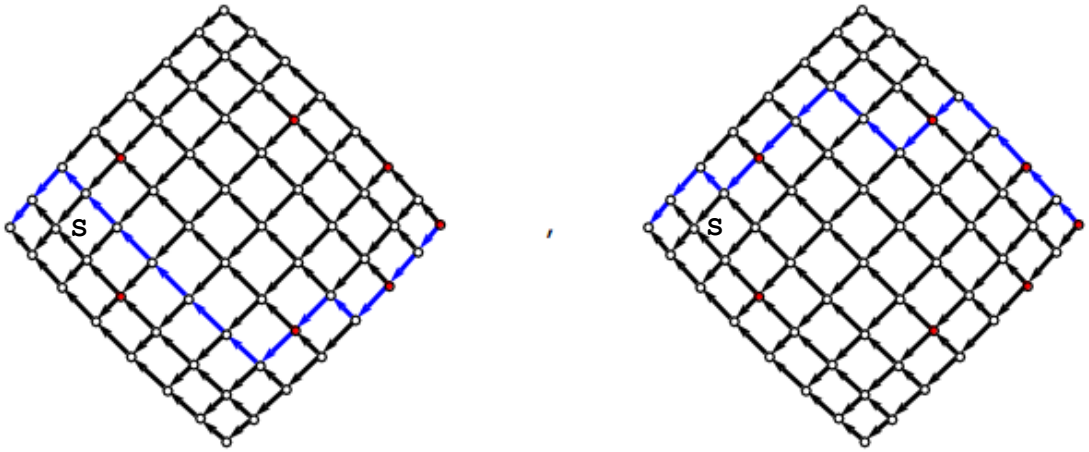


Рис. 15 Допустимые пути исполнения потоков на расчетном графе.

Как видно на рис. 15, допустимые пути не проходят через точку S, отсюда делаем вывод, что алгоритм Петерсона корректно разрешает критическую точку.

2.7. Модель автоматизация поиска состояний гонок

В рамках адаптированной математической модели метода статического анализа была предложена и реализована новая модель организации процесса анализа кода программы, который ориентирован на использование LLVM IR. Предложенная модель изображена на рис. 16.

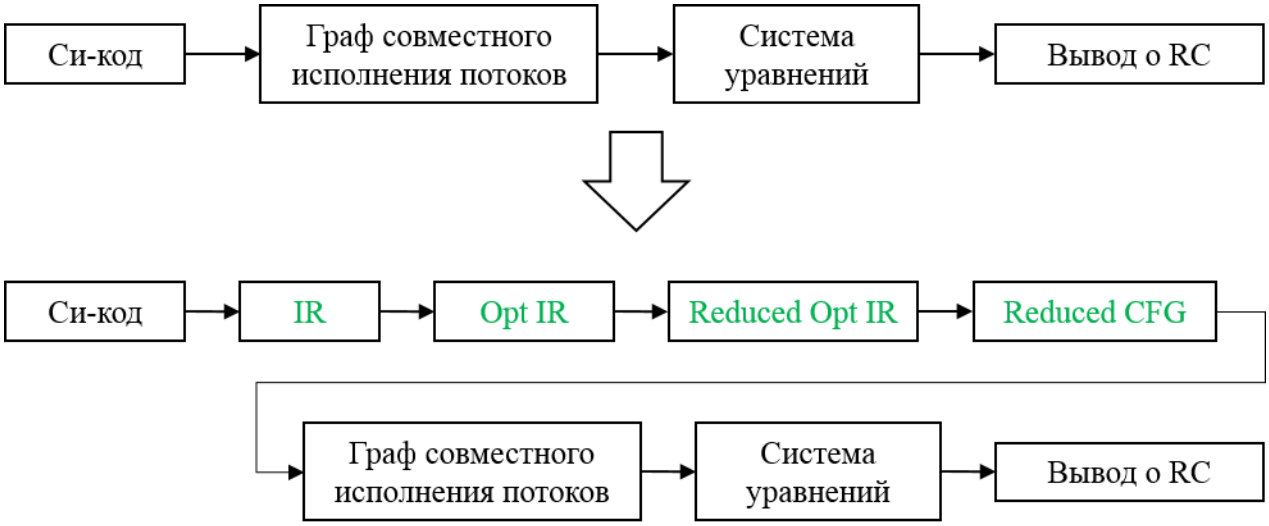


Рис. 16. Модель организации поиска состояний гонок.

Предложенная модель добавляет значительные этапы по подготовке анализируемого контекста. Анализируемый контекст существенно сокращается, но самой сильной стороной является возможность его применения к большому числу промышленных программ. Универсальность LLVM IR позволяет гибко применять метод статического анализа на различных платформах и архитектурах.

Архитектура реализованного комплекса программ включает как собственную реализацию утилит на скриптовых языках, так и использование сторонних программ. Предложенная модель позволяет с высокой степенью автоматизировать процесс анализа кода программы.

Разработанный комплекс программ по анализу включает в себя реализованный Анализатор IR, который получает на вход код программы, компилируемый компилятором CLANG, после чего анализатор создает контекст программы Reduced CFG, который подается на вход анализатору написанный в системе компьютерной алгебры Wolfram Mathematica. Анализатор в Wolfram Mathematica по алгоритмам, описанным ранее в работах, строит граф совместного исполнения потоков, выделяет классы эквивалентности, представители классов, затем расчетный граф и анализ путей на нем. На выходе из анализатора на Wolfram Mathematica происходит вывод результатов: отображаются возможные способы исполнения. Если есть пути, в которых все потоки одновременно проходят через критическую секцию, значит автоматически делается вывод о возможности неразрешимого состояния гонки. Архитектура системы поиска состояний гонок изображена на рис. 17.

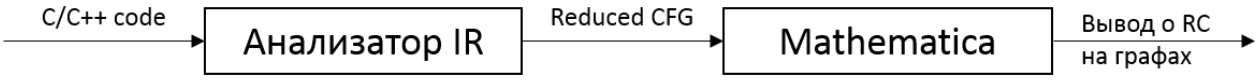


Рис. 17. Архитектура системы поиска состояний гонок.

Reduced CFG подается на вход анализатору Wolfram Mathematica в преобразованном виде. Анализатор транслирует Reduced CFG в листинг следующего вида:

```
Init
<тип операции>     <переменная>    <операция инициализации>
...
Thread
<тип операции>     <переменная>    <операция>     <внутренний тип>
...
Thread
<тип операции>     <переменная>    <операция>     <внутренний тип>
...
End
```

Листинг разделен на логические блоки: Init – начальная инициализация разделяемых переменных; Thread – блок описания одного из потока исполнения, блок заканчивается началом очередного блока Thread либо признаком окончания листинга End.

Элементы листинга означают следующее:

<тип операции> – операция чтения или записи в смысле метода статического анализа. Возможные обозначения: W и R.

<переменная> – переменная в Reduced CFG, над которой происходит операция чтения или записи.

<операция> – описание операции исходя из описания Reduced CFG на LLVM IR.

<внутренний тип> – дополнительный признак операции для анализатора Wolfram Mathematica. Возможные обозначения: U – запись, C – проверка условия, S – атомарная. Во внутреннем типе возможна конкатенация обозначений в зависимости от инструкций LLVM IR. Данный тип как раз описывает дуги-условии в расчетном графе и соответствующие инструкции в LLVM IR.

Таким образом, созданный комплекс программ позволяет с высокой степенью автоматизации наладить процесс верификации комплексов программ методом статического анализа на основе графа совместного исполнения потоков. Пример и разбор модельных задач рассмотрен в следующей главе.

Глава 3. Практические результаты и анализ применения программной реализации модели поиска состояний гонок в многопоточных алгоритмах

3.1. Спинлок (спин-блокировка)

Спин-блокировка – алгоритм, реализующий критическую секцию [29, 35]. В алгоритме используется разделяемая переменная, которая индицирует потокам о том, занята критическая секция или нет. В реализации алгоритма используются атомарные операции, такие как CAS.

Критерий корректности алгоритма – наличие в критической секции не более одного потока исполнения. На графе совместного исполнения потока и расчетном графе данный критерий означает, что не должно быть допустимых путей, проходящих через критическую секцию. На графах критическая секция обозначена точкой S.

Рассмотрим результаты работы Анализатора IR на Си-коде, одна из реализации которого приведена ниже. Как видно из кода реализации алгоритма, анализировать необходимо процедуру thread_func. Промежуточное представление LLVM IR в виде CFG данной процедуры изображено на рис. 18, что является начальным анализируемым контекстом в системе верификации кода алгоритма. Далее Анализатор IR преобразует первоначальный контекст кода алгоритма в оптимизированное промежуточное представление Opt IR, который приведен на рис. 19. На рис. 19 видно, что контекст существенно сократился.

Дальнейшим важным этапом в Анализаторе IR выступает шаг получения сокращенного промежуточного представления Reduced Opt IR. Стоит заметить, что в контексте присутствует атомарная инструкция cmpxchg, которая входит в особый класс анализируемых инструкций и удалена из контекста не будет. Результат работы анализа и конечный вид Reduced Opt IR приведен на рис. 20.

```
/* Spinlock */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define CAS(a,b,c) __sync_bool_compare_and_swap (a,b,c)
#define SPINLOCK_BUSY 1
#define SPINLOCK_FREE 0

int volatile lock = SPINLOCK_FREE;

static void *thread_func ( void * args_ptr )
{
    while (CAS(&lock,SPINLOCK_FREE,SPINLOCK_BUSY));
    /**** Critical section - Spinlock was captured ****/
    lock = SPINLOCK_FREE;
}

int main () {
    pthread_t thread1,thread2;

    if (pthread_create(&thread1, NULL, thread_func, NULL) != 0)
    {
        return EXIT_FAILURE;
    }

    if (pthread_create(&thread2, NULL, thread_func, NULL) != 0)
    {
        return EXIT_FAILURE;
    }

    if (pthread_join(thread1, NULL) != 0)
    {
        return EXIT_FAILURE;
    }

    if (pthread_join(thread2, NULL) != 0)
    {
        return EXIT_FAILURE;
    }

    return 0;
}
```

Reduced Opt IR показывает пустые CFG-узлы вследствие удаления классов инструкций, не влияющих на статический анализ. Получившийся контекст дает основание удалить CFG-узлы и дуги согласно Алгоритму 1 и доказанным теоремам 1-3. На рис. 20 также изображен Reduced CFG от Анализатора IR.

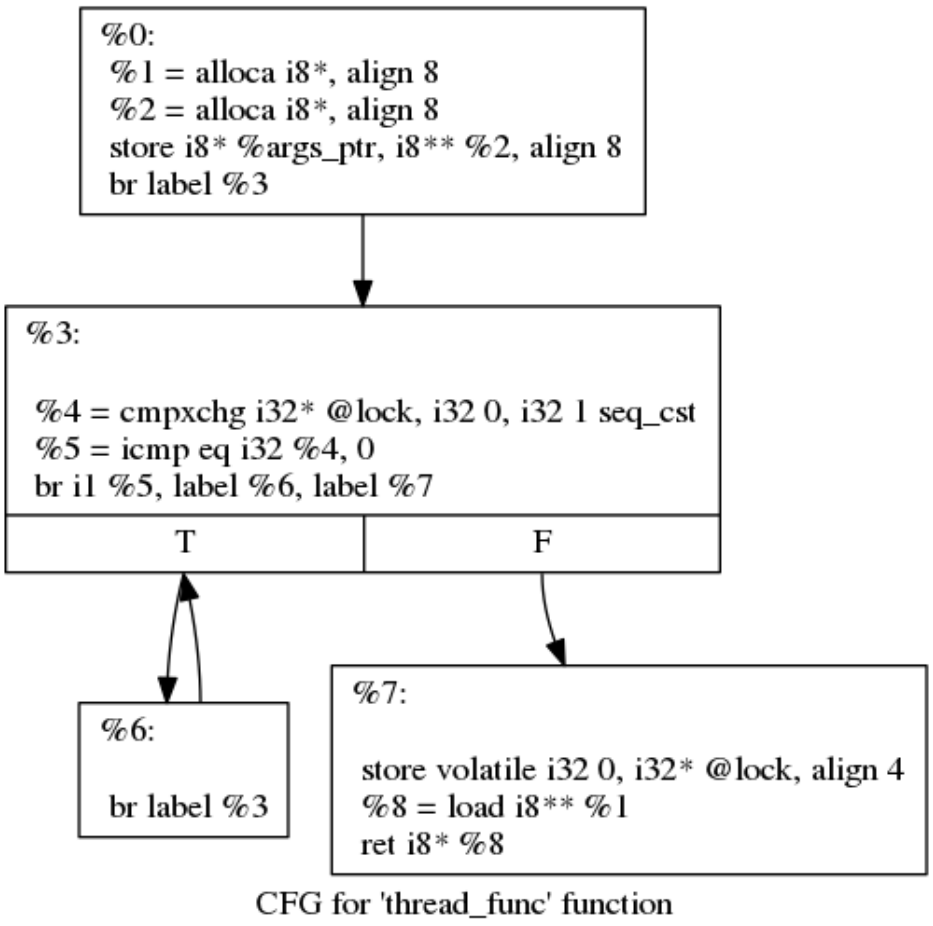


Рис. 18. Начальный CFG процедуры thread_func в алгоритме спин-блокировки.

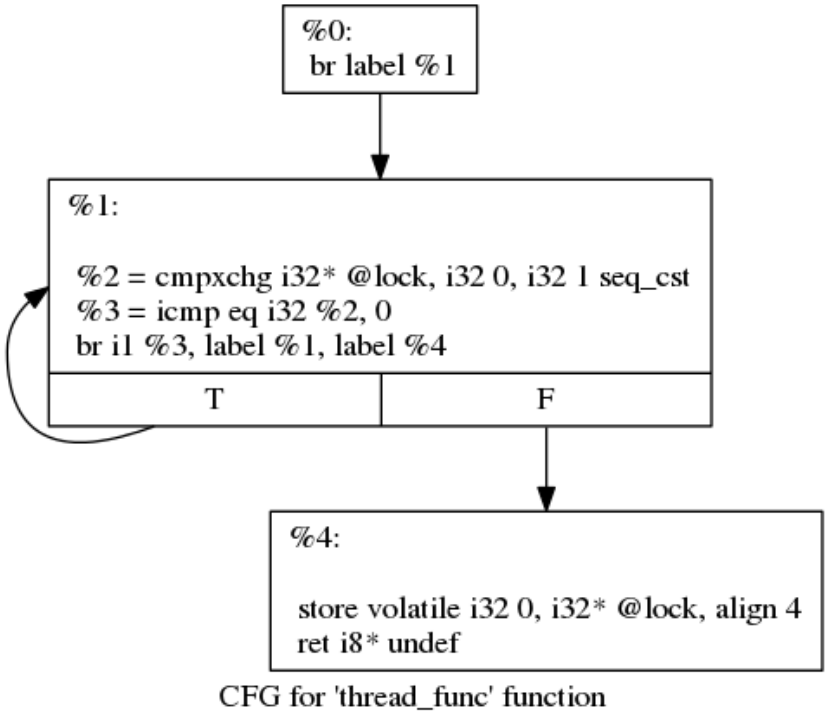


Рис. 19. Opt IR в виде CFG процедуры thread_func в алгоритме спин-блокировки.

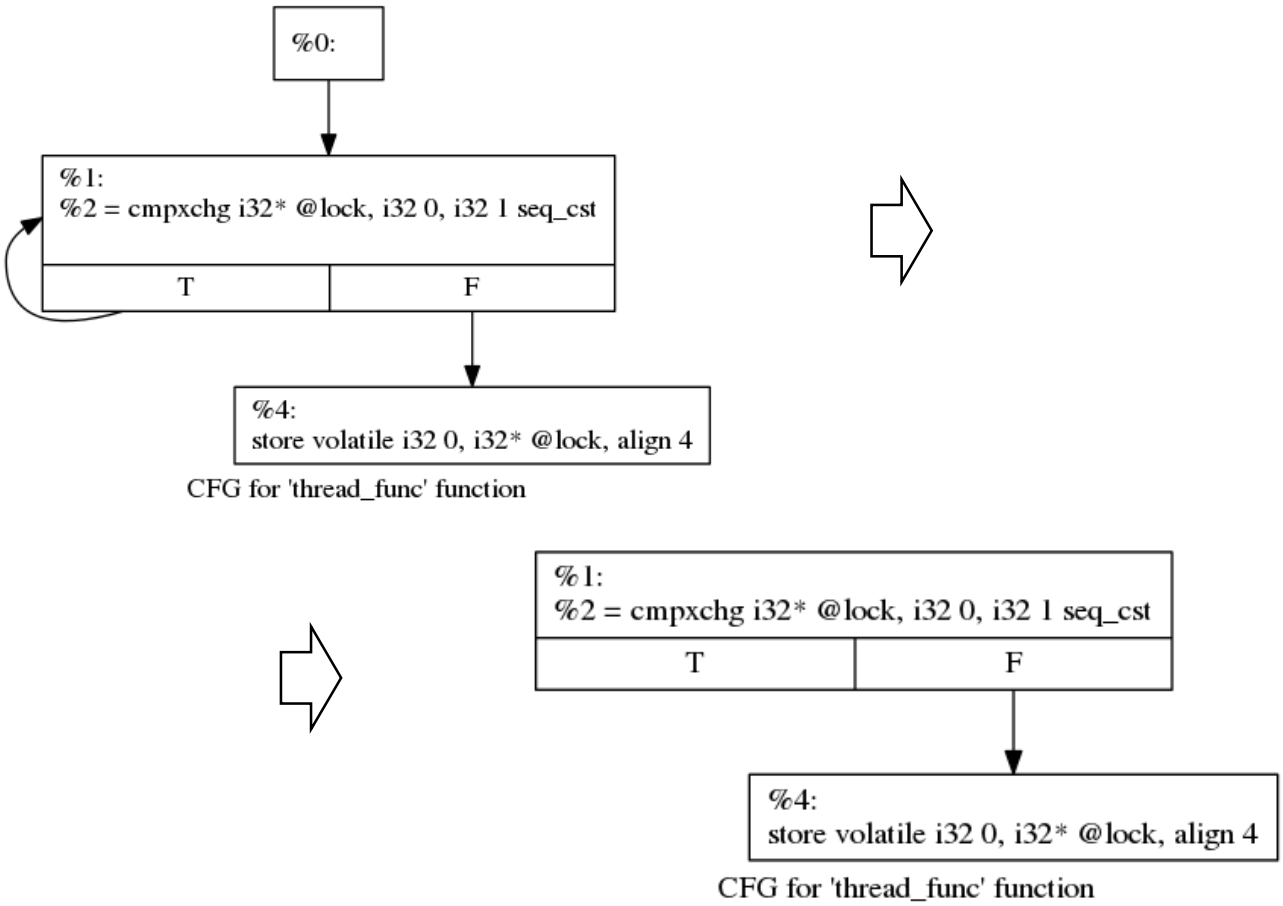


Рис. 20. Преобразование Reduced Opt IR в Reduced CFG процедуры thread_func в алгоритме спин-блокировки.

Финальным этапом оптимизации анализируемого контекста перед статическим анализом является трансляция Reduced CFG в нужного вида листинг, который подается анализатору на Wolfram Mathematica. В этом преобразовании инструкции LLVM IR отображаются в два класса типов операций (W/R), а также в классы внутреннего типа операций. Если инструкция находится из класса атомарных инструкций, то операция помечается как S-типа. Если инструкция принадлежит классу сравнения, операция помечается как C-типа. Если операция не входит ни в один из классов инструкций, то она помечается как U-типа. Таким образом, транслированный листинг Reduced CFG для алгоритма спин-блокировки выглядит следующим образом:

```
Init
W   lock  lock=0
Thread
R   lock   lock==0 CS
W   lock   lock=1  U
R   V2     V2==0   C
W   lock   lock=0  U
Thread
R   lock   lock==0 CS
W   lock   lock=1  U
R   V2     V2==0   C
W   lock   lock=0  U
End
```

Анализатор на Wolfram Mathematica обнаружил для данного листинга Reduced CFG 14 классов эквивалентности (см. Приложение). Анализ на расчетном графе выявил только 2 допустимых путей исполнения на графе совместного исполнения потоков, которые отображены на рис. 21.

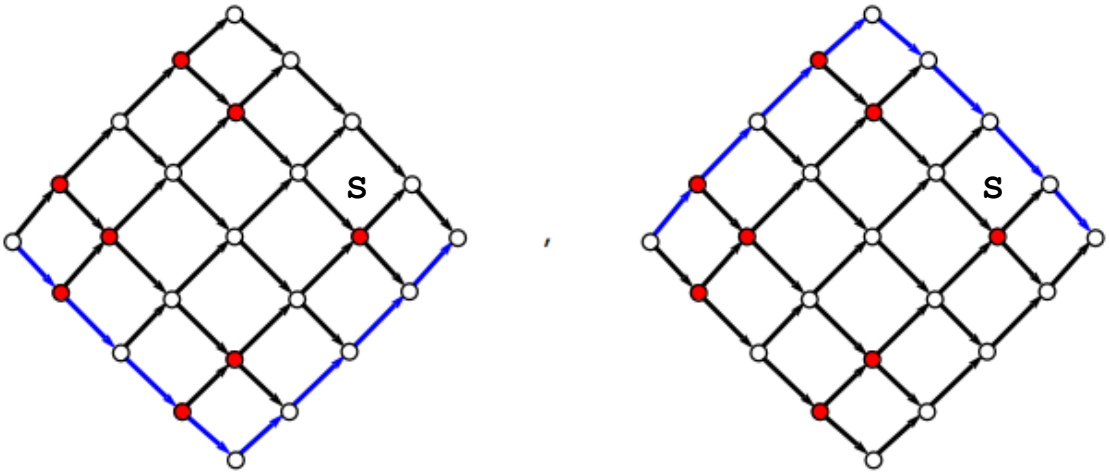


Рис. 21. Допустимые пути совместного исполнения потоков в алгоритме спин-блокировки.

Как видно на рис. 21, ни один из допустимых путей не проходит через точку S, что означает невозможность одновременного наличия потоков в критической секции. Отсюда вытекает вывод, что алгоритм успешно разрешает критическую секцию и является корректным.

3.2. Некорректный алгоритм спин-блокировки

В качестве примера рассмотрим некорректный алгоритм спин-блокировки, заменив атомарные инструкции CAS на аналогичные, но раздельные [37].

```
static void *thread_func ( void * args_ptr )
{
    while ( lock != SPINLOCK_FREE );
    lock = SPINLOCK_BUSY;
    /**** Critical section - Spinlock was captured ****/
    lock = SPINLOCK_FREE;
}
```

Из приведенного примера некорректного алгоритма спин-блокировки конечный контекст программы изображен на рис. 21.

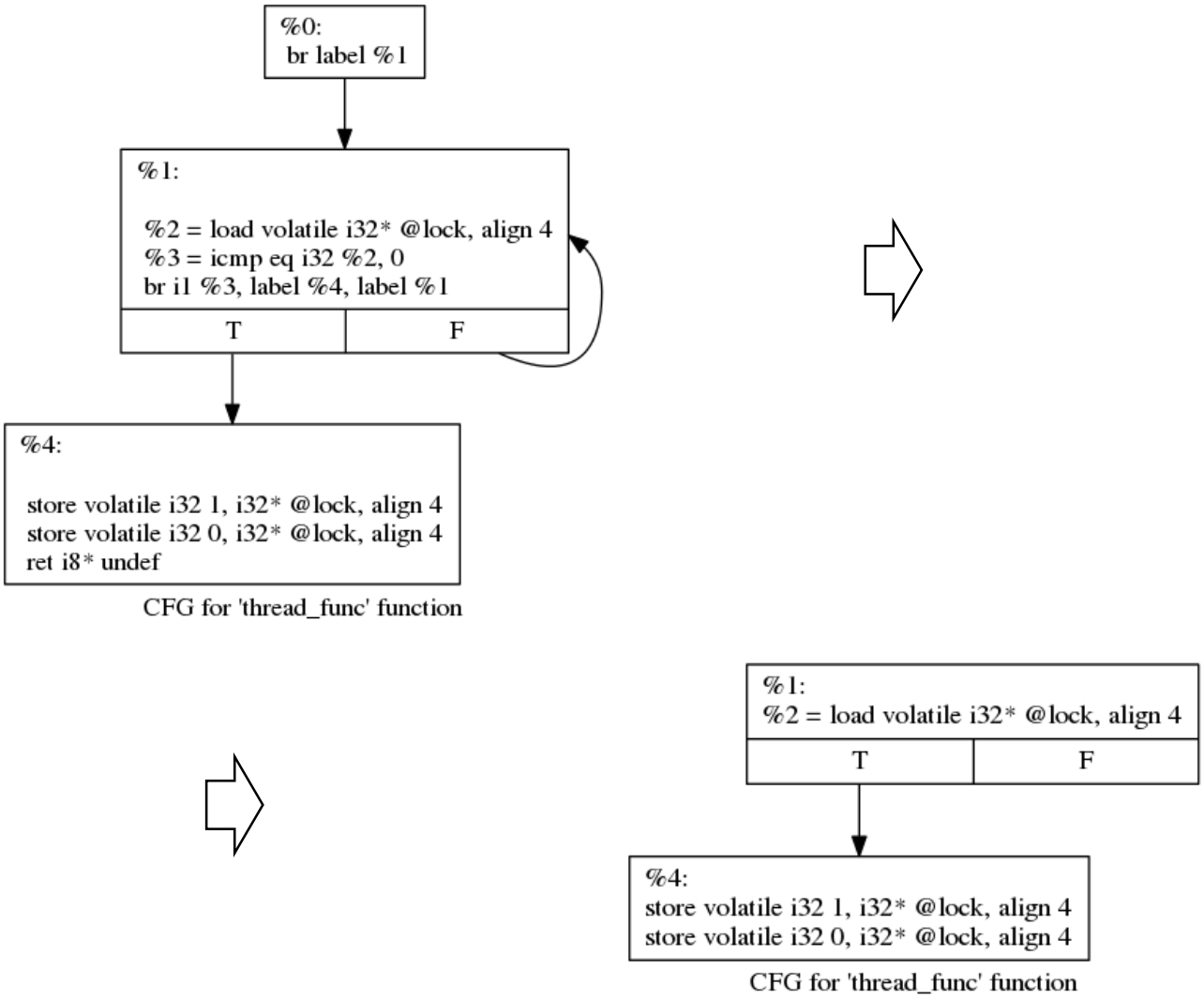


Рис. 21. Преобразование IR в Reduced CF процедуры thread_func в некорректном алгоритме спин-блокировки.

Листинг Reduced CFG при этом будет сгенерирован следующим образом:

```
Init
W   lock    lock=0
Thread
R   lock    V2=lock U
R   V2      V2==0   C
W   lock    lock=1  U
W   lock    lock=0  U
Thread
R   lock    V2=lock U
R   V2      V2==0   C
W   lock    lock=1  U
W   lock    lock=0  U
End
```

Данный листинг также создает 14 классов эквивалентностей (см. Приложение), которые находятся анализатором на Wolfram Mathematica. Анализ путей данных классов эквивалентностей на расчетном графе показывает 8 возможных путей исполнения на графе совместного исполнения потоков, которые изображены на рис. 22.

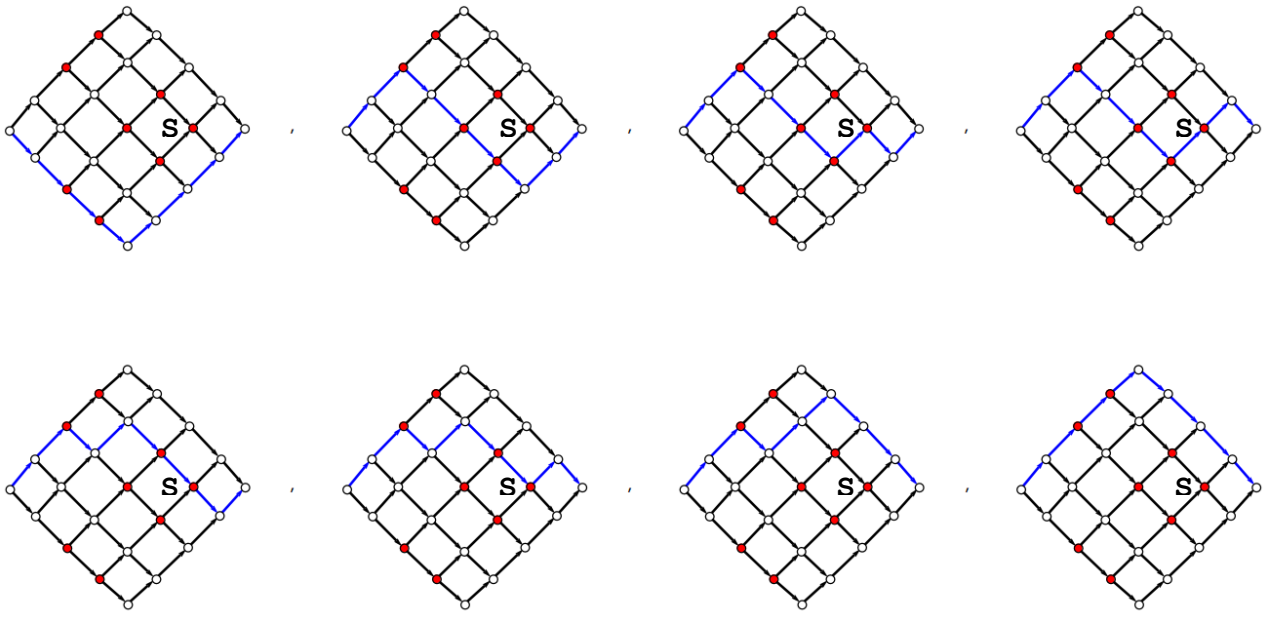


Рис. 22. Допустимые пути совместного исполнения потоков в некорректном алгоритме спин-блокировки.

Как видно на рис. 22 среди найденных допустимых путей есть те, которые проходят через точку S, которая говорит о наличии двух одновременно потоков в критической секции. Отсюда мы заключаем, что данная реализация алгоритма не разрешает критическую секцию и возможно неразрешимое состояние гонки.

Данный пример демонстрирует возможность метода статического анализа на оптимизированном контексте промежуточного представления программы и разработанного комплекса программ обнаруживать состояния гонки.

3.3. Алгоритм Петерсона

Реализация алгоритма представлена в п. 2.2 [32-34]. Поскольку реализация потоков симметричная, то достаточно рассмотреть один из них. Начальное представление потоков в виде LLVM IR показано на рис. 6. Opt IR, полученный Анализатором IR, показан на рис. 7.

Далее Анализатор IR получает Reduced Opt IR, который изображен на рис. 11. Из Reduced Opt IR получается Reduced CFG. Детали этого представления были описаны в п. 2.6, при этом редуцированный граф изображен на рис. 12.

Reduced CFG подается анализатору на Wolfram Mathematica, который строит граф совместного исполнения, ищет классы эквивалентности, определяет пути на каждом классе эквивалентности и проводит их анализ на расчетном графе. Промежуточные итоги работы данного анализатора представлены в Приложении данной работы.

Стоит упомянуть про входные данные, которые подаются в анализатор на Wolfram Mathematica. Reduced CFG из формата LLVM IR преобразован в соответствующий листинг, который имеет следующий вид:

```
Init
W     flag0     flag0=0
W     flag1     flag1=0
Thread
W     turn      turn=1    U
```

```
W     flag0     flag0=1   U
R     turn      VR2=turn  U
R     VR2       VR2==1    C
R     flag1     VR5=flag1 U
R     VR5       VR5==1    C
W     flag0     flag0=0   U
Thread
W     turn      turn=0    U
W     flag1     flag1=1   U
R     turn      VR2=turn  U
R     VR2       VR2==0    C
R     flag0     VR5=flag0 U
R     VR5       VR5==1    C
W     flag1     flag1=0   U
End
```

Результат анализа на Reduced CFG показал 24 класса эквивалентности (см. Приложение), анализ представителей классов показал 2 доступных пути исполнения потоков (см. рис. 15) на расчетном графе, которые не проходят одновременно по критической секции. Отсюда вытекает вывод, что алгоритм корректно разрешает критическую секцию и, как следствие, заключаем, что реализованная модель статического анализа на оптимизируемом промежуточном представлении для алгоритма Петерсона работает корректно.

3.4. Стек Трейбера

Стек Трейбера – неблокирующая реализация стека, алгоритм которого был впервые описан в 1986 г. Р.К. Трейбером [20, 27, 36]. Оригинальная реализация алгоритма была на языке программирования HLASM [21]. Обычно реализация алгоритма предполагает два интерфейса взаимодействия со стеком: PUSH(), POP(). Ниже приведена реализация алгоритма на языке Си.

```
/* Non-blocking stack using Treiber's algorithm */

#define CAS(a,b,c) __sync_bool_compare_and_swap (a,b,c)
typedef struct node Node;

Node * volatile GlobalTop;

struct node {
    Node * next;
    int value;

    node (int val) {
        value = val;
        next = NULL;
    }
};

// -------------------------------------------------------------
// Push an element onto the stack
// -------------------------------------------------------------
void PUSH(int value) {
    Node * oldHead;
    Node * newHead = new Node(value);
    do {
        oldHead = GlobalTop;
        newHead->next = oldHead;
    } while (!CAS(&GlobalTop, oldHead, newHead));
}

// -------------------------------------------------------------
// Pop an element off the stack
// -------------------------------------------------------------
int POP() {
    Node * oldHead;
    Node * newHead;
    do {
        oldHead = GlobalTop;
        if (oldHead == NULL) return 0;
        newHead = oldHead->next;
    } while (!CAS(&GlobalTop, oldHead, newHead));
    return oldHead->value;
}
```

Разделяемыми переменными для данного алгоритма являются элементы стека. В начальный момент разделяемой переменной является только указатель на вершину стека.

Рассмотрим преобразования Анализатора IR. На рис. 23 изображено начальное представление POP() интерфейса, на рис. 24 также изображено

начальное представления PUSH() интерфейса. Анализатор IR преобразует каждый из интерфейсов в Opt IR, результат преобразования изображен на рис. 25. Как видно из рис. 25, линеаризация графа с помощью оптимизаций компилятора LLVM дает существенную выгоду с точки зрения исследуемого метода статического анализа. Далее на рис. 26 приведен Reduced Opt IR интерфейсов, полученных Анализатором IR. При получении Reduced Opt IR для процедуры PUSH() были применены дополнительные правила удаления:

- удалить дополнительно store инструкции из %0 узла (см. рис. 26), поскольку они и их аргументы не связаны с разделяемой переменной GlobalTop;
- не удалять Call инструкцию из %0 узла (см. рис. 26), осуществляющая вызов библиотечной функции new(), которая возвращает указатель на динамическую память.

На заключительном этапе трансформации LLVM IR получаем Reduced CFG исследуемых процедур PUSH() и POP(). На рис. 26 видны пустые CFG-узлы, что дает основание для более глубокой линеаризации CFG. На рис. 27 приведен Reduced CFG соответствующих процедур. Проведем поиск состояний гонок на полученном Reduced CFG процедур.

Рассмотрим три сценария параллельного исполнения двух поток:

1. Первый поток вызывает POP(), второй поток – POP().
2. Первый поток – PUSH(), второй поток – PUSH().
3. Первый поток – PUSH(), второй поток – POP().

Сценарии выше являются базовыми сценариями, и любые другие сценарии оперирования со стеком через PUSH() и POP() есть композиция базовых сценариев. Поэтому доказательство корректной или некорректной работы алгоритма на данных базовых сценариях будет в полной мере полным доказательством корректности или некорректности всего алгоритма.

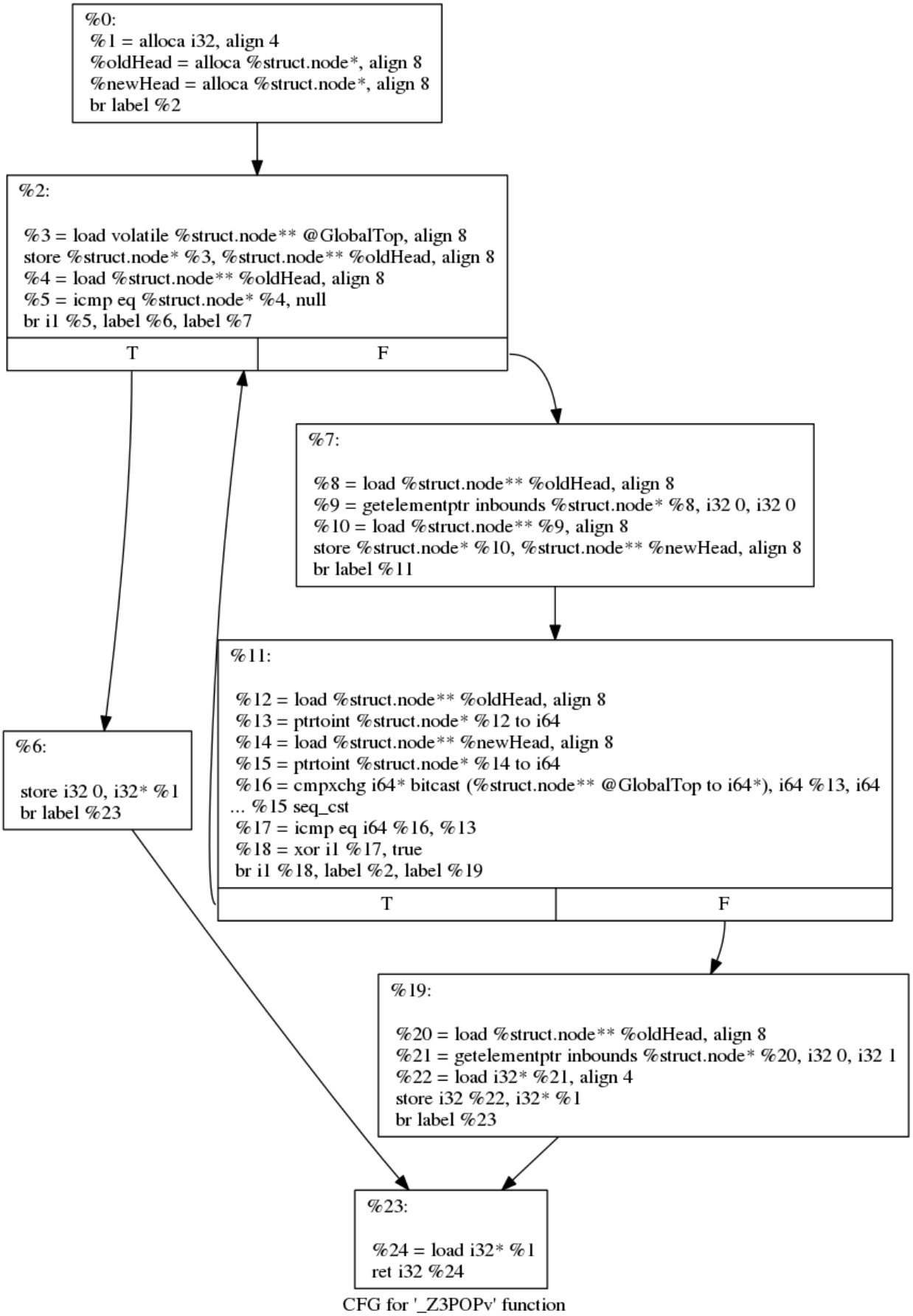


Рис. 23. Начальный CFG процедуры POP() в стеке Трейбера.

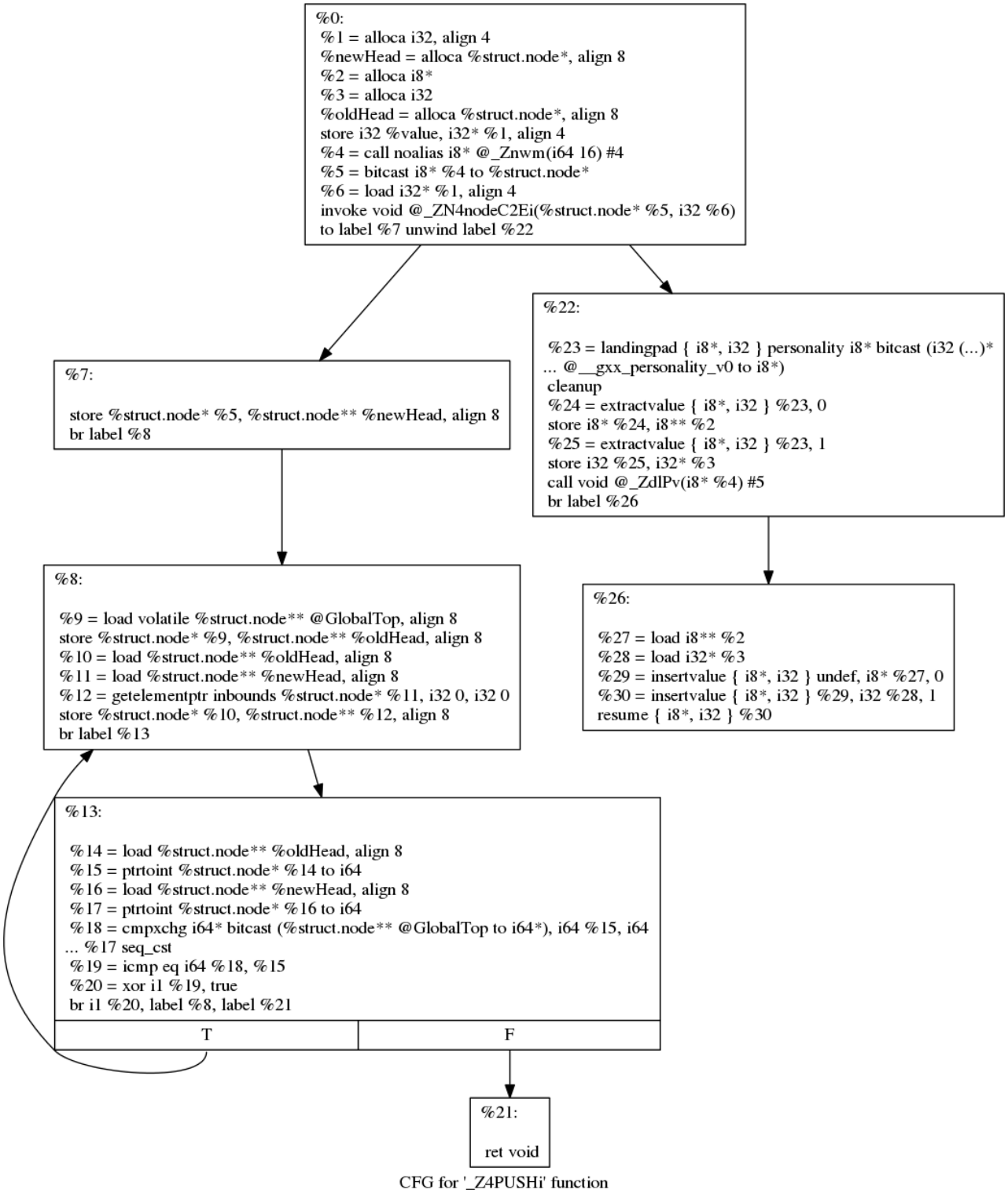


Рис. 24. Начальный CFG процедуры PUSH() в стеке Трейбера.

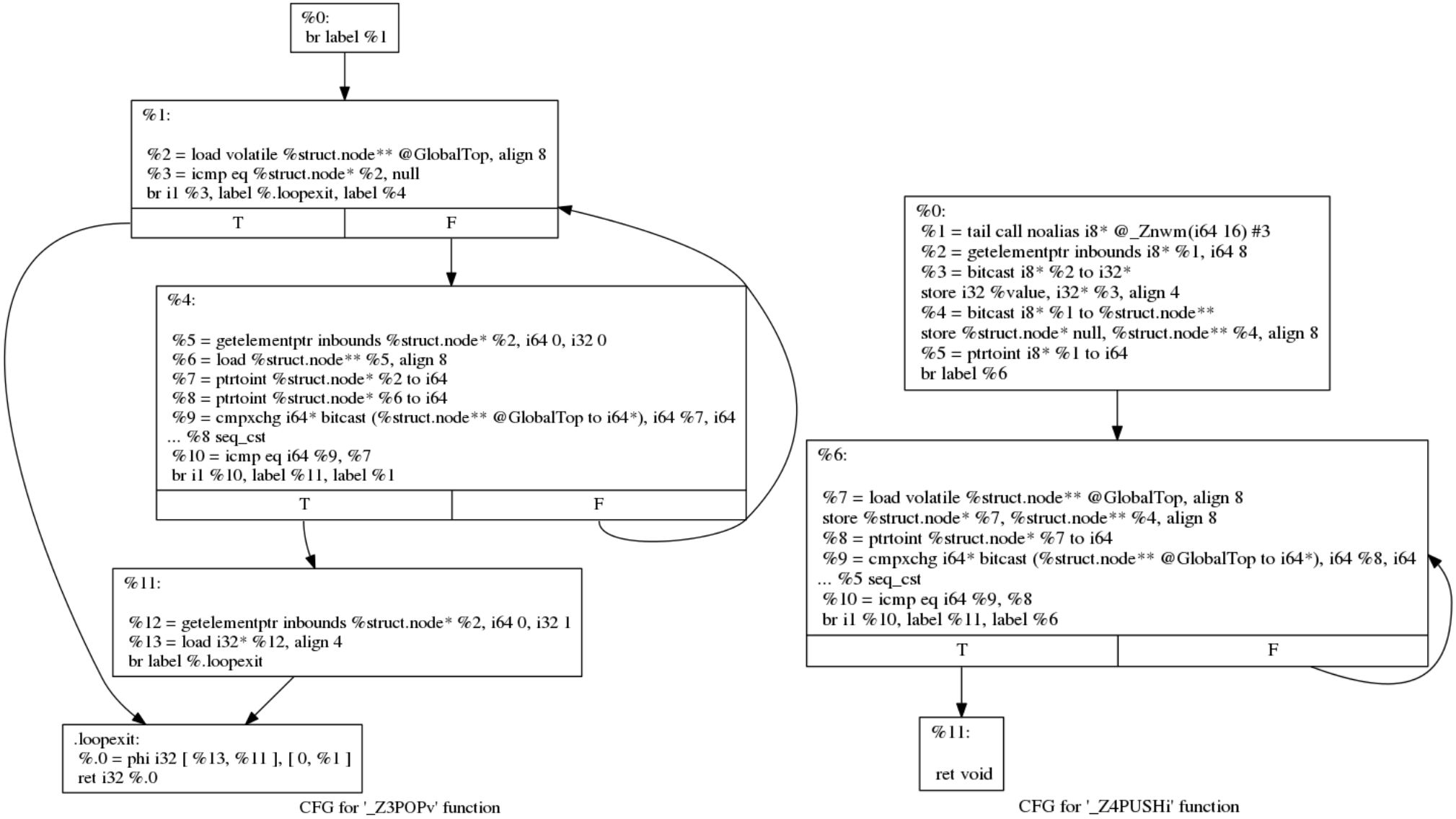


Рис. 25. Opt IR процедур POP() и PUSH() в стеке Трейбера.

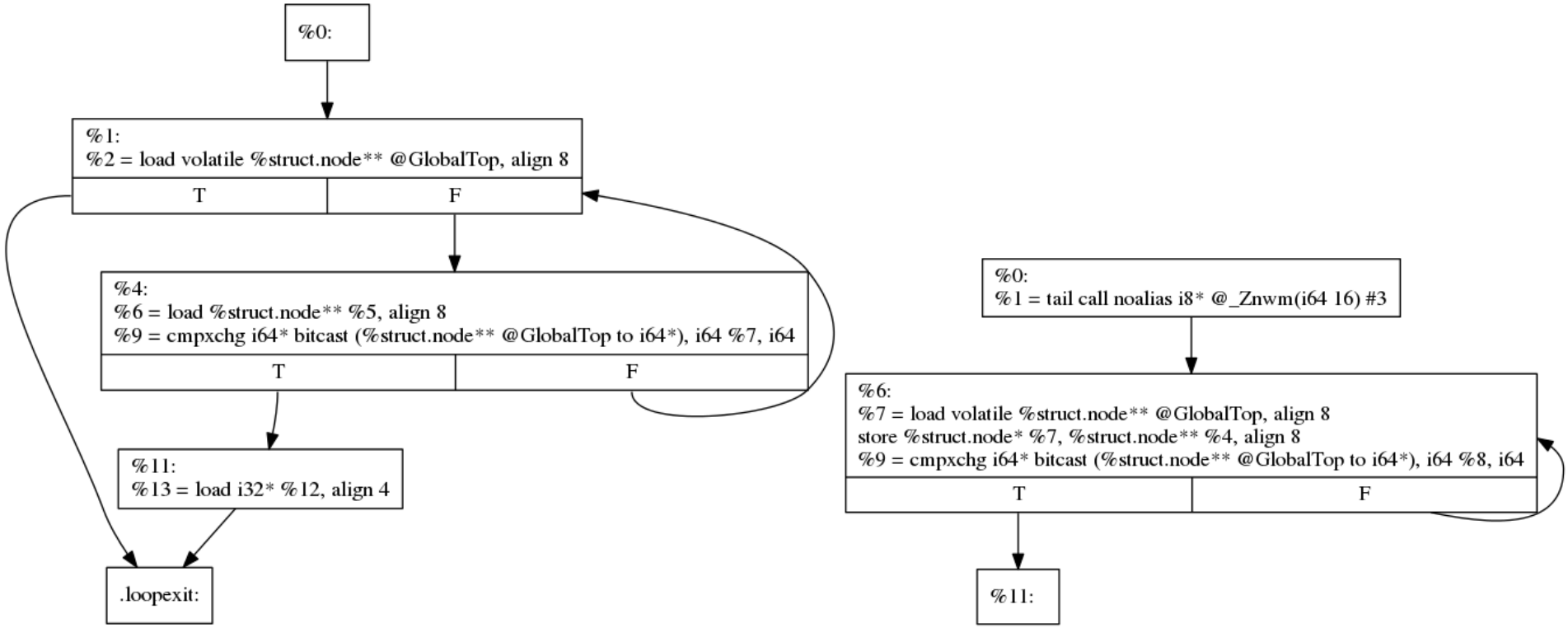
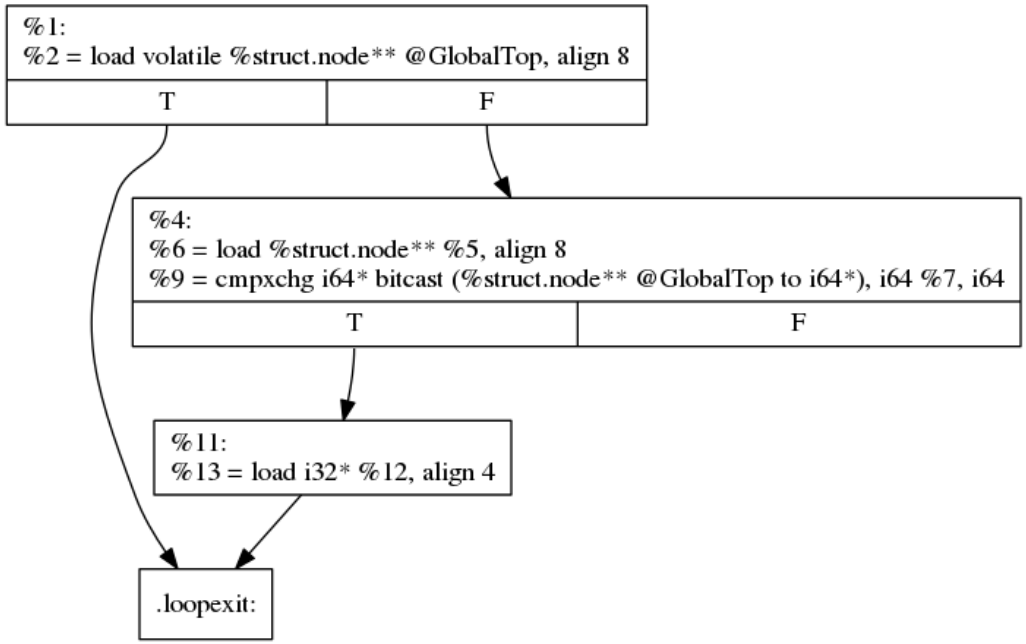
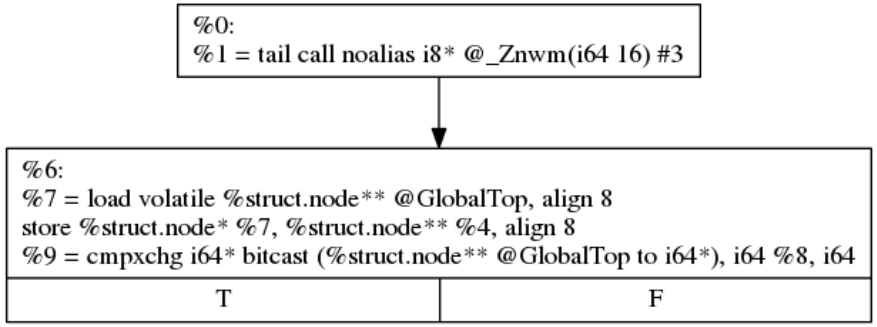


Рис. 26. Reduced Opt IR процедур POP() и PUSH() в стеке Трейбера.



CFG for '_Z3POPv' function



CFG for '_Z4PUSHi' function

Рис. 27. Reduced CFG процедур POP() и PUSH() в стеке Трейбера.

Рассмотрим первый сценарий: первый поток – POP(), второй поток – POP(). Преобразованный Reduced CFG в соответствующий листинг для анализатора на Wolfram Mathematica имеет следующий вид:

```
Init
W   GlobalTop   GlobalTop=<ADDR>
Thread
R   GlobalTop   VR2=GlobalTop   U
R   VR2         VR2!=0          C
R   VR2         VR6=VR2         U
R   GlobalTop   VR2==GlobalTop  CS
W   GlobalTop   GlobalTop=VR6   U
R   GlobalTop   GlobalTop==VR2  C
R   VR2         VR13=VR2        U
Thread
R   GlobalTop   VR2=GlobalTop   U
R   VR2         VR2!=0          C
R   VR2         VR6=VR2         U
R   GlobalTop   VR2==GlobalTop  CS
W   GlobalTop   GlobalTop=VR6   U
R   GlobalTop   GlobalTop==VR2  C
R   VR2         VR13=VR2        U
End
```

Делается предположение, что изначально указатель GlobalTop имеет не нулевое значение <ADDR>. Инструкция cmpxchg отображается в две операции в листинге:

```
R   GlobalTop   VR2==GlobalTop  CS  - атомарное сравнение,
W   GlobalTop   GlobalTop=VR6   U   - запись значения.
```

Если в отображаемой в листинг инструкции есть аргумент, который в листинге еще не объявлен, то используется промежуточное представление Opt LLVM IR для анализа этого аргумента и тех инструкций, которые готовят данный аргумент. Если среди инструкций-предков есть инструкции, связанные с

разделяемой памятью, и данная инструкция уже отображена в листинге, то результирующий виртуальный регистр такой инструкции заменяет аргумент в анализируемой инструкции. В противном случае аргумент инструкции не изменяется.

Таким образом, расчет на полученном листинге Reduced CFG показал 12 классов эквивалентностей (см. Приложение), а анализ представителей этих классов (см. Приложение) показал 4 допустимых пути на графе совместного исполнения потоков (см. рис. 28). Критической секцией в процедуре будет начало операция

«W GlobalTop GlobalTop=VR6 U»,

а точка одновремнного нахождения двух потоков критической секции обозначена буквой S на рис. 28.

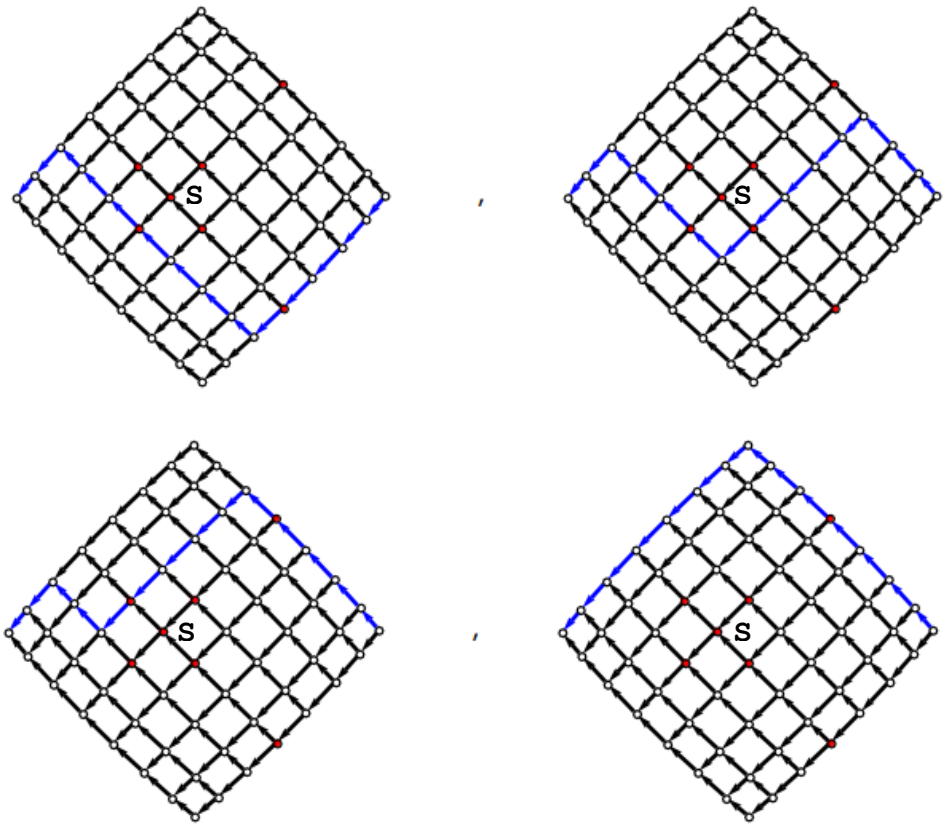


Рис. 28. Допустимые пути совместного исполнения потоков в при параллельном вызове процедуры POP() в каждом из потоке.

Как видно на рис. 28, нет ни одного допустимого пути, который проходил бы через точку S, что означает отсутствие одновременного нахождения двух потоков в критической секции. Отсюда заключаем, что процедура POP() успешно реализует критическую секцию и не приводит к состоянию гонок при параллельном исполнении ее в потоках.

Рассмотрим второй сценарий: первый поток – PUSH(), второй поток – PUSH(). Преобразованный Reduced CFG в соответствующий листинг для анализатора на Wolfram Mathematica имеет следующий вид:

```
Init
W     GlobalTop GlobalTop=<ADDR>
Thread
R   VR1         VR1=<ADDR>      U
R   GlobalTop   VR7=GlobalTop   U
W   VR7         VR4=VR7         U
R   GlobalTop   VR7==GlobalTop  CS
W   GlobalTop   GlobalTop=VR1   U
R   GlobalTop   GlobalTop==VR7  C
Thread
R   VR1         VR1=<ADDR>      U
R   GlobalTop   VR7=GlobalTop   U
W   VR7         VR4=VR7         U
R   GlobalTop   VR7==GlobalTop  CS
W   GlobalTop   GlobalTop=VR1   U
R   GlobalTop   GlobalTop==VR7  C
End
```

Анализатор на Wolfram Mathematica интерпретирует <ADDR> как некоторое уникальное число, ассоциированное с адресом. Таким образом, анализатором были выявлены 20 классов эквивалентности на данном листинге Reduced CFG (см. Приложение). Критическая секция в данном случае начинается в точке перед выполнением операции

«W GlobalTop GlobalTop=VR1 U»,

при этом одновременное нахождение потоков в критической секции обозначена буквой S на расчетном графе (см. рис. 29).

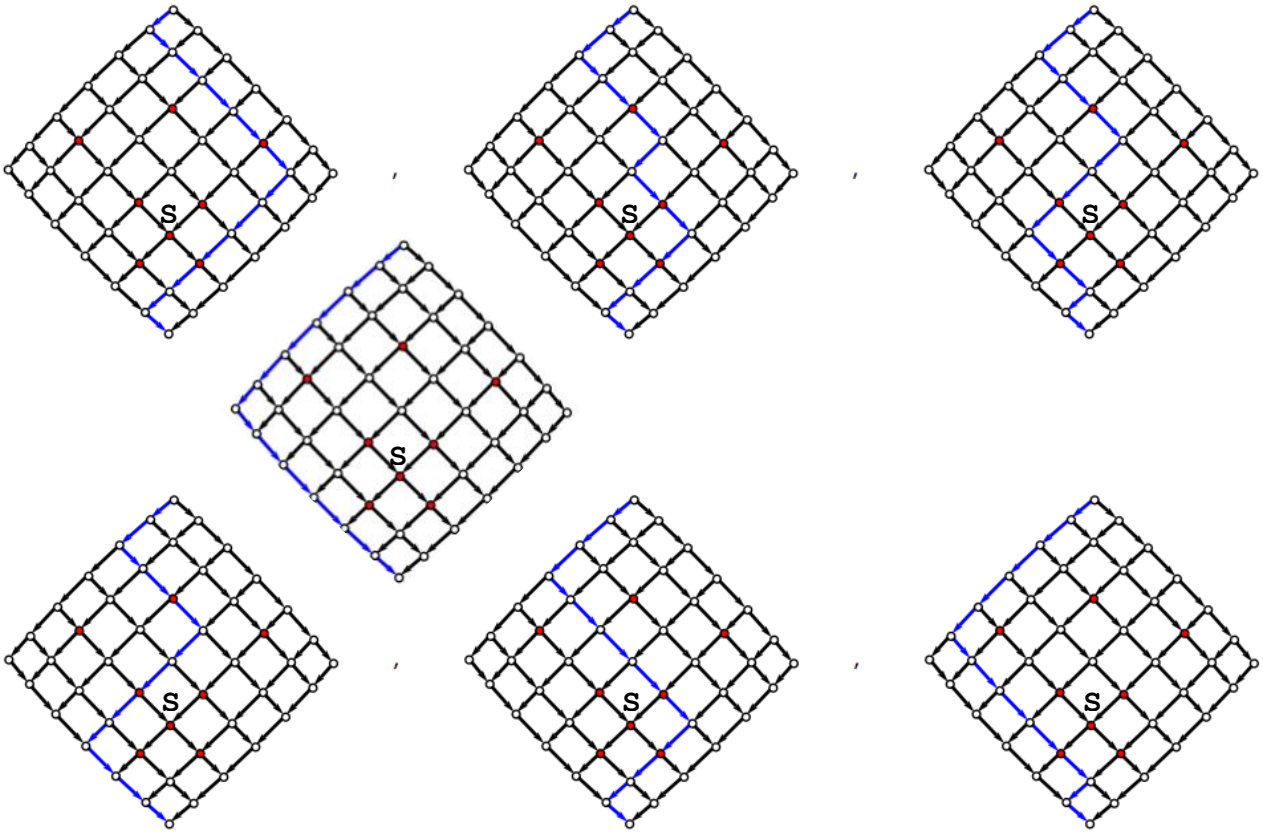


Рис. 29. Допустимые пути совместного исполнения потоков в при параллельном вызове процедуры PUSH() в каждом из потоке.

Анализ представителей классов выявил 7 допустимых путей, которые показаны на рис. 29. Ни одни из допустимых путей не проходит через точку S, что говорит о корректном разрешении критической секции в случае параллельного вызова процедур PUSH() в потоках.

Рассмотрим третий сценарий: первый поток – POP(), второй поток – PUSH(). Преобразованный Reduced CFG в соответствующий листинг для анализатора на Wolfram Mathematica имеет следующий вид:

```
Init
W     GlobalTop GlobalTop==<ADDR>
Thread
R   GlobalTop   VR2=GlobalTop   U
R   VR2         VR2!=0          C
R   VR6         VR6=VR2         U
R   GlobalTop   VR2==GlobalTop  CS
W   GlobalTop   GlobalTop=VR6   U
R   GlobalTop   GlobalTop==VR2  C
R   VR13        VR13=VR2        U
Thread
R   VR1         VR1==<ADDR>     U
R   GlobalTop   VR7=GlobalTop   U
W   VR4         VR4=VR7         U
R   GlobalTop   VR7==GlobalTop  CS
W   GlobalTop   GlobalTop=VR1   U
R   GlobalTop   GlobalTop==VR7  C
End
```

На данном листинге Reduced CFG было обнаружено 12 классов эквивалентности (см. Приложение). Начало критической секции для первого потока начинается в точке выполнения операции

«W GlobalTop GlobalTop=VR6 U»,

в свою очередь для второго потока критическая секция в точке

«W GlobalTop GlobalTop=VR1 U»,

таким образом, точка на расчетном графе, где оба потока находятся в критической секции обозначена точкой S (см. рис. 30). Анализ представителей классов эквивалентности показывает 4 допустимых путей, причем, как видно на рис. 30, ни один из них не проходит через точку S.

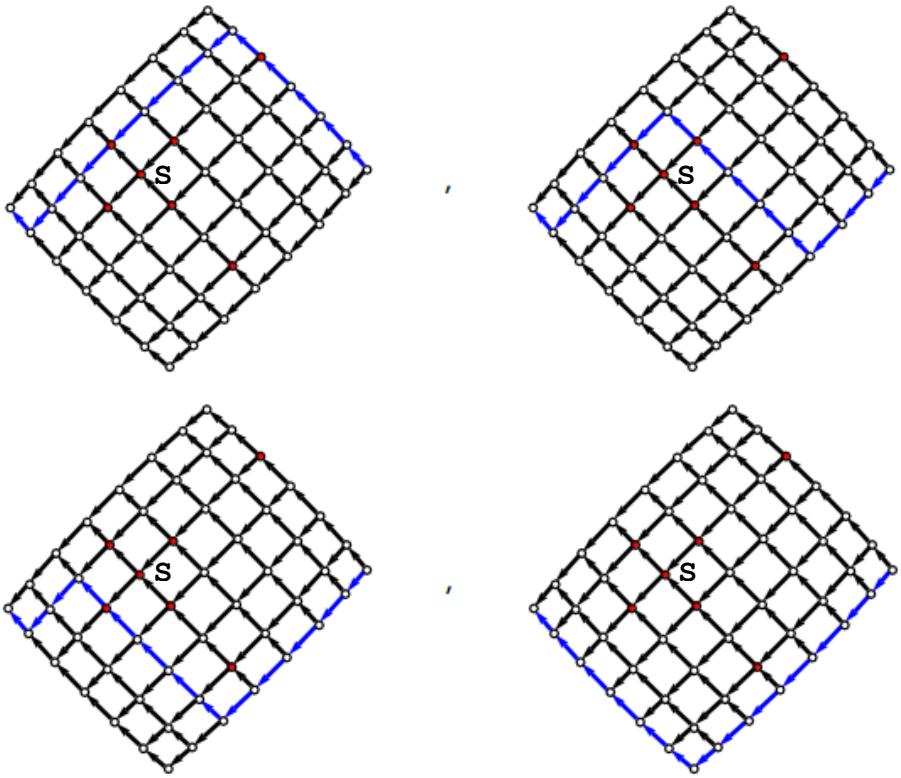


Рис. 30. Допустимые пути совместного исполнения потоков в при параллельном вызове процедур PUSH() и POP() в каждом из потоке.

Поскольку ни один из допустимых путей не проходит через точку S, заключаем, что одновременно два потока в критической секции находиться не могут. Отсюда вывод – процедуры POP() и PUSH() корректно разрешают критическую секцию при параллельном исполнении в потоках.

Все три базовых случая взаимодействия интерфейсов стека Трейбера (POP-POP, PUSH-PUSH, PUSH-POP) показали корректную работу с критической секцией и к состоянию гонки не приводят. Поскольку любое параллельное взаимодействие со стеком через интерфейс PUSH(), POP() можно представить через совокупность базовых взаимодействий, заключаем, что алгоритм стека Трейбера работает корректно и не приводит к состоянию гонок.

3.5. Выводы

Результаты экспериментов указывают на успешность работы разработанной системы верификации многопоточных программ на основе адаптированной математической модели статического анализа с использованием линеаризованного графа потока управления программы. Состоятельность модели следует вследствие совпадения полученных результатов с общепризнанными результатами модельных задач, описанных в компетентных научных трудах.

Платформонезависимость промежуточного представления широко используемого оптимизирующего компилятора CLANG&LLVM позволяет на практике использовать систему верификации для промышленного ПО для различных архитектур и платформ.

Заключение

В диссертационной работе была исследована, адаптирована и реализована математическая модель статического анализа многопоточных программ на основе графа совместного исполнения потоков с использованием оптимизированного промежуточного представления программы в качестве анализируемого контекста в статическом анализе.

Применение платформонезависимого промежуточного представления промышленного оптимизирующего компилятора позволило упростить анализ и повысить применимость исследуемого метода статического анализа в промышленных комплексах программ. Разработанный метод линеаризации графа потока управления программы для статического анализа позволил существенно сократить анализируемый контекст программы, улучшив качества статического анализа.

В процессе исследования получены следующие результаты:

1. Программная реализация системы верификации состояния гонок в многопоточных алгоритмах на основе IR промышленного компилятора, которая позволяет получать IR программ различных языков высокого уровня, разных платформ и архитектур. Использование общепризнанного и популярного компилятора гарантирует применимость метода к широкому классу задач на долгие годы вперед. Возможность получения унифицированного IR программ, ориентированных на такие архитектуры как Intel, ARM, Power, позволяет значительно расширить класс применимости метода в среде коммерческого ПО. Как следствие, появляется весомое средство оценки качества при разработки коммерческого ПО.

2. Адаптированная математическая модель верификации многопоточных алгоритмов. Приведены доказательства утверждений и теорем, которые позволили применять модель на оптимизированном промежуточном представлении.

3. Метод линеаризации графа потока управления для представленного анализа. Особенность представленного метода статического анализа позволила создать дополнительную оптимизацию графа потока управления, с помощью которой удалось в некоторых случаях избавиться от лишних узлов графа потока управления и, как следствие, уменьшить количество ветвления управления на графе.

4. Успешное применение системы верификации состояний гонок на модельных задачах, которые позволили не только подтвердить надежность некоторых многопоточных алгоритмов, в том числе неблокирующих, но обнаружить состояние гонки в заведомо некорректных алгоритмах.

Список использованных источников

1. Битнер В.А. Статический анализ кода алгоритмов с использованием линеаризации графа потока управления // Труды XVI научной конференции «Математическое моделирование и информатика». / Под ред. Д.Ю. Рязанова. – М.: ИЦ ФГБОУ ВПО МГТУ «СТАНКИН», 2014. – 308 с.

2. Битнер В.А. Исследование гипервизоров с открытым кодом для ARM-архитектуры // Актуальные проблемы науки: сб.науч.тр. по мат-лам Междунар. науч.-практ. конф. 30 мая 2011 г.: М-во обр. и науки РФ. Тамбов: Изд-во ТРОО "Бизнес-Наука-Общество", 2011. – 160с.

3. Битнер В.А., Тимербаев Н.Ф. Контекстно зависимая линеаризация графа потока управления в статическом анализе состояний гонок в многопоточных алгоритмах // Вестник Казанского технологического университета, 2014. – Т. 17, №15. – С. 187-192.

4. Битнер В.А., Тормасов А.Г. Анализ и сокращение промежуточного представления программы в модели статического анализа нахождения состояния гонки в многопоточных алгоритмах // Вестник КГТУ им. А.Н. Туполева, 2014. –№3. – С. 203-212.

5. Битнер В.А., Заборовский Н.В. Построение универсального линеаризованного графа потока управления для использования в статическом анализе кода алгоритмов // Моделирование и анализ информационных систем, 2013. – Т. 20, №2. – С. 166-177.

6. Битнер В.А. Интерпретация промежуточного представления исходной программы в оптимизирующем компиляторе на основе трансляции представления в программный код языка Си // Сборник научных трудов XXXV Международной молодежной научной конференции Гагаринские чтения. – М.: МАТИ, 2009. – С. 111-112.

7. Битнер В.А. Применение линеаризованного графа потока управления в модели статического анализа состояний гонок в многопоточных программах компиляторе // Труды 57-й научной конференции МФТИ «Актуальные проблемы фундаментальных и прикладных наук в современном информационном обществе». Управление и прикладная математика. Том 1. – М.: МФТИ, 2014. – 182 с.

8. Битнер В.А. Система интерпретации промежуточного представления программы в оптимизирующем компиляторе // Труды 52-й научной конференции МФТИ «Современные проблемы фундаментальных и прикладных наук». Часть I. Радиотехника и кибернетика. Том 1. – М.: МФТИ, 2009. – 182 с.

9. Дроздов А.Ю. Компонентный подход к построению оптимизирующих компиляторов: автореферат дис. ... доктора технических наук: 05.13.11 – Москва, 2010. – 50 с.

10. Дроздов А.Ю., Степаненков А.М. Методы комбинирования алгоритмов анализа и оптимизаций в современных оптимизирующих компиляторах. // Компьютеры в учебном процессе. – 2004. – №11. – С. 3-12.

11. Дроздов А.Ю., Новиков С.В. Исследование методов преобразования программы в предикатную форму для архитектур с явно выраженной параллельностью. // Компьютеры в учебном процессе, № 5, 2005.

12. Дроздов А.Ю., Новиков С.В., Шилов В.В. Эффективный алгоритм преобразования потока управления в поток данных. // Приложение к журналу “Информационные технологии”, № 2, 2005. С. 24-31.

13. Заборовский Н.В. Расчетная модель нахождения состояний гонок в многопоточных алгоритмах: дис. ... канд. физ.-мат. наук: 05.13.18 – Москва, 2011. – 104 с.

14. Кудрин М.Ю., Прокопенко А.С., Тормасов А.Г. Метод нахождения состояний гонки в потоках, работающих на разделяемой памяти. // Сборник научных трудов МФТИ. – М.: МФТИ, 2009. – № 4. – Том 1. – С. 181-201.

15. Прокопенко А.С. Статический анализ условий гонки в параллельных программах на разделяемой памяти: дис. … канд. физ.-мат. наук: 05.13.18 – Москва, 2010. – 107 с.

16. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compilers: principles, techniques, and tools, Addison-Wesley, Reading, Massachusetts, 1986.

17. Bacon D.F., Graham S.L., Sharp O.J. Compiler transformations for high performance computing // ACM Computing Surveys, 26(4), pp. 345–420, 1994.

18. Deutsch A. Interprocedural May-Alias Analysis for Pointers : Beyond k-limiting // In Proc. Programming Language Design and Implementation. – ACM Press, 1994.

19. Emanuelsson P., Nilsson U. A Comparative Study of Industrial Static Analysis Tools // Elsevier Science Publishers. – 2008. – Amsterdam, Netherlands.

20. Hendler D., Shavit N., Yerushalmi L. A Scalable Lock-free Stack Algorithm // Journal of Parallel and Distributed Computing, 70(1), pp. 1-12, 2010.

21. I. Corporation. IBM System/370 Extended Architecture, Principles of Operation. IBM Publication No. SA22-7085, 1983.

22. LLVM API Documentation [Электронный ресурс] – Режим дост.: http://llvm.org/docs/doxygen/html/IfConversion_8cpp.html

23. LLVM Documentation [Электронный ресурс] – Режим дост.: http://llvm.org/docs/

24. LLVM’s Analysis and Transform Passes [Электронный ресурс] – Режим дост.: http://llvm.org/docs/Passes.html

25. Muchnick, Steven S. Advanced Compiler Design and Implementation Morgan Kauffman, San Francisco, 1997, chapter 7.2.

26. Muchnick, Steven S. Advanced Compiler Design and Implementation, San Francisco: Morgan Kauffman, 1997.

27. Treiber R. K. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

28. Steensgaard B. Points-to Analysis in Almost Linear Time // In ACM POPL. – 1996.

29. Anderson T. E. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors // IEEE Transactions on Parallel and Distributed Systems, 1(1), pp. 6-16, 1990.

30. Vilgelm Bitner. Comparative analysis of open source hypervisors for ARM-architecture // The 8th Congress of the International Society for Analysis, its Applications, and Computation. – M.: PFUR, 2011. – 517 p. ISBN 978-5-209-04088-0

31. Peterson G. L. Myths about the mutual exclusion problem // IPL, vol. 12, No. 3, pp. 115-116, June 1981.

32. Alagarsamy K. Some myths about famous mutual exclusion algorithms // ACM SIGACT News, vol. 34, No. 3, pp. 94-103, September 2003.

33. Ball T., Rajamani S. The SLAM project: debugging system software via static analysis // In Proceedings of the 29 the Principles ACM of SIGPLAN-SIGACT Programming Languages Symposium on (POPL’02). – ACM Press, 2002. – С. 1-3.

34. James E. Burns, Yoram Moses Simple, fast, and practical non-blocking and blocking concurrent queue algorithms // PODC '96 Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, pp. 267-275, May 1996.

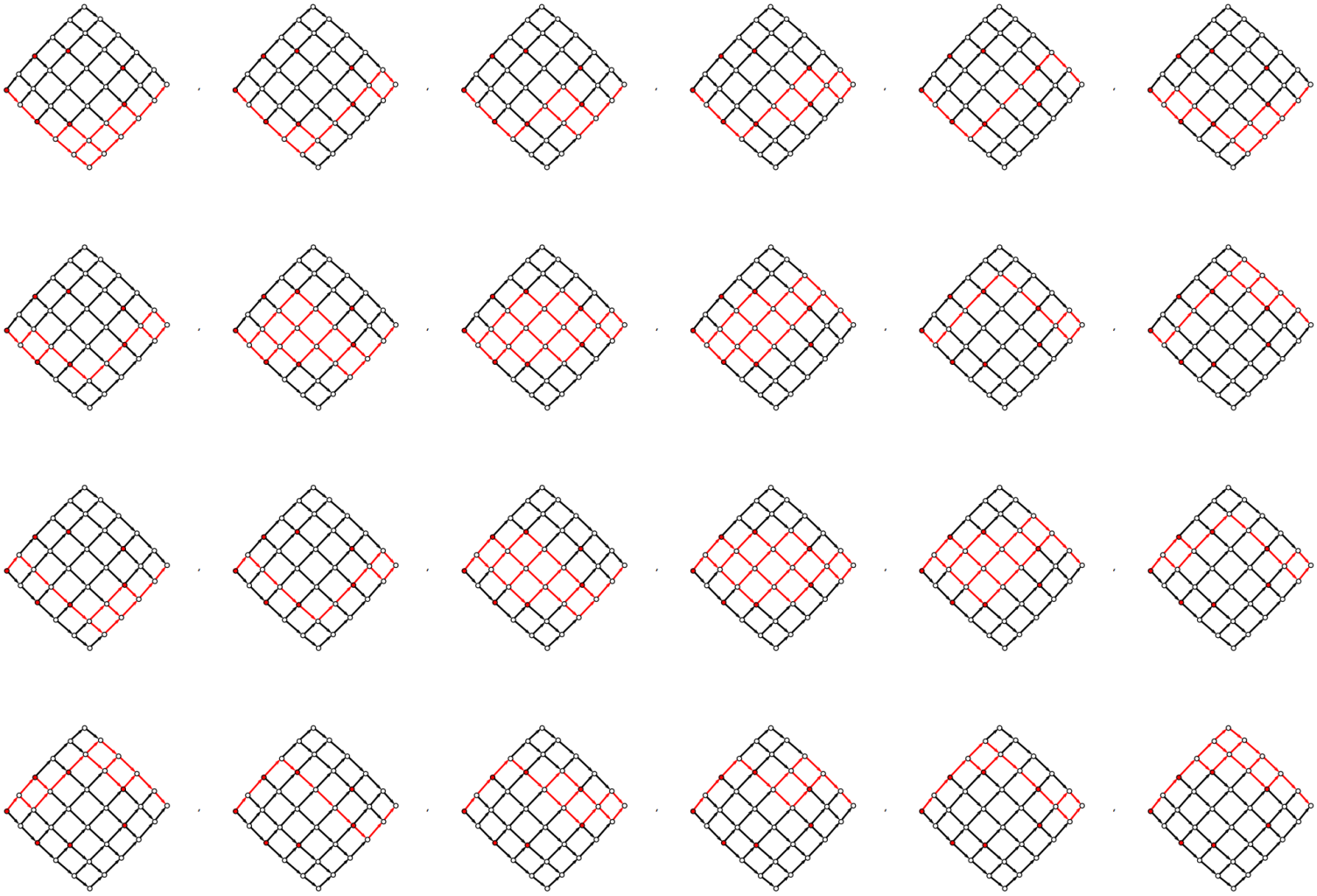
35. Barnes G. A method for implementing lock-free shared data structures // Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures, pp. 261–270, 1993.

36. Gao H., Hesselink W.H. A general lock-free algorithm using compare-and-swap // Information and Computation, pp. 225-241, February 2007.

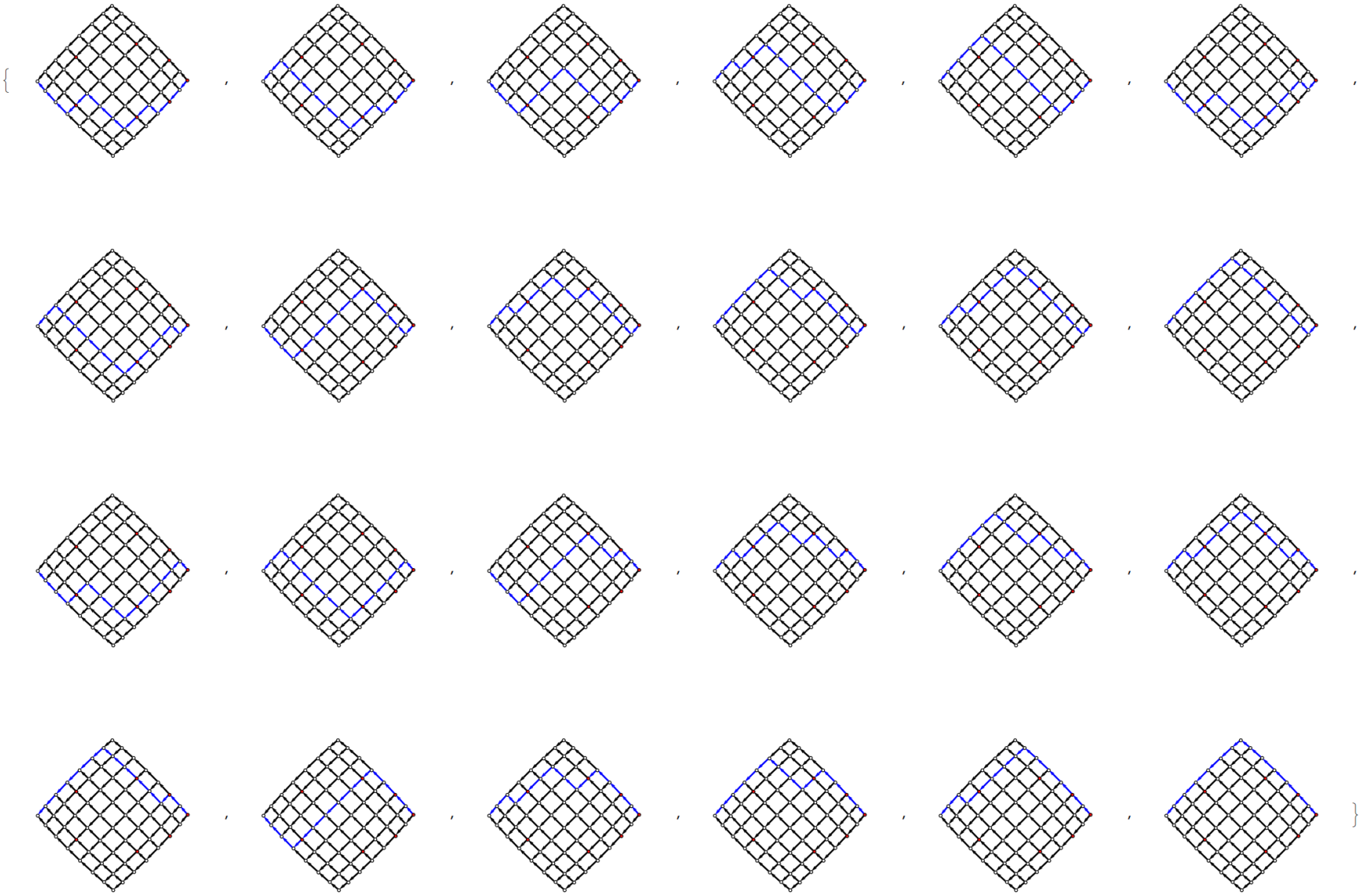
37. Herlihy M . A methodology for implementing highly concurrent data objects // ACM Transactions on Programming Languages and Systems (TOPLAS), pp. 745-770, November 1993.

Приложение

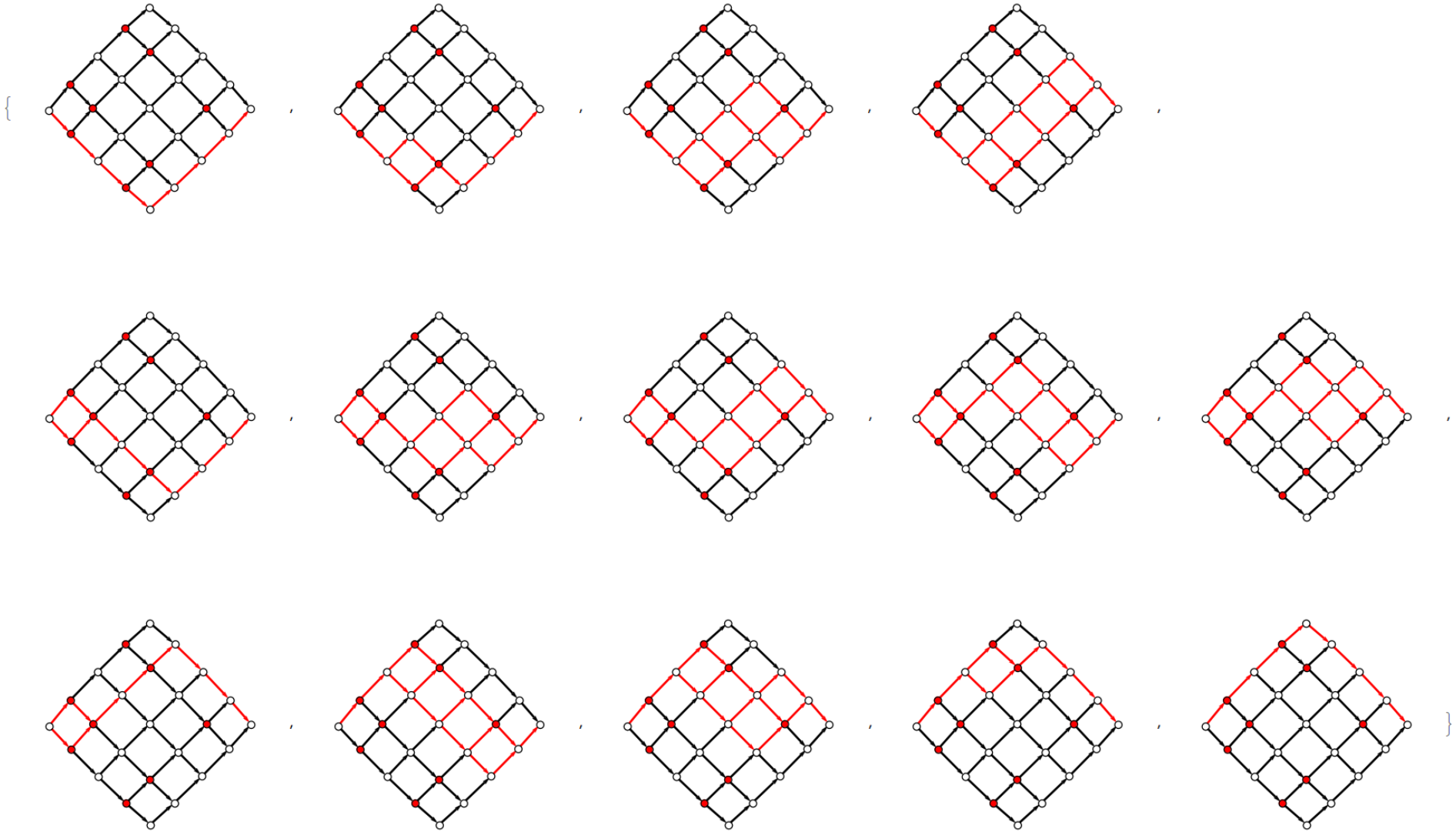
Классы эквивалентности (обозн. красным) в алгоритме Петерсона для двух потоков на графе совместного исполнения потоков построенный на Reduced CFG



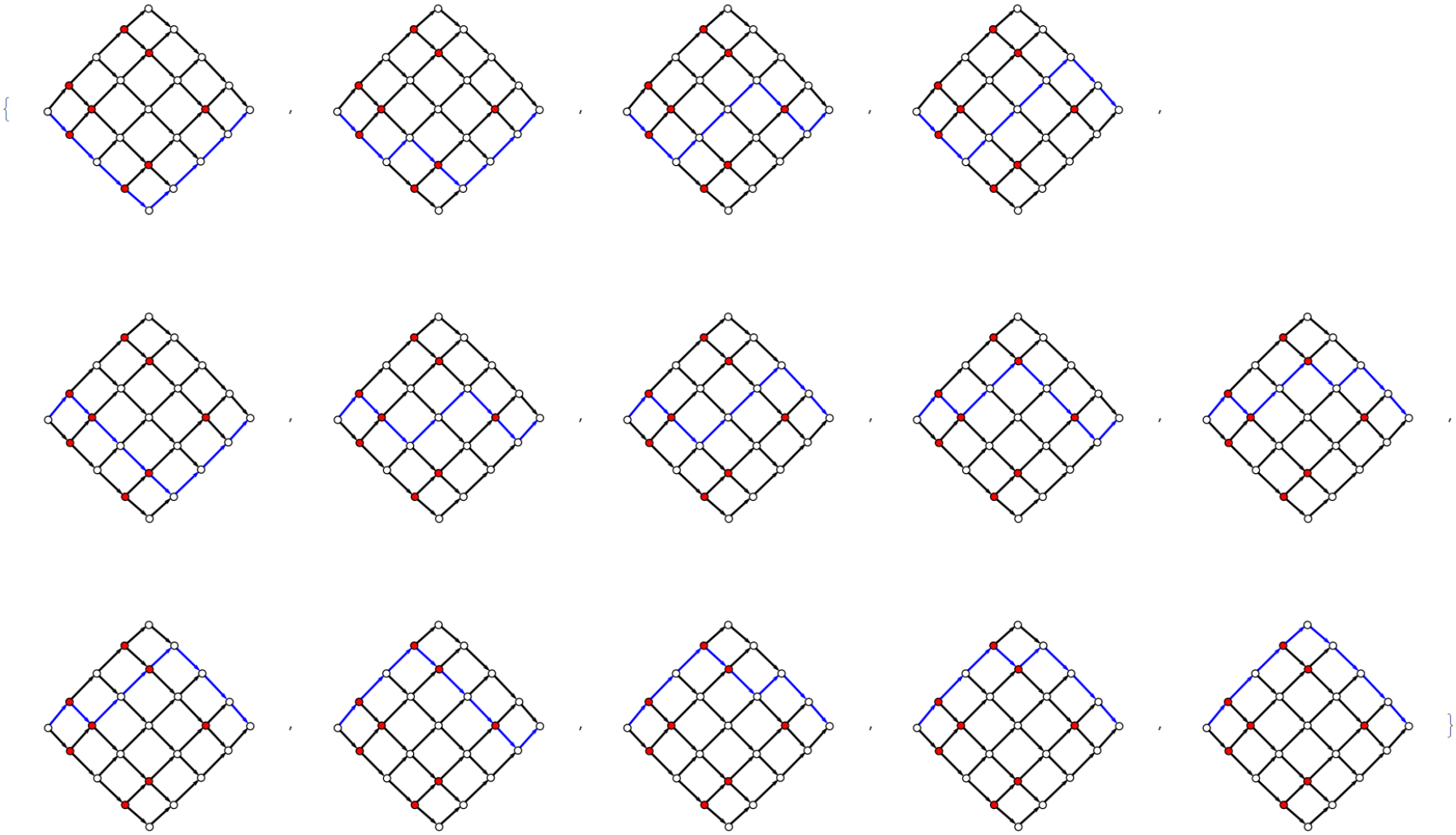
Представители классов эквивалентности (обозн. синим) в алгоритме Петерсона для двух потоков расчетном графе



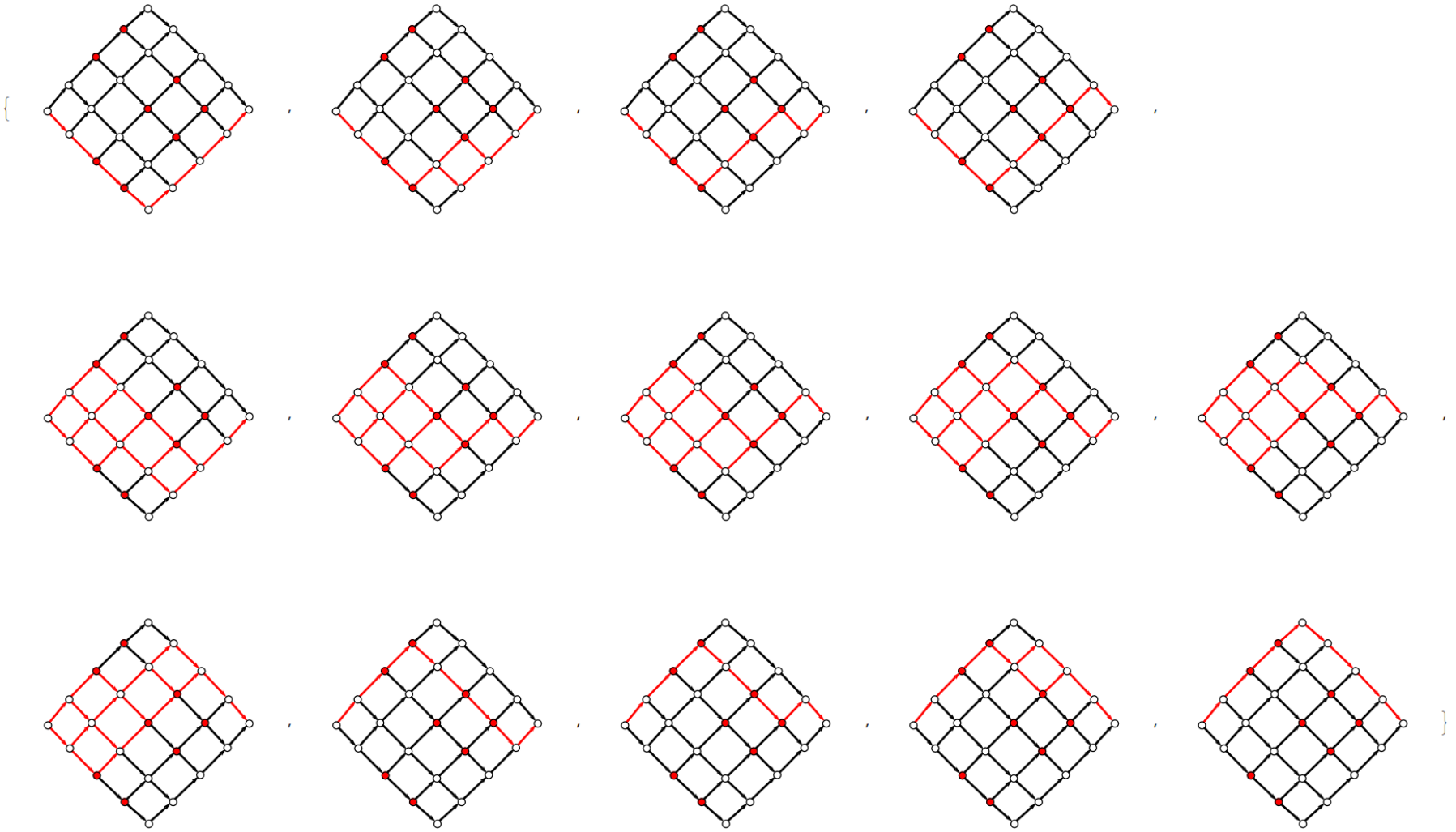
Классы эквивалентности (обозн. красным) в алгоритме Спин-блокировки для двух потоков на расчетном графе



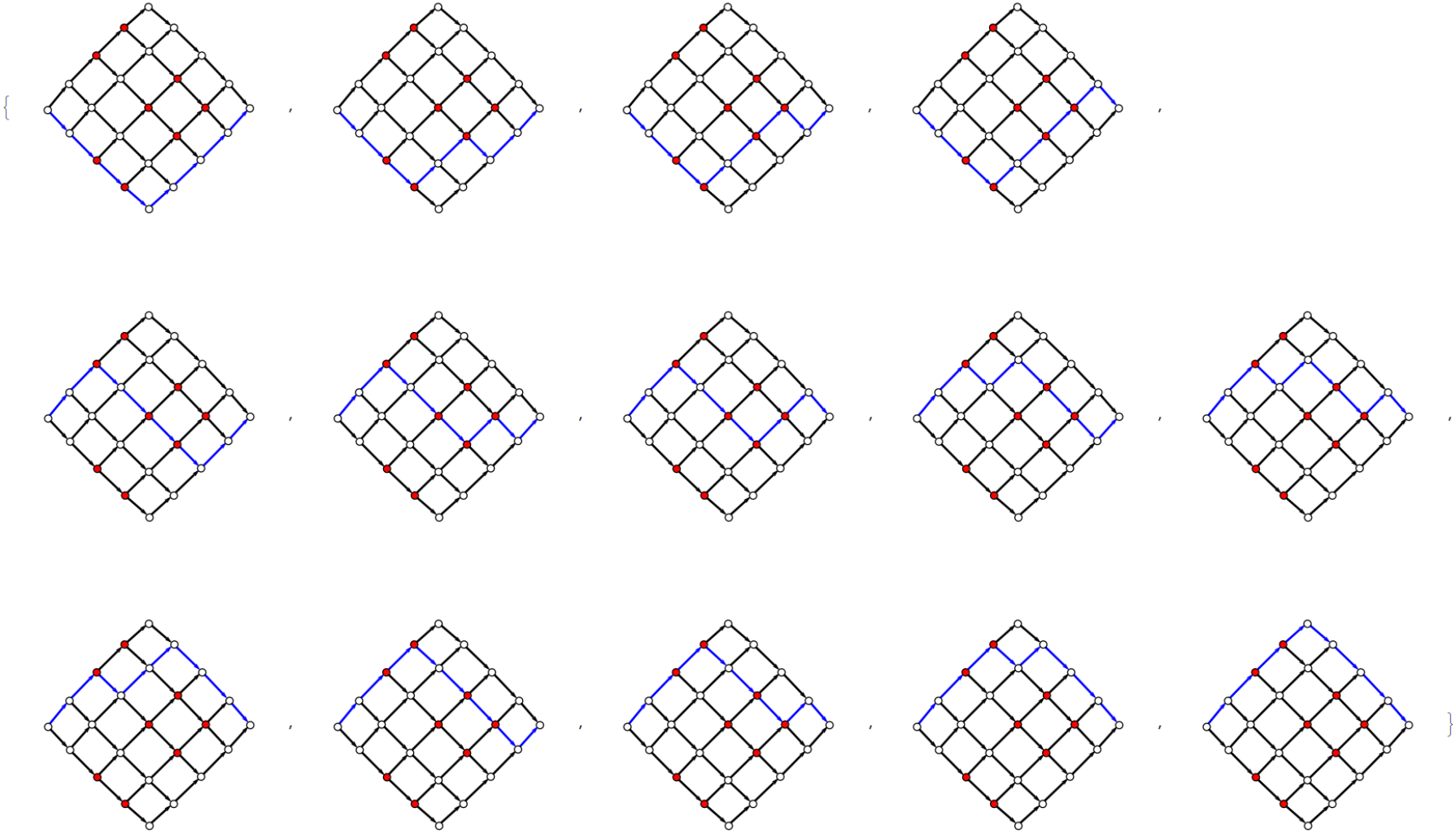
Представители классов эквивалентности (обозн. синим) в алгоритме Спин-блокировки для двух потоков расчетном графе



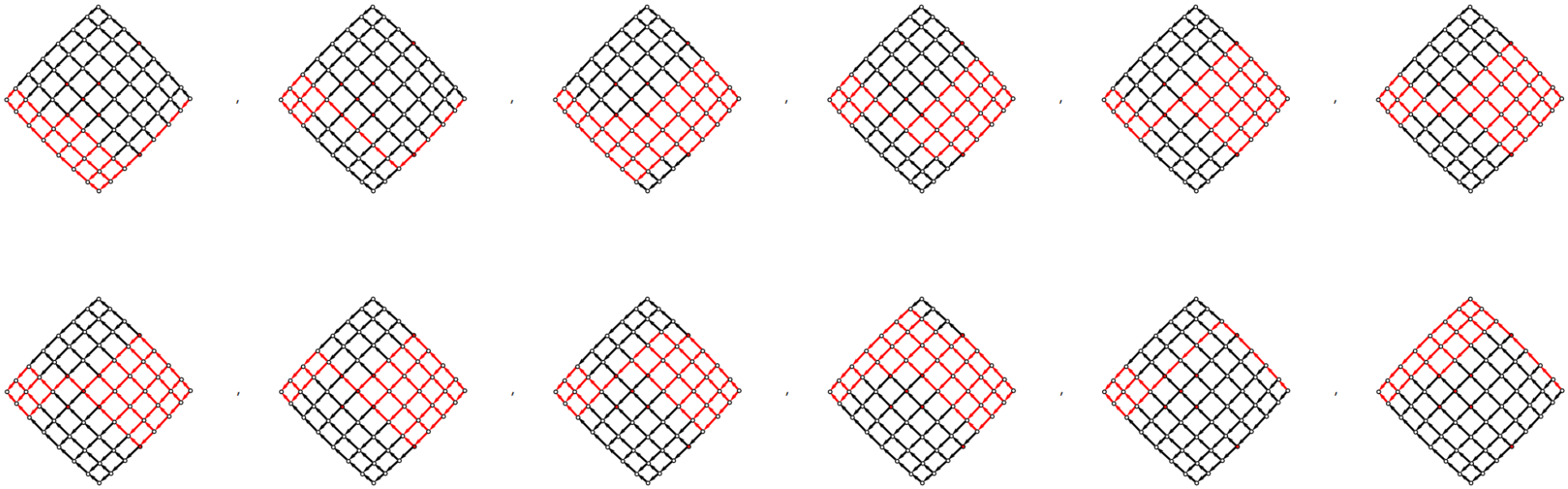
Классы эквивалентности (обозн. красным) в **некорректном** алгоритме Спин-блокировки для двух потоков на расчетном графе



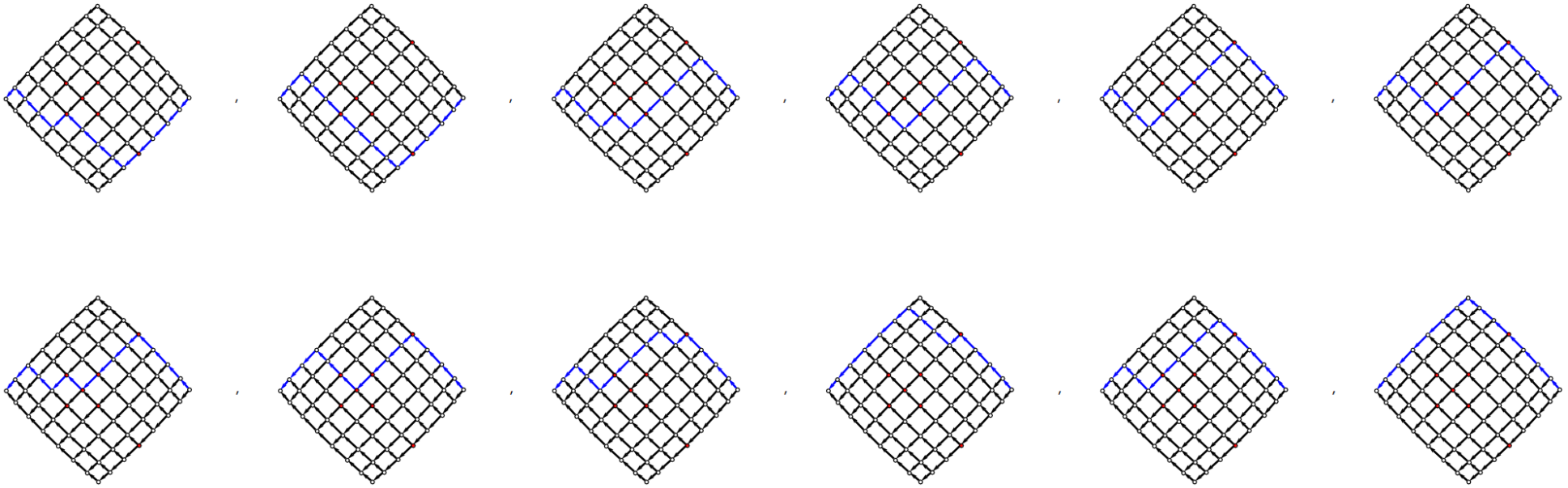
Представители классов эквивалентности (обозн. синим) в **некорректном** алгоритме Спин-блокировки для двух потоков расчетном графе



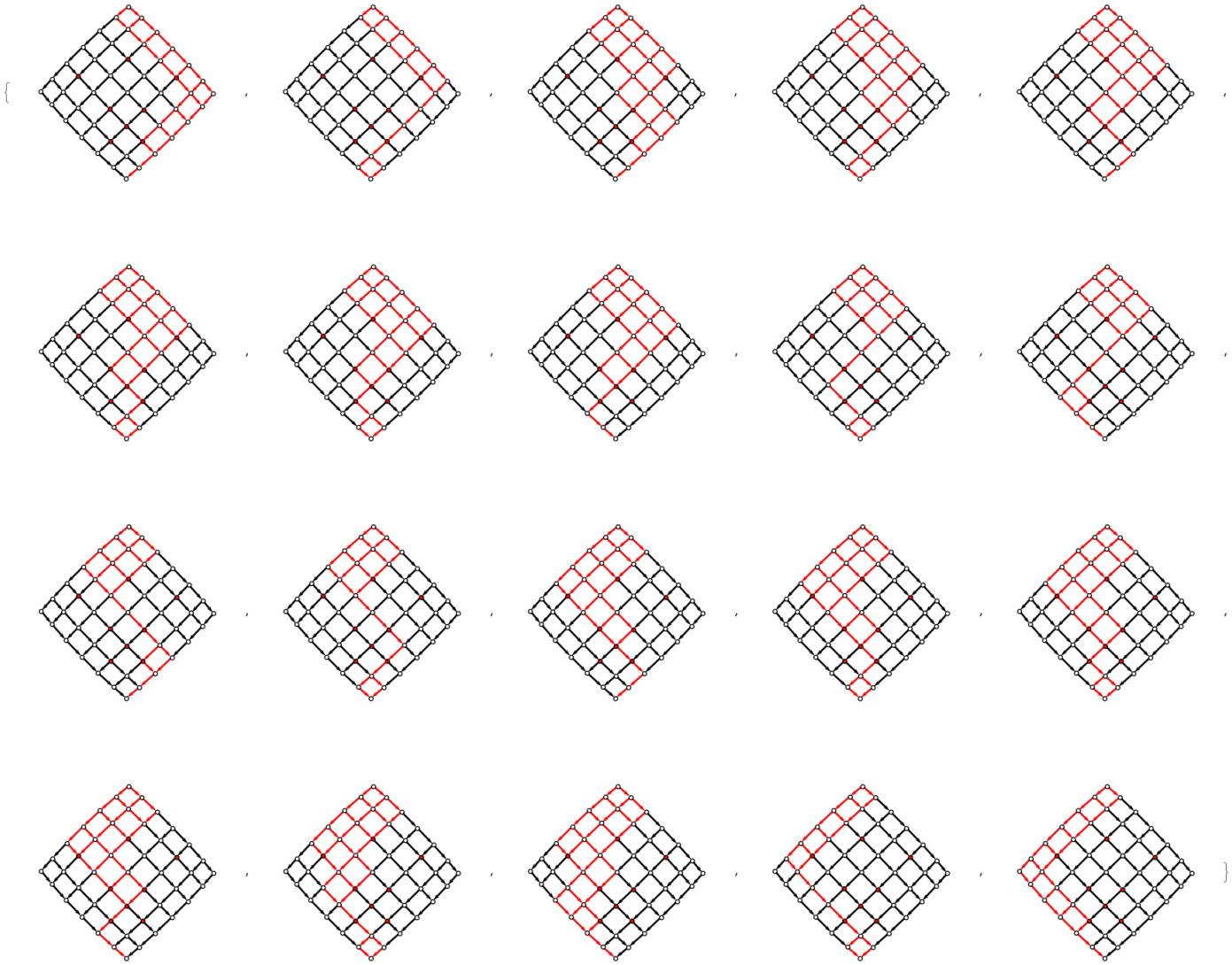
Классы эквивалентности (обозн. красным) в алгоритме Стэк Трейбера для двух потоков на расчетном графе в случае параллельного вызова процедуры POP() в каждом из потоке



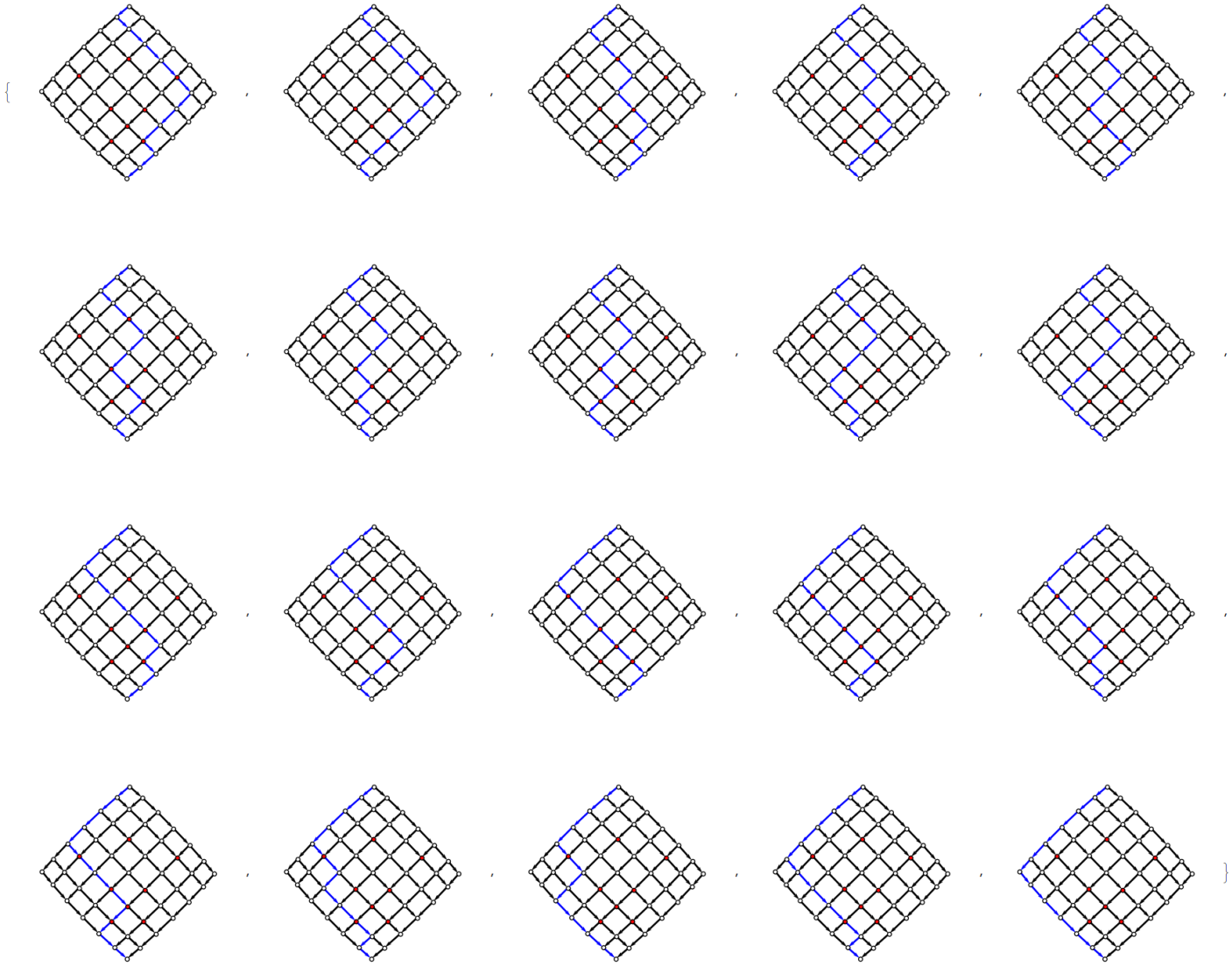
Представители классов эквивалентности (обозн. синим) в алгоритме Стэк Трейбера для двух потоков на расчетном графе в случае параллельного вызова процедуры POP() в каждом из потоке



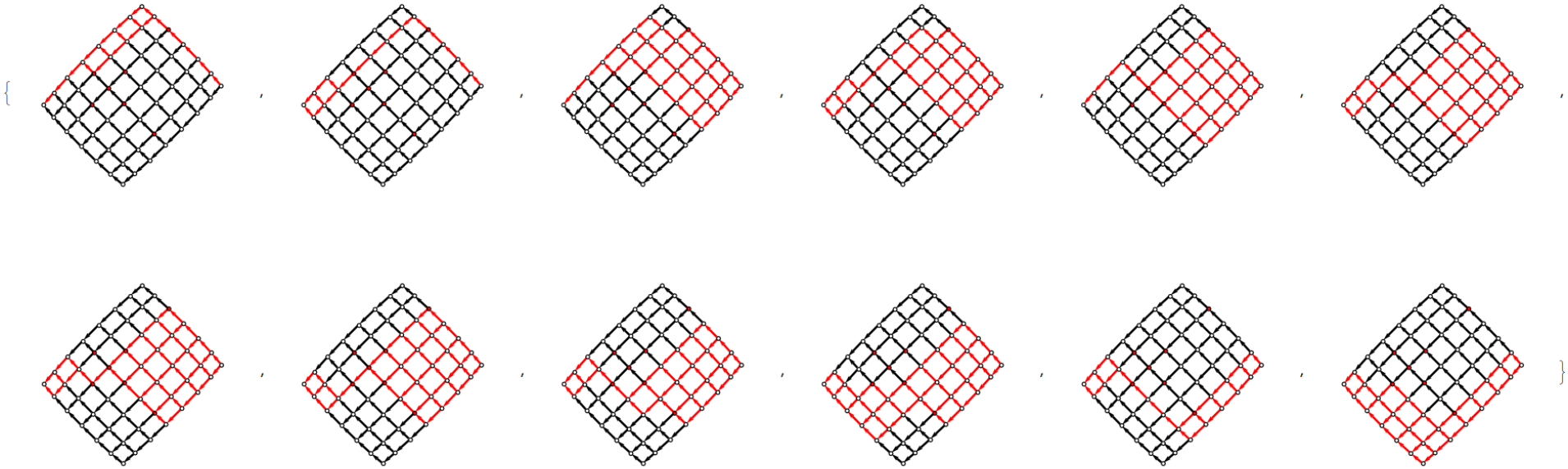
Классы эквивалентности (обозн. красным) в алгоритме Стэк Трейбера для двух потоков на расчетном графе в случае параллельного вызова процедуры PUSH() в каждом из потоке



Представители классов эквивалентности (обозн. синим) в алгоритме Стэк Трейбера для двух потоков на расчетном графе в случае параллельного вызова процедуры PUSH() в каждом из потоке



Классы эквивалентности (обозн. красным) в алгоритме Стэк Трейбера для двух потоков на расчетном графе в случае параллельного вызова процедур POP() и PUSH() в каждом из потоке



Представители классов эквивалентности (обозн. синим) в алгоритме Стэк Трейбера для двух потоков на расчетном графе в случае параллельного вызова процедур POP() и PUSH() в каждом из потоке