Введение Современная тенденция развития микропроцессоров и вычислительного оборудования, применяемого, в частности, при разработке современных деревообрабатывающих комплексов [1÷3] обуславливает создание новых подходов к промышленной разработке программного обеспечения, отвечая все более высоким требованиям потребителя и современным условиям конкуренции. К примеру, повышение количества ядер в микропроцессорах заставляет программных разработчиков использовать многопоточные алгоритмы и применять технологии многопоточной программной разработки не только для высокопроизводительных систем, но уже и для персональных компьютеров и мобильных устройств. 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 связанные 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). 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 [4]. 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. Наиболее полный обзор данных методик и программных технологий, использующих подобные методики, представлен в работе [5]. В рамках диссертационной работы за основу взята математическая модель статического анализа многопоточных программ, описанных в трудах [6÷7]. Общие понятия о модели поиска состояний гонок в многопоточных алгоритмах на основе графа совместного исполнения потоков Существуют различные методы статического анализа [5], которые обладают как преимуществами, так и недостатками, но в

рамках работы интерес представляет метод, который основывается на графах совместного исполнения потоков и расчетного графа [5÷7]. 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: 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. 1. 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. Общая идея подхода к задаче о нахождении состояния гонки на основе графа совместного исполнения потоков и расчетного графа заключается в следующем [5]: 1. Находятся классы эквивалентности полных путей на графе совместного исполнения потоков. 2. Анализируются полные пути на графе совместного исполнения потоков, которые принадлежат разным классам эквивалентности. На основе анализа выбираются пути, для которых возможно состояние гонки. 3. На расчетном графе анализируются только пути, выбранные в п.2, и выводится результат о наличии или отсутствии гонки. Рис. 1 - Пример графа совместного исполнения потоков при k = 4, n = 3 Подобный метод статического анализа также, как и многие другие, чувствителен к условным переходам и циклам в программах. Усложняется не только анализ, но и появляется риск неприменимости анализа. Таким образом, линеаризация представления программы, на котором строится граф совместного исполнения потоков и расчетный граф, позволяет расширить класс задач, к которым можно применить статический анализ. В работе ставится задача выработать обоснованную концепцию получения представления программы, которое содержало бы наименьшее количество условных и цикловых конструкций, а также общее количество анализируемых объектов программы. Предполагается, что результаты работы позволят строить более удобный для статического анализа представление программы - линеаризованный граф потока управления. Анализ промежуточного представления программы Промежуточное представление (Intermediate Representation, IR) - 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 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. Форма статического единственного присваивания (SSA) является одной из самых распространенных форм представления потока данных программы и активно используется в большинстве современных оптимизирующих компиляторов [8]. 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. Исследуемый метод статического анализа многопоточных алгоритмов основывается на графе совместного исполнения потоков, узлы которого - атомарные операции. 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, что в терминах LLVM IR являются инструкции «load» и «store». Таким образом, можно сделать утверждение, что инструкции, оперирующие только с регистрами, не представляют интереса в методе статического анализа, т.к. не влияют на возникновения RC. Исследуя многопоточные алгоритмы нельзя не упомянуть класс атомарных инструкций для осуществления транзакционных операций в память. 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, наблюдаются общая концепция - выполнение несколько операций за единый неделимы период времени: чтение из памяти, сравнение и изменение состояние памяти в зависимости от результатов сравнения. Ярким примером такой инструкции является CAS (Compare and Swap/Set), которая применяется в реализации неблокирующих алгоритмов. Атомарные инструкции типа CAS могут писать в память, таким образом заключаем, что атомарные операции типа CAS нельзя удалять из контекста

программы, т.к. 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 статический анализ. В LLVM IR φ-узел называется φ-инструкцией. Особенность SSA-формы LLVM IR [10] включает в себя то, что все φ-инструкция в узлах графа потока управления располагаются вначале, 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 значение - это результат исполнения каких-либо инструкций. Результат инструкций в SSA располагается на новую 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. 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. Линеаризация графа потока управления на сокращенном промежуточном представлении программы Граф потока управления (Control Flow Graph, CFG) [8÷9] - 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. 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 не значимые и могут быть удалены из CFG. В рамках данной работы был выбран алгоритм удаления пустого узла следующим образом: 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 одно выходящей дугой будут удалены. 3. Пустые узлы, у которых две и более исходящих дуги, остаются в CFG без удаления. В алгоритм п. 3 авторами введен для просты соблюдения корректности и связности графа потока управления. В качестве примера рассмотрим промежуточное представление взаимоисключающего алгоритма Петерсона для 2 потоков, одна из реализации которого на языке Си выглядит следующим образом: /* Algorithm of Peterson */ int volatile turn; 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; } Реализация каждого из потоков симметричная, поэтому достаточно рассмотреть LLVM IR для «thread1_func». Для удобства представим LLVM IR в виде графа потока управления. Рис. 2 - CFG треда в алгоритме Петерсона Уберем в LLVM IR не влияющие на анализ инструкции и построим на сокращенном LLVM IR новый граф потока управления. На рис. 3 видно, что значительно уменьшилось количество инструкций и выявились пустые CFG-узлы. Проведем удаление незначащих CFG-узлов согласно описанному нами алгоритмом. Результат применения алгоритма представлен на рис. 4. Как видно из рис. 4 удалось избавиться от цикла и переходов в CFG представленной программы. Эффект от применения контекстно зависимой линеаризации CFG на реальных задачах будет больше, что сделает применение статического анализа, описанного в работе [5÷7], более доступным. Рис. 3 - CFG треда в алгоритме Петерсона после сокращения IR Рис. 4 - CFG треда в алгоритме Петерсона после контекстно зависимой линеаризации Заключение В работе был проведен анализ инструкций промежуточное представление LLVM IR, который позволил обозначить класс инструкций в LLVM IR, не требующих анализа в методе статического анализа многопоточных алгоритмов на основе графа совместного исполнения потоков. Авторами проведено доказательство, что удаление класса инструкций из контекста программы не нарушает модель статического анализа, повышая при этом его эффективность и надежность. На основе сокращенного контекста программы удалось построить контекстно зависимый линеаризованный граф, который значительно упрощает применения метода статический анализа на основе графов совместного исполнения потоков и расчетного графа. Результаты данной работы расширяют ранее проделанные изыскания [4] и являются существенным дополнением в повышении эффективности и расширении применимости исследуемого статического анализа. Авторами также разработан программный комплекс для внедрения полученных результатов и применения их в анализе промышленных комплексов программ с целью выявления возможных

состояний гонок.