На правах рукописи

Заборовский Никита Владимирович

РАСЧЕТНАЯ МОДЕЛЬ ДЛЯ НАХОЖДЕНИЯ СОСТОЯНИЙ ГОНОК В МНОГОПОТОЧНЫХ АЛГОРИТМАХ

Специальность 05.13.18 – математическое моделирование, численные методы и комплексы программ

АВТОРЕФЕРАТ

диссертации на соискание ученой степени
кандидата физико-математических наук

Москва - 2011

Работа выполнена на кафедре информатики
Московского физико-технического института
(государственного университета)

Научный руководитель:

доктор физико-математических наук, профессор
ТОРМАСОВ Александр Геннадьевич.

Официальные оппоненты:

доктор технических наук
ДРОЗДОВ Александр Юльевич,

кандидат физико-математических наук
СОКОЛОВ Евгений Владимирович

Ведущая организация:

Вычислительный центр им. А. А. Дородницына РАН

Защита состоится 15 декабря 2011 года в 10^{20} часов на заседании диссертационного совета Д 212.156.05 при Московском физико-техническом институте (государственном университете) по адресу: 141700, Московская обл., г. Долгопрудный, Институтский пер., д. 9, ауд. 903 КПМ.

С диссертацией можно ознакомиться в библиотеке Московского физико-технического института (государственного университета).

Автореферат разослан 14 ноября 2011 года.

Ученый секретарь диссертационного
совета Федько О.С.

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Актуальность темы

Многопоточная организация программ достаточно давно применяется программистами. При одноядерной архитектуре процессоров использование нескольких потоков обеспечивало упрощение процесса программирования, работу с блокирующими операциями и т.д. Чтобы эффективно использовать современные аппаратные платформы, нужны совершенно новые принципы и подходы. Под «эффективностью» понимается выигрыш от использования многоядерной платформы по сравнению с одноядерной. Иногда программы, написанные для одноядерной архитектуры, запускают на четырехядерной в надежде получить прирост производительности, а в результате оказывается, что производительность исполнения таких программ даже уменьшается.

Универсальных способов создавать эффективные программы не существует. Подход для каждой - строго индивидуален. Создание новых многопоточных программ даёт возможность подобрать для индивидуальной задачи по-настоящему подходящий и эффективный алгоритм. Как одно из решений можно упомянуть неблокирующие алгоритмы – альтернативу алгоритмам, использующим явные средства синхронизации.

Отсутствие универсальных подходов влечёт за собой также возможные ошибки программистов. Одной из наиболее сложных для обнаружения ошибок является состояние гонки. Состояния гонки (race condition, конкурентного доступа к памяти) – ситуация, когда состояние общей для нескольких потоков ячейки памяти зависит от распределения процессорного времени между потоками. Предугадать это состояние на уровне алгоритма невозможно. Понять аналитически, что возникло состояние гонки, очень сложно, т.к. для этого требуется анализ экспоненциально растущего количества ситуаций. Именно поэтому так важно контролировать программы ещё на стадии описания. Создание средств контроля корректности программ становится всё более необходимым с выпуском каждого нового процессора и написанием каждого нового сложного программного комплекса.

Различают два принципиально разных подхода к анализу многопоточных алгоритмов на предмет наличия состояний гонки: динамический и статический. Динамический подход представляет собой эмпирическую процедуру «прогона» программы при различных входных данных (data-race-test в Google, Intel Thread Checker). Принципиальный недостаток подхода в целом - невозможно проверить сложную систему, поскольку количество тестов экспоненциально зависит от числа входных параметров. Статический подход анализирует архитектуру программы и программный код без его запуска. К статическому анализу относится формальное доказательство корректности кода программы. Для современных программных продуктов сложность формального доказательства значительна, что делает его неприменимым в промышленных масштабах. Кроме того, есть вероятность ложного срабатывания. Один и тот же алгоритм может содержать состояние гонки и не содержать его в зависимости от значений входных параметров.

Цели работы, задачи исследования

Существует несколько подходов к статическому анализу кода неблокирующих алгоритмов в поисках состояния гонки. Однако, большая часть из них носит чисто теоретический характер, подходит только для модельных задач и дает неточные результаты (большой процент ложных срабатываний и не обнаружений гонки, когда она есть, false-negative). Необходим подход к анализу исходных кодов современных прикладных программ, учитывающий их сложность и специфику.

Для этого, во-первых, необходимо разработать математическую модель, представляющую ход исполнения потоков с точки зрения работы с общими разделяемыми переменными. Модель должна быть применима для анализа реальных современных программ и моделировать исполнение современной многопоточной программы.

Во-вторых, на основе этой модели должен быть разработан метод, позволяющий:

- определять состояние гонки, если оно есть, не давая ложных срабатываний в определенном классе задач;
- давать результат за время, приемлемое для использования при тестировании программных продуктов внутри компании-разработчика.

Кроме модели и метода должен быть создан комплекс программ, анализирующий многопоточные алгоритмы с целью выявления состояний гонки и обладающий следующими свойствами:

- возможность анализировать исходный код на языках высокого уровня и выдавать промежуточное представление исходного кода, аналогичное тому, которое получается на стадии компиляции программы;
- реализация разработанного метода и указание мест возникновения гонки и приводящих к ней условий.

Научная новизна

Научная новизна работы заключается в математическом моделировании поведения многопоточных алгоритмов, содержащих вервления и циклы, использующих разделяемую память. Математическая модель, предлагаемая в работе, учитывает начальные и промежуточные значения разделяемых переменных. Для модели разработана методика поиска состояний гонки в программном коде для определенного класса задач.

Хорошо известен ряд динамических и синтаксических анализаторов, обладающих следующими недостатками: тяжеловесность, зависимость от контекста, малая точность. Разработанный метод не зависит от контекста, определяет состояние гонки, если оно есть, и не даёт ложных срабатываний. Что касается тяжеловесности, то для статических анализаторов эта проблема стоит менее остро, чем для динамических. В работе в качестве примера приведён легковесный алгоритм анализа.

Практическая ценность

Созданные модель и метод ориентированы на практическую реализацию. В работе предложены методика анализа кода на присутствие состояния гонки, использующая некое промежуточное представления кода, и техническая реализация получения промежуточного представления из исходного кода. В качестве средства обработки исходного кода может быть использовано любое другое подходящее средство, позволяющее получить специфицированное промежуточное представление.

Положения, выносимые на защиту

1. Математическая модель исполнения многопоточных программ, учитывающая зависимость исполнения от начальных значений разделяемых переменных и описывающая, в том числе, такие конструкции программного кода как циклы, ветвления и атомарные операции.
2. Общий метод анализа многопоточных алгоритмов на предмет наличия гонок, в основе которого лежит анализ значений разделяемых переменных, и критерии корректности алгоритмов в понятиях разработанной модели.
3. Комплекс программ и результаты вычислительных экспериментов для выявления наличия гонок в некоторых многопоточных алгоритмах, в том числе неблокирующих.

Публикации и апробация результатов

По теме диссертации опубликовано 12 работ, в том числе три [1,2,3] – в изданиях, рекомендованных ВАК РФ.

Результаты работы докладывались и получили одобрение специалистов на научных семинарах и конференциях:

- XXXVI и XXVII международные молодежные научные конференции «Гагаринские чтения» (Москва, 2009, 2010 г.),
- 5-ая международная конференция им. П.Л. Чебышева (Ногинск, 2011 г.),
- XVI Международная научная конференция студентов, аспирантов и молодых учёных «ЛОМОНОСОВ- 2009» (Москва, 2009 г.),
- 49-ая и 53-ая научные конференции МФТИ (Москва, 2007, 2009 г).

Структура и объем работы

Диссертация состоит из введения, пяти глав, заключения и списка использованных источников. Работа изложена на 104 страницах, список использованных источников содержит 62 наименования.

СОДЕРЖАНИЕ РАБОТЫ

Во **введении** обосновывается актуальность диссертации, и формулируются ее цели, характеризуются научная новизна и практическая ценность работы.

Первая глава

В первой главе приводится краткий обзор существующих методов анализа исходного кода с точки зрения его корректности, а также реализующих их программных средств. Приводятся примеры статических анализаторов, указываются известные проблемы и возможные пути решений. Рассматриваются такие анализаторы как Viva64, Intel thread checker, data-race-test, KISS, CHESS и другие.

Вторая глава

Во второй главе описывается один из подходов к моделированию многопоточных исполнений алгоритмов, взятый за основу в данной работе.

Основы моделирования многопоточного исполнения

Уже известна идея построения модели взаимного исполнения атомарных инструкций двумя потоками на разделяемой памяти и принцип ее анализа с целью определения состояния гонки. Общая процедура работы с исходным кодом для выявления состояний гонки следующая:

1. Для каждого потока выделяются операции с интересующими нас разделяемыми переменными.
2. Строится граф, определенный ниже, на основании операций с разделяемыми переменными.
3. На графе выделяются определенные пути.
4. Выделенные пути анализируются, например, с помощью метода неопределенных коэффициентов.
5. По состоянию в финальной вершине делается вывод о наличии гонок.

В основе модели лежит граф совместного исполнения потоков G.

$$G = (V, A),$$

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

.

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 вверху, а конечная – внизу, а все ребра имеет направление сверху вниз. Начальная вершина – исходное состояние двух потоков. Начальной вершине сопоставлены значения всех разделяемых переменных после инициализации. Каждое ребро в таком графе – единичная атомарная операция. После совершения потоком очередной операции

(v_j^i, v_j^{i+1}) происходит переход системы из состояния v_j^i в состояние v_j^{i+1}. Конечная вершина – финальное состояние исполнения потоков.

Построение графа совместного исполнения n потоков

Покажем, как строится расчетный граф для двух потоков. Имеем начальную вершину, в которой состояние системы определяется значениями, которыми проинициализированы переменные. Множество ребер A будем обозначать как a_j^i, где i — номер операции первого потока, а j — второго. Тогда каждой вершине можно присвоить два индекса, означающие количество операций, выполненных каждым из потоков, чтобы оказаться в этой вершине. Из каждой вершины v_j^i, считая от начальной, будет исходить два ребра: a_j^{i+1} и a_{j+1}^i до тех пор, пока в одном из потоков не закончатся операции — тогда из вершины будет исходить только одно ребро, соответствующее операции другого потока. Построенный таким образом граф, как уже было сказано, имеет вид ромба (рис. 1). Метод построения обобщается на случай n > 2 потоков: вершины и ребра будут иметь n индексов, а из каждой вершины будет исходить не менее n ребер. В общем случае граф совместного исполнения n потоков будет иметь вид n-мерного ромбовидного графа.

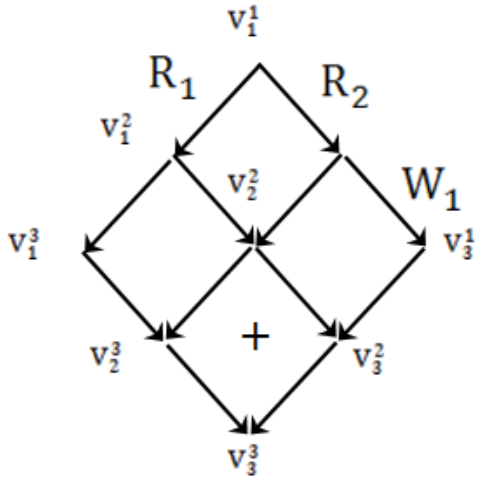


Рис. 1. Граф совместного исполнения двух потоков на разделяемой памяти.

Для модели также определена функция корректности, отражающая субъективное понятие о корректном или некорректном исполнении программы. Одна и та же многопоточная программа может быть корректной с точки зрения одной функции корректности и некорректной – с другой. Отметим, что часто функцию корректности можно переформулировать в терминах частей графа G. К примеру, задача о нахождении неразрешенного состояния гонки может быть сведена к наличию двух различных путей на графе, приводящих к различным значениям одной из разделяемых переменных.

Сильное место подхода – небольшая сложность анализа. Количество вершин на каждом из ребер ромба линейно по количеству операций, и каждая вершина проходится при анализе только один раз. Таким образом, можно оценить, что сложность подхода для двух потоков – $O(k \cdot n)$, где k и n – количество атомарных операций каждого из потоков. Важно заметить, что рассматриваются только операции с разделяемыми переменными.

Далее с построенным графом поступаем следующим образом. Рассматриваем произвольный путь на графе из начальной вершины в конечную. Такой путь описывает взаимную последовательность исполнения атомарных

операций потоками. При проходе обращаем внимание на то, что есть вершины, из которых исходят рёбра с операциями над разными ячейками памяти или только операциями чтения одной и той же разделяемой переменной. В таком случае из вершины можно двигаться по любому из исходящих ребер – на состояние памяти в финальном состоянии это не повлияет. В остальных случаях результат алгоритма будет зависеть от порядка исполнения инструкций - от того, какой из путей из вершины будет выбран. При этом операции на рёбрах назовём не коммутирующими, а соответствующую ячейку графа пометим крестом (рис. 1).

На этом наблюдении основывается построение классов эквивалентности – наборов полных путей на графе G, для которых содержимое ячеек памяти в финальном состоянии одинаково. После нахождения всех классов путей, достаточно взять по одному представителю из каждого класса, чтобы охватить все возможные последовательности смены ячеек памяти.

Третья глава

В третьей главе излагаются сложности и узкие места при статическом анализе программного кода существующими методами. Рассматривается реальный практический код, часто имеющий сложную структуру, содержащий циклы и ветвления, а также атомарные операции.

Проблемы современных систем статического анализа

Для существующих в настоящее время средств настоящей проблемой является ошибочное определение состояния гонки (false-positive) и не нахождение гонки при ее наличии (false-negative). Такие проблемы неминуемо возникают при анализе программного кода из-за наличия циклов, ветвлений, псевдонимов и других конструкций, усложняющих код. Обычная практика для синтаксических анализаторов – использование набора эвристик для определения состояния гонки. Эвристики хорошо работают на элементарных тестах и модельных задачах, но показывают очень плохие результаты на реальных задачах.

В модели исполнения алгоритма в нескольких потоках сделана попытка учесть все возможные варианты исполнения путем введения «не коммутирующих» операций и классификации вариантов совместного исполнения на их основе. Для определенных задач анализ модели исполнения позволяет свести к нулю возможность ошибки false-negative (что очень важно). Однако, класс задач, для которых метод так хорошо работает, довольно узок. В работе предложено расширение модели, позволяющее значительно расширить класс анализируемых программ.

Анализ значений переменных.

Возьмем алгоритм взаимного исключения Петерсона. Критическая секция является таковой за счет проверки значений переменных, ограничивающих исполнения для всех потоков, кроме того, что находится в критической секции. Если статический анализ не включает контроль значений, то критическая секция с точки зрения такого анализа не является критической. В общем случае, задача

контроля переменных равносильна полноценному исполнению программы и вычислению значений всех переменных.

Четвертая глава

В четвертой главе автором предлагается и обосновывается расширенный подход к статическому анализу практических (не модельных) задач, включающий специфическую терминологию и методы.

Предлагаемое расширение модели

Гонка (в терминах графа G) – существование двух разных полных путей, для которых в финальной вершине 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) – в 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:

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

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

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

где D_1 и D_2 – множества бинарных коэффициентов. Анализ таких обыкновенных конструкций кода, как ветвления и циклы, составляет сложность с точки зрения подхода, так как они добавляют на ребро графа совместного исполнения потоков дополнительные переходы. Например, простое ветвление типа if-else добавляет новый переход из одной вершины в другую, а цикл – добавляет переход назад. Несложно увидеть, что конструкция for – композиция перехода назад и условного ветвления.

Автором предлагается дополнить определение исходного графа с тем, чтобы он работал и для ветвлений и циклов, используя, однако, принципы анализа, сформулированные в более ранних работах по теме моделирования одновременного исполнения кода в нескольких потоках.

Ветвления.

Простейшее ветвление, которое можно представить – одиночный оператор if следующего вида: if (condition) {operation;}. Отметим, что одна из идей построения графа совместного исполнения потоков – однозначное отображение один в один инструкций исходного кода на вершины на периметре графа. Именно такое построение обеспечивает линейную по набору инструкций сложность по каждому потоку. В случае простейшего ветвления имеется набор операций,

которые могут выполняться в зависимости от выполнения условия condition. Используя принцип неопределенных коэффициентов, будем считать, что набор операций operation выполняется, если β равно 1 и не выполняется, если β равно 0.

Например, в двух потоках исполняется следующий код при начальном значении x = 0:

```
if (x == 0)
{
   x += 2;
}
```

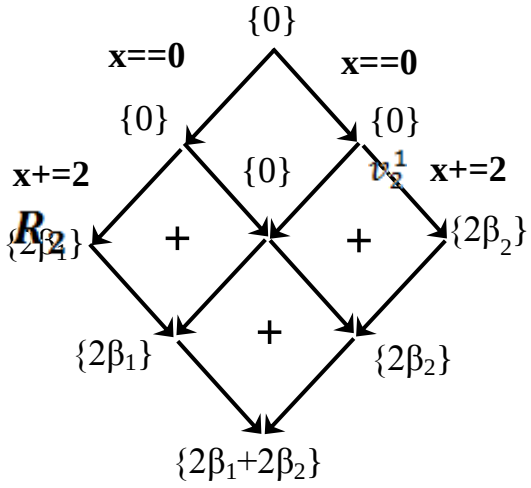


Рис. 2. Граф совместного исполнения двух.

С точки зрения графа он будет иметь вид ребра, на котором значение x меняется на величину 2β. Итак, в финальной вершине графа G значение i-той переменной равно $f_i(\vec{x}_0, D, B)$, где B – множество коэффициентов β. Анализ на предмет наличия гонки проводится с неопределенными коэффициентами B полностью аналогично анализу коэффициентов D. В вершине графа ставятся значения разделяемых переменных, в ребра подписываются операцией, которой ребро соответствует. Для рассмотренного примера получили значение разделяемой переменной (в виде арифметического выражения) $2\cdot\beta_1+2\cdot\beta_2$. Коэффициенты в выражении определяют ветвь, для которой моделируется исполнение. Существуют такие значения неопределенных коэффициентов, при которых разделяемая переменная принимает различные значения, то есть присутствует неразрешенное условие гонки.

Коэффициенты D отвечают за очередность исполнения операций, а B – за ветвления. В общем случае любое ветвление является композицией элементарных ветвлений. Также необходимо заметить, что в случае одиночного if в финальное выражение коэффициент β может войти как множитель, что говорит о вариативности исполнения кода под условием. В случае конструкции if – else в финальное выражение войдут члены с коэффициентами β и $(1-\beta)$, отражающие тот факт, что выполняться будет либо блок кода под if, либо – под else. Корректность описанного представления докажем в следующей теореме.

Теорема 1. Наличие гонки в предлагаемой модели с ветвлениями эквивалентно наличию гонки в задаче (другими словами: предложенное выше представление корректно описывает ветвление в терминах и смысле анализа графа совместного исполнения потоков).

Доказательство. Рассмотрим множество фиксированных значений коэффициентов B в формуле f_i, при этом анализируется линейный участок кода (пути по ветвлениям заданы множеством B). Наличие гонки в предлагаемой модели описывается формулой (3) с учетом коэффициентов β так:

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

В терминах исполнения алгоритма это означает, что есть такие два пути по ветвлениям, которые дают разные значения разделяемой переменной номер i. Таким образом, из состояния гонки в модели следует наличие гонки в алгоритме.

Обратно: путь есть такие два пути по ветвлениям алгоритма, которые приводят к гонке. В соответствии с формулой (3) $\exists i, D_1, D_2 : f_i(\vec{x}_0, D_1) \neq f_i(\vec{x}_0, D_2)$. Оба этих пути описываются с помощью множества коэффициентов B. Таким образом, наличие гонки в модели эквивалентно наличию гонки в алгоритме.

Циклы.

Исходная задача с циклами несколько сложнее, потому что циклы подразумевают переходы назад, а анализ графа совместного исполнения потоков – нисходящий. Таким образом, нет формального описания анализа циклов. Введем формальное правило анализа цикла и докажем его корректность с точки зрения поиска гонок.

Теорема 2. Для описания цикла в анализирующем графе достаточно одного повторения тела цикла.

Доказательство. Доказывать будем от противного. Допустим, необходимо повторение тела цикла два раза для того, чтобы обнаружить наличие гонки, которое бы не обнаружилось при проходе тела цикла один раз. Рассмотрим рисунок 3, приведенный ниже. На рисунке 3 жирной линией обозначен произвольный путь на графе. По предположению два подобных этому пути приводят к гонке, тогда как никакие два из путей в графе с одним повторением тела цикла к гонке не приводят. Рассматриваем отдельно каждый подграф, соответствующий одному повторению тела цикла и учитываем, что в нём гонка отсутствует.

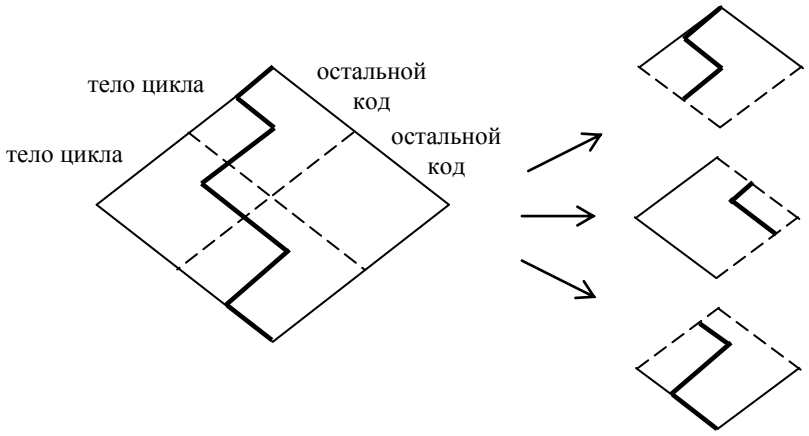


Рис. 3. Представление графа с несколькими повторениями тела цикла в виде подграфов с одним повторением в каждом.

Поскольку в анализе используются только попарные сравнения, а не последовательность использования операций, то из отсутствия гонки в каждой из подзадач следует отсутствие гонки в общей задаче, что противоречит исходному

предположению. Таким образом, предположение о том, что надо повторять граф совместного исполнения с повторением более, чем одного раза тела цикла, неверно. Обобщаем на случай n > 2 повторений тела цикла: обращаем внимание на то, что рассуждения выше не зависели от количества повторений цикла. Отметим, что при замене цикла на графе телом цикла, исчезают обратные переходы, что позволяет воспользоваться разработанным ранее подходом.

Разберём простейший пример применения подхода. Пусть в двух потоках исполняется следующий код:

```
while(true) {
    if (i == 0) {
        i = 1;
        x++;
    }
    else
        break;
}
```

$$c(\vec{x}, i) = \begin{cases} true, x = 1 \\ false, x \neq 1 \end{cases} \quad (4)$$

Исходные значения: x = 0, i = 0. Оставляем тело цикла и приписываем каждому из ветвлений коэффициент. Строим граф совместного исполнения потоков и отмечаем на нём операции чтения и записи. Рассмотрим путь, в котором потоки исполняются по очереди по одной инструкции, и выписываем значения разделяемых переменных с помощью метода неопределенных коэффициентов:

$i = \alpha_1$, $x = \alpha_1 + \alpha_2$,

откуда при $\alpha_1 = \alpha_2 = 1$ получим значение x, при котором функция корректности имеет значение false. Значит, состояние гонки присутствует относительно переменной x.

Общее представление о расчетном графе

Определение. 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. Пусть в графе выделена начальная и конечная вершины, соответствующие начальному состоянию переменных перед исполнением и конечному — после исполнения. Проход по графу моделирует один из вариантов совместного исполнения нескольких потоков путем точного указания следования инструкций одна за другой.

Множество значений разделяемых переменных будем называть состоянием системы. Вектор, компонентами которого являются значения разделяемых переменных, назовем вектором состояния. На расчетном графе определим функцию, соотносящую каждой вершине множество значений разделяемых переменных и функцию, соответствующую операции, производимой над системой на каждом из ребер:

$$S : V \to \vec{x} = (x_1, ..., x_n)$$
$$F : E \to f(\vec{x}) \qquad (5)$$

$\vec{x}$ — состояние системы, а f — вектор-функция, показывающая, как меняется состояние системы после прохода по ребру. Определим на этом графе также функцию условного перехода:

$$\mathrm{M} : (\mathrm{v}_j^i, \mathrm{v}_j^{i+1}) \to P(\vec{x}) = 0, \qquad (6)$$

где P — функция, определенная на множестве значений вектора разделяемых переменных. Это предикат, определяющий возможность передвижения системы из текущей вершины по рассматриваемому ребру, что соответствует условным переходам в программном коде.

Обозначение на рисунках и схемах

Значения всех разделяемых переменных будем обозначать на графе в фигурных скобках: {01}. Если $F(\vec{x})$ меняет значение переменной, то в вершине, куда ребро ведет, новое значение будет подчеркнуто: {0$\underline{1}$}. Каждой атомарной операции соответствует одно ребро (так же, как и в графе совместного исполнения потоков). Отдельно можно выделить специальный вид ребер — ребра условия. Они содержат пустую операцию над переменными: $F \equiv 1$ и функцию-предикат $P(\vec{x})$, определяемую содержанием условия цикла (или содержимым обычного if). Такие ребра появляются, когда в программном коде есть ветвления. Условие прохода по ребру определяет достижимость вершин, куда ребро ведет.

Представление циклов и ветвлений

Линейный программный код (без ветвлений) отображается на ребро графа без каких-либо дополнительных действий. Операции чтения соответствует ребро с $F \equiv 1$ и, возможно, нетривиальным предикатом $P(\vec{x})$, если речь идет, например, о конструкции if (x == 1). При появлении нелинейных участков, таких, как циклы и ветвления, анализ усложняется. Будем использовать для представления ветвлений неопределенные коэффициенты β (аналогично коэффициентам α). Каждой из ветвей приписывается свой коэффициент, и только один из них равен 1, остальные — 0. На выходе имеем выражение с неопределенными коэффициентами, представляющее собой состояние системы на выходе из ветвления. Коэффициенты в выражении определяют ветвь, для которой моделируется исполнение. Обратных ребер в расчетном графе нет. Вместо них у ребер, входящих в тело цикла, появляется зависимость от параметра — номера итерации в цикле. Не всегда зависимость от номера итерации может быть задана явно, однако для реальных задач она, как правило, именно явная. Дополнительное условие налагается на систему условием выхода из цикла. Исходный цикл приобретает форму нескольких однонаправленных ребер, представляющих тело цикла, что, безусловно, облегчает задачу анализа.

Введем класс алгоритмов, для которого и будем выполнять анализ. В рассматриваемый класс входят алгоритмы, в которых все операции с

разделяемыми переменными – линейны, т.е. все выражения содержат переменные в виде $\sum k_i x_i$.В качестве примера можно привести операцию записи в разделяемую переменную «$x_1 = 2x_2 + 4 + x_3$» и операцию чтения разделяемых переменных «$if\,(x_2 == 3 \&\& 2x_3 + 3 == 5)$».

Класс довольно широк: как показывает практика, в него входят (или сводятся) почти все действительные многопоточные алгоритмы.

Конструктивное построение расчетного графа

Построение расчетного графа аналогично построению графа совместного исполнения потоков. Как и в случае расчетного графа, вершины представляют собой состояния системы, отличающиеся друг от друга на одну атомарную операцию чтения/записи, а ребра соответствуют самим атомарным операциям. Отличия от графа совместного исполнения потоков заключаются в данных, сопоставляемых вершинам и ребрам. Каждому ребру соотносят функции (5) и (6). Для определенности будем подразделять ребра на два вида: ребра условия, где *P* — нетривиальна, *f* — тривиальна и ребра операций, где *P* — тривиальна, *f* — нетривиальна. Проход по такому графу из начальной вершины в конечную однозначно определяет последовательность взаимного выполнения инструкций двух потоков. Построение расчетного графа для *n* потоков полностью аналогично.

Расчеты на графе

После того как расчетный граф, состоящий из ребер операций и ребер условий, построен, и начальной вершине соотнесено множество исходных значений разделяемый переменных, можно приступать к расчетам. Каждая из итераций — проход по одному из путей из начальной вершины в конечную. В общем случае число путей достаточно велико — C_n^k. Будем рассматривать только те пути, которые были отобраны с помощью классов эквивалентности на графе совместного исполнения потоков. Таких путей значительно меньше — *O*(*n*·*k*).

<u>Поток 1:</u>
A1: *y* = 1
A2: *z* = 0
A3: while(*x*==1&&*z*==0);
A4: // critical section
A5: *y* = 0

<u>Поток 2:</u>
B1: *x* = 1
B2: *z* = 1
B3: while(*y*==1&&*z*==1);
B4: // critical section
B5: *x* = 0

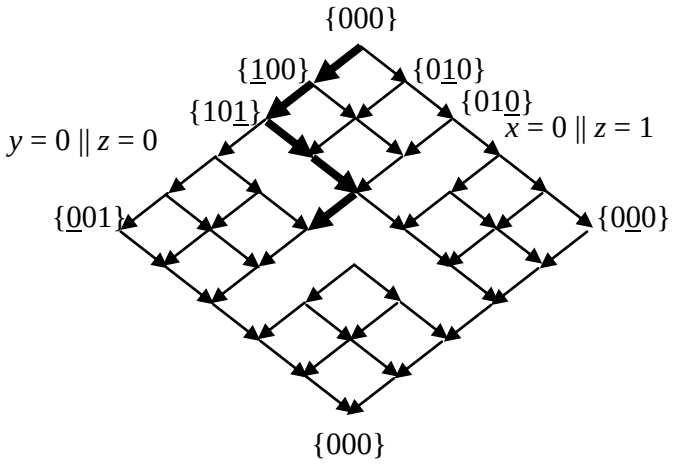


Рис. 4. Расчётный граф для алгоритма Петерсона.

Элементарная часть итерации — проход их текущей вершины по ребру в следующую в соответствии с выбранным путем и затем либо изменение вектора состояния, либо проверка условия достижимости и проход по ребру (или признание ребра недостижимым). Если в графе были ветвления, то ребра, соответствующие ветвям цикла, будут иметь коэффициент β и $(1\text{-}\beta)$. Коэффициенты войдут в выражение для значений переменных в финальной вершине. В зависимости от рода задачи и функции корректности по графу делается вывод о недостижимости критической секции, присутствии состояния гонки для конкретной разделяемой переменной и других подобных условий. Рассмотрим анализ алгоритма Петерсона в качестве иллюстрации применения предлагаемого подхода (рис. 4). Внутренние ребра намеренно не подписаны, но подразумевается, что расчеты производятся и для них тоже.

Рассмотрим путь, выделенный жирным, и вершину *A* на нем. Может ли система пойти из этой вершины по правому ребру? В вершине *F* значение вектора состояния {1 1 0}. Условие попадания из *F* в *A* — выполнение предиката y = 0 || z = 0. Если предикат верен, ребро, входящее в *A*, достижимо. Достижимость ребра *AS* определяется предикатом $x = 0 \,||\, z = 1$. При векторе состояния {1 1 0} значение предиката ложно, поэтому ребро недостижимо.

Доказательство корректности подхода

Лемма. У путей, принадлежащих одному классу эквивалентности, порядок операций с любой разделяемой переменной - одинаков.

Это означает, что если взять всё множество последовательностей операций, соответствующих всем путям одного класса эквивалентности, выбрать любую из разделяемых переменных и посмотреть на порядок следования операций, совершаемых с этой переменной, то окажется, что порядок операций с этой переменной всегда одинаков.

Доказательство. Рассмотрим произвольный граф совместного исполнения потоков и произвольный класс эквивалентности этого графа. На рис. 3 приведен пример такого графа. Знаком «+» обозначены ячейки графа, образованные ребрами с не коммутирующими операциями, жирными линиями – класс эквивалентности. Доказывать будет от противного. Выберем класс эквивалентности, два пути на нем и некоторую разделяемую переменную.

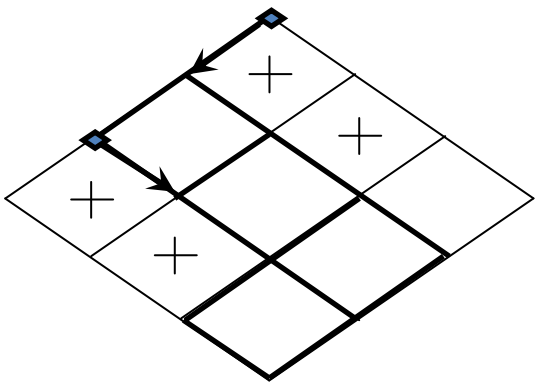


Рис. 5. Положение класса эквивалентности относительно коммутирующих ячеек.

Предположение: допустим, порядок операций с этой переменной для выбранных путей – различный.

Обратим внимание, что в конечной вершине все операции так или иначе будут выполнены, значит, имеет место перестановка операций чтения/записи. Значит, состоялась одна из трех перестановок:

1) R↔R
2) W↔R
3) W↔W

Первый вариант нас не интересует т.к. в обоих потоках происходит считывание значения переменной, и случай для нас неразличим. Рассмотрим перестановки 2 и 3. В них очередность исполнения явно влияет на содержимое разделяемой переменной. Обратим внимание, что местами могут меняться только операции из разных потоков, внутри одного потока порядок зафиксирован («порядок выполнения программы»). Значит, существует ячейка с ребрами W и W/R, каждое из которых входит в состав класса эквивалентности. Но такая ячейка является не коммутирующей, что исключает возможность её принадлежности к одному классу эквивалентности. Предположение не верно, порядок следования операций – одинаковый.

Теорема 3. Для того чтобы сделать вывод о наличии или отсутствии состояния гонки посредством анализа значений разделяемых переменных на введенном классе алгоритмов, *достаточно* рассмотреть представителей классов эквивалентности из графа совместного исполнения потоков.

Доказательство. При наличии единственной разделяемой переменной утверждение теоремы прямо следует из доказанной леммы. В ситуации, когда есть несколько разделяемых переменных, утверждение теоремы не очевидно и требует дополнительного доказательства. Далее предлагается доказательство от противного.

Пусть существуют два пути, принадлежащие одному классу эквивалентности, приводящие к различным значениям разделяемых переменных. Допущение сформулировано, используя отрицание формулировки теоремы и исходя из приведенного выше определения понятия гонки. Из доказанной леммы и различия значений переменных после исполнения следует, что существует некая строчка исходного кода или полученная по ней инструкция платформы, которые приводят к разным значениям одной из переменных. Это может быть либо явное присваивание переменной нового значения, например:

shared_x = func(**shared_y**, **shared_z**);

либо логическое ветвление кода, приводящее к той же записи в переменную:

```
if (shared_y == 0)
        shared_x = 1;
else
        shared_x = 0;
```

В любом из случаев на значение разделяемой переменной влияют значения остальных разделяемых переменных.

Вернемся к сформулированному выше допущению: предполагаем, что проход по разным путям в графе приводит к различным значениям переменных после исполнения программы. Отметим, что исходные значения разделяемых переменных не зависят от пути на графе совместного исполнения потоков. К интересующему нас моменту ветвления или записи значения переменных одинаковы для обоих потоков. Это следует из доказанной ранее леммы для каждой из переменных и из того факта, что до этого никаких ветвлений и записи

не было. Если ранее были ветвления или запись, то повторим доказательство для этого предыдущего ветвления или записи.

Но если значения всех переменных к моменту ветвления или записи одинаковы, то и результат соответствующих операций сравнения и записи будет одинаковым, следовательно, предположение о том, что для разных путей на графе в результате исполнения получаются разные значения, неверно.

Обоснованность использования представителей класса эквивалентности в расчетном графе.

Воспользовавшись доказанной теоремой 3, получаем одинаковую очередность операций с разделяемыми переменными в рамках каждого класса эквивалентности. Можно рассматривать это так, что каждый из потоков получает управление строго в определенный момент, когда это повлияет на дальнейшее исполнение программы, и до момента следующей развилки исполняется «в одиночестве» и так до момента выхода на не коммутирующие ячейки. Используя этот факт, сложность анализа программного кода можно существенно сократить, взяв по одному пути из каждого класса эквивалентности. С учетом того, что классы эквивалентности покрывают все возможные пути, заключаем, что анализ представителей классов в расчетном графе дает ответ на вопрос о наличии гонок в алгоритме.

Пятая глава

В заключительной пятой главе приведены аспекты практической реализации комплекса программ для анализа многопоточных алгоритмов, а также примеры работы прототипа для различных алгоритмов.

Комплекс программ для полуавтоматического анализа

Автором был разработан комплекс программ полуавтоматического анализа многопоточных алгоритмов на предмет наличия состояний гонки. Программный комплекс реализующий принципы, заложенные в расширенную модель. В качестве исходного материала анализа выступает текст программы, предполагаемый для запуска в многопоточном режиме. Кроме того, для соответствующего кода должен быть определен «регламент»: сколько потоков, какие функции доступны из каких потоков. Чтобы использовать разработанный метод, код должен быть преобразован в специальное промежуточное представление. Его можно сравнить объектными файлами, получаемыми после компиляции программы: в нем «зашита» информация о переменных, их область видимости, очередность исполнения. Получить такую информацию позволяет Phoenix Toolkit, этот инструмент дает доступ ко всем сущностям программы в виде объектов. При помощи Phoenix Toolkit можно получить формализованное промежуточное представление программы из исходного кода.

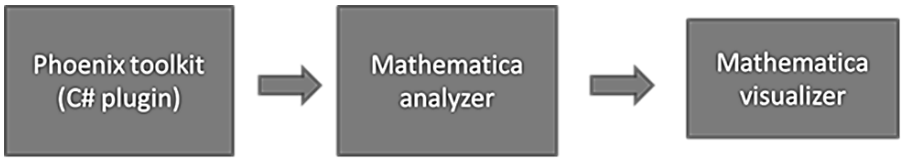


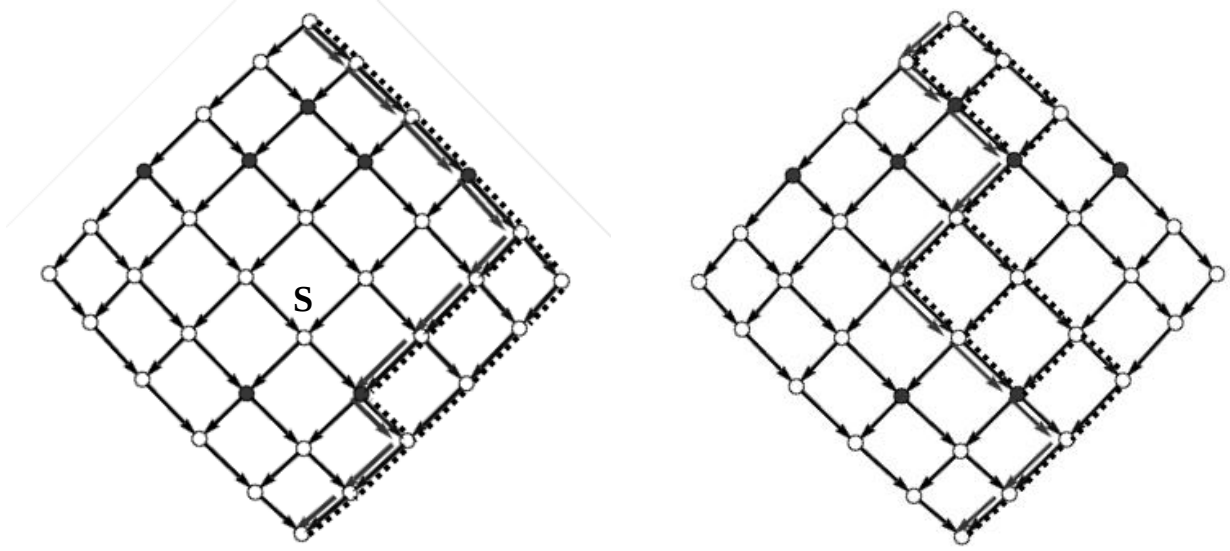
Рис. 6. Комплекс программ для статического поиска состояний гонки в многопоточных алгоритмах.

Результатами обработки в Phoenix Toolkit является промежуточное представление программы. Важно заметить, что представление для языка C++ формируется с помощью Phoenix Toolkit, но оно может быть получено и с помощью другого программного инструментария. Могут быть проанализированы другие языки – лишь бы нашелся нужный разборщик, работающий подобно компилятору и позволяющий использовать результаты компоновки. Следующий шаг анализа – подача промежуточного представления на вход анализатору, прототип кода которого написан в Mathematica. С помощью разработанных методов строится граф совместного исполнения потоков, выделяются классы эквивалентности, представители классов, затем – расчетный граф и анализ путей на нем. Визуализация результатов так же часто выполнена в программе Mathematica: отображаются возможные способы исполнения и выводятся значения разделяемых переменных после исполнения.

Модельные эксперименты.

1. Алгоритм Петерсона.

Текст алгоритма Петерсона приведен на рис. 4. Вершина графа совместного исполнения потоков, соответствующая ситуации, когда оба потока оказываются в критической секции, обозначена буквой S на рис. 7.



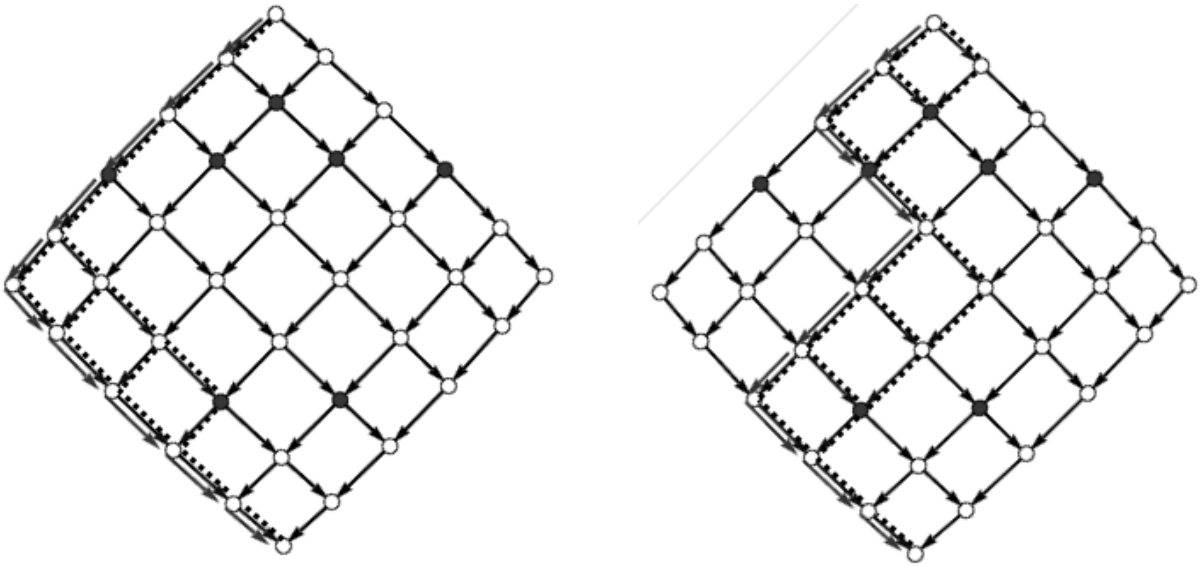


Рис. 7. Допустимые классы эквивалентности и их представители для алгоритма Петерсона.

Пунктиром обозначены классы эквивалентности, сдвоенными стрелками – представитель класса. Множество допустимых классов эквивалентности не содержит вершины S, поэтому алгоритм действительно реализует критическую секцию. Сложность алгоритма по числу операций – $O(N^3)$. Количество классов эквивалентности – 16, возможных вариантов исполнения – 4. Алгоритм корректен.

2. Стек Трейбера.

Стек Трейбера – неблокирующая реализация стека. Алгоритм и классы эквивалентности представлены на рис. 8. Имеем несколько серий решений:

$$\begin{cases} \{0, a\} = \{a, b\} \\ \boldsymbol{top} = \{0, a\} \end{cases}$$

$$\boldsymbol{top} = \{0, 0\}$$

$$\begin{cases} \{0,0\} = \{a, b\} \\ \boldsymbol{top} = \{b, 0\} \end{cases}$$

Серии приведены для операции **push** в одном потоке и **pop** – в другом. Система получается в финальной вершине. Каждая серия соответствует одному из трех возможных вариантов взаимного исполнения потоков. Параметрами задано исходное состояние верхней ячейки стека. Решение каждой из систем приводит к одному и тому же значению ячейки стека.

```
PUSH(value) {
  node = new node();
  node->value=value;
  node->next=NULL;
  repeat {
     top = GlobalTop;
     node->next=top;
  } until CAS(&GlobalTop,
              top, node)
}

POP() {
  repeat {
     top = GlobalTop;
     if (top==NULL)
        return;
  } until CAS(&GlobalTop,
              top, top->next)
}
```

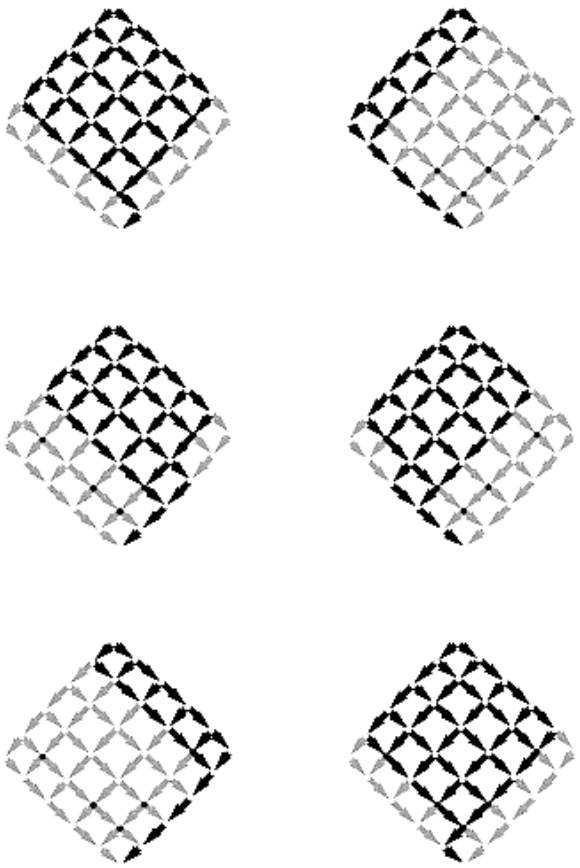


Рис. 8. Стек Трейбера: классы эквивалентности и алгоритм.

Сложность анализа алгоритма стека Трейбера равна $O(N^3+N^2M^2)$, где N – количество операций, M – количество условных конструкций в алгоритме. Количество классов эквивалентности – 6, возможных вариантов исполнения – 3. Алгоритм корректен.

3. Неблокирующая хеш-таблица.

Реализация неблокирующей хеш-таблицы с открытой адресацией описана в статье «Almost Wait-free Resizable Hashtables» авторами Gao, Groote, Hesselink. Далее приведен текст части кода добавления элемента в таблицу.

Будем анализировать ситуацию, когда два потока одновременно пытаются добавить элемент в хеш-таблицу. С точки зрения реализации хеш-таблицы, корректный исход такого добавления – добавление обоих элементов в таблицу, а некорректный – перезапись элемента в таблице.

```
proc insert (v:Value \ {null}): Bool =
  local rEValue ;kl;nNat ;
  h:pointer to Hashtable ;
  suc :Bool a:Address
27: a:=ADR(v);h=H[index];
28: if h.occ >h.bound then newTable ();
30: h:=H[index]; fi
31: n0;l:=h:size ; suc :=false ;
  repeat
32: k:=key (a;l;n);
33: <r: = h.tablef[k];
  {if a=ADR(r) then (iS) fi} >
35a: if oldp(r) then refresh ();
36: h:=H[index];
37: n:=0; l:=h.size ; elseif r=null then
35b: <if h.table[k]=null then suc :=true ;
   h.table[k]=v; {(iS)} fi>
   else n++ ;fi
   until suc || a=ADR(r);
41: if suc then <h.occ++ ;> if
42:return suc
```

```
proc getAccess ()=
loop
59: index :=currInd;
60: <prot[index]++ ;>
61: if index =currInd then
62: <busy[index]++ ;>
63: if index =currInd then return
else releaseAccess (index);fi
65: else <prot[index]-- ;> fi
end end .
proc releaseAccess (i:1. . 2P)=
local h:pointer to Hashtable ;
67: h :=H[i];
68: <busy[i]-- ;>
69: if h ≠ nil ^ busy[i]=0 then
70: <if H[i]=h then H[i]:=nil;>
71: deAllocate (h);fi fi
72: <prot[i]-- ;>
end .
```

Построим классы эквивалентности и их представителей. Представители классов эквивалентности отображены на рисунке 9. Две точки на графе совместного исполнения, соответствующие некорректному поведению алгоритма, обозначены на рисунке как A и B. Прохождение пути на графе через эти точки означает, что добавляемый элемент перезапишет уже существующий или одновременное добавление элементов в одну ячейку таблицы. По представителям классов эквивалентности видно, что ни один путь через эти точки не проходит.

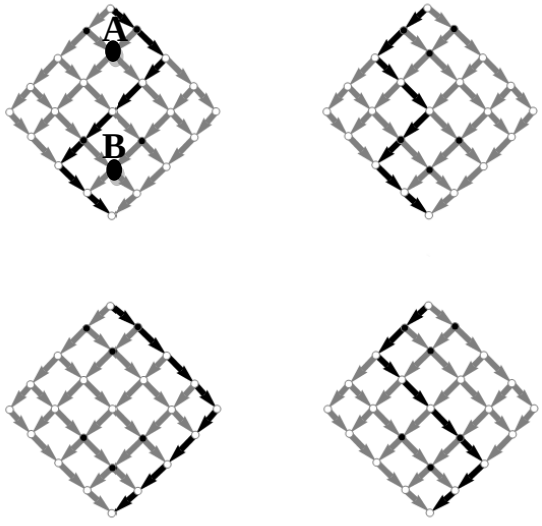


Рис. 9. Представители классов эквивалентности для хеш-таблицы.

По результатам одновременного исполнения операции вставки в двух нитях в таблицу добавляются два элемента. Вставка корректна.

Авторы оценили сложность формального доказательства корректности всех операций с хеш-таблицей в два человеко-года. Сложность анализа работы алгоритма на операциях добавления элемента с помощью разработанного подхода по числу операций – $O(N^3+N^2M^2)$, где N – количество операций, M – количество условных конструкций в алгоритме. Количество классов эквивалентности – 26, возможных вариантов исполнения – 4.

В **заключении** приведены основные результаты работы и намечены направления дальнейшей работы.

ОСНОВНЫЕ РЕЗУЛЬТАТЫ РАБОТЫ

1. Разработана математическая модель исполнения многопоточных программ, учитывающая зависимость исполнения от начальных значений разделяемых переменных и позволяющая анализировать многопоточные алгоритмы, содержащие циклы, ветвления и атомарные операции.
2. Предложен метод полуавтоматического поиска состояний гонки (конкурентного доступа к памяти) в классе алгоритмов, содержащих только линейные операции с разделяемыми переменными, а также сформулированы критерии корректности для нескольких алгоритмов.
3. Создан комплекс программ для исследования многопоточных алгоритмов, содержащих циклы, ветвления и атомарные операции с целью выявления состояний гонки.
4. На основании результатов серии вычислительных экспериментов с использованием предложенной математической модели, методов и комплекса программ получены численные оценки сложности для ряда алгоритмов.

СПИСОК ПУБЛИКАЦИЙ ПО ТЕМЕ ДИССЕРТАЦИИ

1. *Заборовский Н.В., Тормасов А.Г.* Моделирование многопоточного исполнения программы и метод статического анализа кода на предмет состояний гонки // Прикладная информатика – М.: 2011, №4(34). - С. 105-110.
2. *Заборовский Н.В., Тормасов А.Г.* Расчетный подход к статическому анализу программного кода на предмет наличия состояний гонки // Программная инженерия – М.: 2011, №7. – С.46-54.
3. *Заборовский Н.В., Тормасов А.Г.* Статическое обнаружение гонок в коде, содержащем ветвления и циклы // Прикладная информатика – М.: 2011, №6(36). - С. 75-87.
4. *Заборовский Н.В., Тормасов А.Г.* Подход к нахождению состояния гонки в практических многопоточных программах // Сборник научных трудов

«Математические модели и задачи управления» - М.: МФТИ, 2011. – С. 198-207.
5. *Заборовский Н.В.* Подход к моделированию вычислений в нескольких потоках // Сборник тезисов 5-й международной конференции им. П.Л. Чебышева – Обнинск: МИФИ, 2011. – С. 113.
6. *Заборовский Н.В., Кудрин М.Ю., Прокопенко А.С., Тормасов А.Г.* Автоматизация алгоритма поиска логических гонок в программах на разделяемой памяти // XXXVI Гагаринские чтения. Международная молодежная научная конференция. Научные труды. Т. 4. – Москва: МАТИ, 2010. – С. 91-92.
7. *Заборовский Н.В.* Моделирование функциональности системного реестра ОС Windows для использования в виртуализованных системах // Сборник тезисов XXXV Гагаринских чтений. – М. МАТИ: 2009. – С. 126-127.
8. *Заборовский Н.В.* Сборник тезисов XVI Международной научной конференции студентов, аспирантов и молодых учёных «Ломоносов-2009», секция «Вычислительная математика и кибернетика». - М.:МГУ, 2009 - С. 31-32.
9. *Заборовский Н.В., Шейнин В.В.* Эффективная работа с данными при решении задач коллаборативной фильтрации // Сборник трудов VI Всероссийской конференции студентов, аспирантов и молодых ученых «Технологии Microsoft в теории и практике программирования» - М.:МГУ, 2009. – С. 13-15.
10. Заборовский Н.В., Петров В.Н., Разработка и реализация технологии шаблонного хранения данных в реестре Windows // Современные проблемы фундаментальных и прикладных наук. Часть VII. Прикладная математика и экономика: Труды XLIX научной конференции. /Моск. физ. – техн. ин-т. – М. – Долгопрудный, 2006. – С. 55.
11. *Заборовский Н.В.* Поиск состояния гонки в многопоточных алгоритмах // Современные проблемы фундаментальных и прикладных наук. Часть VII. Прикладная математика и экономика: Труды LIII научной конференции. /Моск. физ. – техн. ин-т. – М. – Долгопрудный, 2010. – С. 164.
12. *Заборовский Н.В.* Моделирование реестра ОС Windows для использования в виртуализованных системах // Сборник научных трудов «Модели и методы обработки информации» - М.:МФТИ, 2009. - С. 194-197.

В работах с соавторами **лично соискателем выполнено следующее**: [1,2,3,4] – разработка математической модели исполнения многопоточных программ и метода полуавтоматического поиска состояний гонки, создание комплекса программ и анализ ряда известных алгоритмов с целью выявления состояний гонки и получение для этих алгоритмов численных оценок сложности, [6,9,10] – описание поведения потоков в многопоточных приложениях при использовании средств синхронизации.

Заборовский Никита Владимирович

РАСЧЕТНАЯ МОДЕЛЬ ДЛЯ НАХОЖДЕНИЯ СОСТОЯНИЙ ГОНОК В МНОГОПОТОЧНЫХ АЛГОРИТМАХ

АВТОРЕФЕРАТ

Подписано в печать 01.11.2011 Формат 60 × 84 1/16. Усл. печ. л. 1,0.
Тираж 70 экз. Заказ № 770.
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Московский физико-технический институт (государственный университет)»
Отдел оперативной полиграфии «Физтех-полиграф»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9