ИСП Институт Системного Программирования
Российской Академии наук

Труды Института Системного Программирования

Том 23

Москва 2012

ИСП **Учреждение Российской академии наук**
Институт системного программирования РАН

Труды Института Системного Программирования

Том 23

Под редакцией
академика РАН В.П. Иванникова

Москва 2012

УДК51

Труды Института системного программирования: Том 23.
/Под ред. В.П. Иванникова/ – М.: ИСП РАН, 2012. – 477 с.

В двадцать втором томе Трудов Института системного программирования публикуются статьи, содержащие результаты работ, проводимых в Институте в 2011-2012 годах.

ISBN 978-0-543-57631-6

© Институт Системного Программирования РАН, 2012

Содержание

П р е д и с л о в и е

В 22-м томе Трудов ИСП РАН публикуются статьи сотрудников Института (главным образом, молодых специалистов), в которых описываются результаты научных исследований, полученные в конце 2011-го – начале 2012 гг.

Том открывается двумя статьями А.И. Аветисяна. Первая из них называется «Двухэтапная компиляция для оптимизации и развертывания программ на языках общего назначения» и содержит описание метода двухэтапной компиляции программ на языках общего назначения (Си/Си++), основанного на компиляторной системе LLVM и позволяющего проводить оптимизации программ с учетом профиля пользователя и особенностей его целевой машины, а также организовывать развертывание программ в облачном хранилище с дополнительной оптимизацией и поиском дефектов программ.

Во второй статье А.И. Аветисяна – «Планирование команд и конвейеризация циклов на современных архитектурах» – предлагается метод планирования команд и конвейеризации циклов, основанный на расширяемой архитектуре планировщика, содержащей компоненты выявления и использования параллелизма на уровне команд.

В статье Романа Жуйкова, Дмитрия Мельника и Рубена Бучацкого «Программная конвейеризация циклов на платформе ARM» описывается адаптация для архитектуры ARM алгоритма поворотного модульного планирования, используемого в компиляторе GCC. Для обеспечения поддержки архитектуры ARM были в этот алгоритм были внесены значительные изменения, которые позволили ускорить ряд тестовых приложений на 3-4%.

Статья Романа Жуйкова, Дмитрия Плотникова и Мамикона Варданяна «Автоматическая настройка оптимизационных преобразований компилятора GCC для платформы ARM» посвящена описанию системы для автоматической настройки параметров компиляции, предназначенной для использования на встраиваемых платформах. С использованием этой системы, в частности, были выявлены недочеты в работе компилятора GCC, приводящие к генерации неоптимального кода для платформы ARM.

Кирилл Батузов представил статью «Задача локального распределения регистров во время динамической двоичной трансляции», в которой рассматриваются возможные улучшения алгоритма локального распределения регистров в QEMU, эмуляторе аппаратного обеспечения, основанного на использовании динамической двоичной трансляции. Практические результаты

показали, что дальнейшее улучшение алгоритма возможно только при изменении имеющихся ограничений, накладываемых на этот алгоритм.

К. Батузов, П. Довгалюк, В. Кошелев и В. Падарян в своей статье «Два способа организации механизма полносистемного детерминированного воспроизведения в симуляторе QEMU» предлагают два подхода к реализации полносистемного детерминированного воспроизведения в симуляторе QEMU, которые отличаются тем, какие компоненты виртуальной машины включаются в воспроизведение. Оцениваются достоинства и недостатки этих подходов, оценивается уровень накладных расходов при записи журнала невоспроизводимых событий..

В статье А.И. Аветисяна и А.И. Гетьмана «Восстановление структуры бинарных данных по трассам программ» рассматривается задача восстановления форматов бинарных данных. Предложены подходы к решению данной задачи, а также описывается реализация этих подходов в рамках разрабатываемой системы восстановления форматов.

Статью «Инкрементальный анализ исходного кода на языках C/C++» написали В.О. Савицкий и Д.В. Сидоров. В статье описывается метод построения статического анализатора кода, позволяющий существенно сократить время повторного поиска дефектов для языков C\C++. Используется свойство малого отношения количества лексем из исходного файла к количеству лексем из заголовочных файлов. Метод реализован в программном продукте для среды разработки MS Visual Studio.

Статья А.Ю.Тихонова и А.И. Аветисяна «Комбинированный (статический и динамический) анализ бинарного кода» посвящена обсуждению проблем анализа программ в бинарном коде для распознавания алгоритмов составляющих их функций, выяснения особенностей реализации алгоритмов, обнаружения недокументированных возможностей. В ряде случаев статический анализ может не дать результатов. Предлагается использовать в таких случаях в дополнение к статическому анализу динамический анализ (анализ трасс программы). Описывается разработанная и реализованная авторами среда динамического анализа бинарного кода TrEx и ее интеграция с известной средой статического анализа Ida Pro.

И.Н.Ледовских и М.Г.Бакулин являются авторами статьи «Подход к восстановлению потока управления запутанной программы». В статье описывается подход к восстановлению потока управления программы, запутанного комбинацией различных преобразований – от вставки недостижимого кода с помощью непрозрачных предикатов до виртуализации. Приводятся результаты тестирования прототипной реализации данного

подхода на модельном примере, запутанном с применением различных инструментальных средств защиты.

В статье В.Н. Игнатьева «Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования» отмечается, что для достижения переносимости, надёжности и безопасности программ на C и C++ могут вводиться дополнительные ограничения на язык и стиль программирования. Предлагается новый метод формализации и классификация таких ограничений, описывается система их автоматической проверки, основанная на применении статического анализа, а также способ интеграции с распространёнными системами сборки проектов.

А.О. Кудрявцев, В.К. Кошелев и А.И. Аветисян в статье «Перспективы виртуализации высокопроизводительных систем архитектуры x64» исследуют перспективы применения технологий виртуализации в области высокопроизводительных вычислений на платформе x64. Рассматриваются основные причины падения производительности при запуске параллельных программ в виртуальной среде. Подробно рассматриваются системы виртуализации KVM/QEMU и Palacios. В качестве тестовых пакетов используются HPC Challenge и NAS Parallel Benchmarks.

Статья Андрея Белеванцев, Алексея Меркулова и Владимира Платонова «Использование стандарта OpenCL для программирования ПЛИС» посвящена использованию стандарта OpenCL для облегчения программирования логических интегральных схем (ПЛИС), применяемых в качестве акселератора в гетерогенной вычислительной системе. Описывается схема реализации подмножества стандарта, обеспечивающая обмен памятью и управление выполнением задач на ПЛИС в предположении, что ПЛИС связана с центральным процессором через шину PCI-express.

Статья «Оптимизация расчётов в пакете OpenFOAM на GPU» представлена А.В. Монаковым. В статье рассматривается задача повышения скорости расчётов в пакете OpenFOAM за счёт переноса части вычислений на графические акселераторы (GPU). Приводится краткий обзор пакета и анализ переноса на GPU метода сопряжённых градиентов. Описаны несколько оптимизаций, часть из которых специфична для рассматриваемой реализации, а часть применима не только для GPU. Приводятся первичные результаты тестирования производительности.

Целью статьи Игорять Бурдонова и Александра Косачева «Финальные модели спецификации» является выделение подмножества трасс спецификации, достаточного для генерации полного набора тестов. Такое подмножество авторы назвали финальной трассовой моделью спецификации. Традиционная

LTS-модель удобна тем, что является способом конечного представления регулярных множеств трасс. Для представления финальной трассовой модели в работе предлагается разновидность LTS, названная финальной RTS (Refusal Transition System). Такая модель обладает целым рядом свойств, полезных для генерации тестов.

В статье С. П. Вартанова и Д.В. Сидорова «Оптимизация задачи проверки выполнимости булевских ограничений при помощи кэширования промежуточных результатов» предложена оптимизация алгоритма проверки выполнимости булевых формул DPLL (Davis — Putnam — Logemann — Loveland) с помощью кэширования промежуточных результатов при решении задачи нахождения входных данных для неинтерактивных программ. Дополнительная информация для оптимизации алгоритма запоминается на одном из предыдущих запусков алгоритма. Возможность подобного рода модификации алгоритма основана на особенности последовательностей проверяемых формул.

Мандрыкин М.У., Мутилин В.С., Новиков Е.М. и Хорошилов А.В. – авторы статьи «Обзор инструментов статической верификации Си программ в применении к драйверам устройств операционной системы Linux». В работе анализируются методы, используемые в современных инструментах статической верификации Си программ. Представлен обзор текущих возможностей инструментов в части поддержки конструкций языка Си, масштабируемости, видов проверяемых свойств и достоверности результатов анализа. Рассмотрены особенности применения инструментов статической верификации для анализа исходного кода драйверов устройств операционных систем и описаны существующие системы верификации драйверов, построенные на основе таких инструментов.

Статья В.С. Мутилина и М.У. Мандрыкина «Интерполяция формул с кванторами в CSIsat на основе инстанцирования» посвящена реализации на основе инструмента CSIsat интерполяции Крейга для формул с кванторами, заданных в рамках комбинации теорий вещественной линейной арифметики и неинтерпретируемых функций. Предложен способ реализации интерполирования таких формул с помощью инстанцирования подкванторных выражений с использованием внешнего SMT-решателя CVC3. Описываются изменения, которые были внесены в интерполятор и решатель для реализации поддержки кванторной интерполяции. Приводятся результаты тестирования модифицированного интерполятора как на задачах, полученных из набора SMTLIB, так и на тестах, специально сгенерированных для этих целей.

Статью «Анализ типовых ошибок в драйверах операционной системы Linux» представили В.С. Мутилин, Е.М. Новиков и А.В. Хорошилов. В статье предлагается методика выявления и классификации типовых ошибок и

соответствующих им правил на основе изменений, вносимых в драйверы операционной системы Linux. Приводятся результаты применения данной методики, обсуждаются полученная классификация и распределение типовых ошибок по классам.

В статье Андрея Третьякова «Автоматизация построения расписаний для периодических систем реального времени» рассматривается проблема построения расписаний для систем, имеющих жесткие ограничения по времени работы. Эта проблема является одной из основных при проектировании систем авионики. В работе проанализированы существующие алгоритмы, выделены их недостатки и предложено собственное решение, удовлетворяющее необходимым требованиям. Описываются разработанные инструменты для визуализации, редактирования и валидации создаваемых расписаний.

А.Г. Гомзин и А.В. Коршунов написали статью «Системы рекомендаций: обзор современных подходов». Статья представляет собой обзор основных алгоритмов, используемых в системах рекомендаций. Рассмотрены методы коллаборативной фильтрации, методы, анализирующие содержимое объектов, и методы, использующие базы знаний. Все рассматриваемые методы имеют свои недостатки. В статье предложены способы комбинирования методов построения рекомендаций, позволяющие избавиться от этих недостатков.

Статья М.М. Берновского и Н.Н. Кузюрина «Случайные графы, модели и генераторы безмасштабных графов» посвящена обсуждению моделей случайных графов, описывающих реальные сети, которые возникают в различных областях: биологии, компьтерных науках, инженерии, социологии. Особое внимание уделено моделям, описывающим социальные сети.

В статье В.А.Захарова и Т.А. Новиковой «Полиномиальный по времени алгоритм проверки логико-термальной эквивалентности программ» описывается новая модификация алгоритма проверки логико-термальной эквивалентности программ, основанной на операции вычисления точной нижней грани в решетке конечных подстановок. Показано, что трудоемкость предложенного алгоритма оценивается величиной, где - размер анализируемых программ.

Статью «Об одной задаче Коффмана-Шора, связанной с упаковкой прямоугольников в полосу» представил М.А. Трушников. В статье предложен новый онлайновый алгоритм упаковки прямоугольников в полосу, существенно превосходящий известный алгоритм Коффмана-Шора по качеству получаемой упаковки.

В статье А. В. Шокурова «Сравнение сложностей задач нахождения базиса

Гребнера идеала и решений этого идеала» сравниваются сложности задач нахождения решения системы алгебраических уравнений и базисов Гребнера идеалов этих систем.

Том завершается статьей Я.А. Хетагурова «О построении аппроксимирующих функций характеристик системы». В этой статье выявляется взаимное влияние характеристик на показатели системы, части, устройства посредством построения общих математических моделей (ММ), использующих одинаковые характеристики систем, частей, устройств. Для получения уравнений ММ применяются линейные и гиперболические аппроксимирующие зависимости, основанные на данных характеристик систем, частей и устройств, близких по назначению и применению.

Академик РАН В.П. Иванников

Расширение модели ParJava для случая кластеров с многоядерными узлами[1]

М.С. Акопян
manuk@ispras.ru

Аннотация. В работе описывается расширение модели параллельной SPMD программы возможностью использования потоков Java. Использование потоков в программе позволяет лучше утилизировать ресурсы многоядерного процессора. Разработанная модель позволяет оценивать время выполнения параллельной программы с явными обращениями к библиотеке MPI, где в каждом процессе можно использовать параллельные потоки Java. Однако следует учесть трудности, возникающие при использовании потоков в среде Java. В работе приводятся рекомендации призванные улучшить производительность многопроцессно-многопоточной программы связанные с управлением памяти JVM, настройкой сборщика мусора, управлением локальных буферов и.т.д.

Ключевые слова: параллельные вычисления; моделирование параллельной по данным программы; оценка времени выполнения; оценка масштабируемости; многоядерность; доводка производительности Java программ.

1. Введение

В начале 2000-ых годов каждый узел высокопроизводительной вычислительной платформы с распределенной памятью (кластер) содержал процессор с одним вычислительным устройством (ядро). Процесс параллельной программы использовал все доступные ресурсы узла. С развитием технологий стало возможно за минимальные накладные расходы на чипе процессора разместить более одного ядра, тем самым увеличив количество выполняемых модулей, что привело к повышению производительности процессора. Также увеличились размер кэша первого уровня, количество конвейерных устройств. Однако такие аппаратные ресурсы как кэш второго уровня, канал памяти представляют собой разделяемые ресурсы. На данный момент появились процессоры с 8-ю ядрами. Как показывает опыт, использование потоков в рамках одного узла на современных кластерах с многоядерными процессорами может увеличить

[1] Работа выполнена при финансовой поддержке Минобрнауки РФ, контракт 07.514.11.4001.

производительность параллельной программы за счет использования общей памяти и уменьшения накладных расходов.

Среда ParJava[1] предоставляет прикладному программисту набор инструментов, поддерживающих разработку прикладных параллельных программ для вычислительных систем с распределенной памятью (высокопроизводительных кластеров) на языке Java с явными обращениями к библиотеке MPI [2]. Многие инструменты среды ParJava могут выполняться на инструментальном компьютере, используя вместо разрабатываемой параллельной программы ее модель [3], которая может интерпретироваться как на целевой вычислительной системе, так и на инструментальном компьютере. Модель параллельной программы адекватно отражает ее поведение на кластере, позволяя исследовать динамические характеристики (в частности, время выполнения) разрабатываемой программы, или ее частей. Изменения, внесенные в [4] ускорили время интерпретации модели и позволили проводить интерпретацию реальных параллельных приложений с большим объемом данных на инструментальном компьютере.

В данной статье представлено расширение модели возможностью использования потоков Java - среда ParJava позволяет разрабатывать и моделировать параллельные приложения на современном языке Java, оставаясь в рамках промышленного стандарта MPI, причем каждый процесс MPI может содержать определенное количество потоков Java, утилизируя тем самым ядра процессора и общие ресурсы.

В разделе 2 описывается использование потоков в параллельной MPI программе и приводится краткое описание библиотеки для работы с потоками Java. Раздел 3 посвящен описанию расширения модели ParJava возможностью интерпретации Java потоков. В разделе 4 приводится описание эффектов призванных повысить производительность параллельной Java программы. В разделе 5 описываются результаты численных экспериментов.

2. Использование Java потоков в параллельной MPI программе. Библиотека времени выполнения для работы с потоками

Согласно стандарту MPI в параллельной программе каждый MPI процесс представляет собой многопоточную программу. В каждом потоке могут вызываться функции MPI, однако сами потоки не адресуемы (параметр rank в коммуникационных функциях MPI относится к процессу). Сообщение, отправленное процессу с идентификатором rank, может быть получено любым потоком, выполняющимся в данном процессе. Функции для работы с потоками в стандарте не определены и оставлены на усмотрение пользователя.

Как показывает опыт, использование потоков в рамках одного узла на современных кластерах с многоядерными процессорами может увеличить

производительность параллельной программы за счет использования общей памяти и уменьшения накладных расходов.

Среда ParJava позволяет разрабатывать и моделировать параллельные приложения на современном языке Java, оставаясь в рамках промышленного стандарта MPI, причем каждый процесс MPI может содержать определенное количество потоков Java. Коммуникации в параллельных SPMD программах в среде ParJava обеспечиваются двумя способами: а) обмен данных с помощью коммуникационных функций MPI (библиотека mpiJava.mpi) между процессами программы, б) функции взаимодействия (синхронизации) между потоками одного процесса в рамках одного узла вычислительной платформы (библиотека mpiJava.threads).

Коммуникационная библиотека mpiJava.mpi реализующая MPI основана на пакете mpiJava[2], который представляет собой привязку языка Java через интерфейс JNI к существующей реализации MPI (MPICH,LAM …). В библиотеке реализованы оберточные функции для стандарта MPI1.1[5].

Пакет mpiJava был расширен методами поддерживающими использование потоков согласно стандарту MPI:

- Реализованы функции из стандарта MPI MPI_INIT_THREAD, MPI_QUERY_THREAD, MPI_IS_THREAD_MAIN
- Разработан пакет mpi.threads реализующий функции взаимодействия между потоками основанный на пакете java.util.concurrent[6].

Описание пакета mpiJava.threads приведено [7], ниже дается краткое описание основных методов для работы с потоками. Методы из пакета mpiJava.threads можно разбить на следующие группы:

- Создание и управление потоками. Методы из данной группы позволяют создавать пул потоков, помещать новые задания в очередь заданий и.т.д.
- Методы атомарных операций. Методы из данной группы позволяют производить две операции над переменными программы атомарно (обращение к данным переменным другими потоками возможно только после завершения данного метода).
- Методы для работы с критическими секциями и семафорами.
- Методы, не вошедшие в другие группы.

Разработанная библиотека времени выполнения предоставляет возможность пользователю в рамках MPI процесса создавать пул потоков и выполнения программы добавлять в очередь заданий пула новые задания. Использование пула потоков позволяет избежать накладных расходов при создании и запуска новых потоков в ходе выполнения программы. Синхронизация между потоками производится посредством критических секций и семафоров.

В ParJava работа с потоками в рамках процесса MPI ведется следующим образом: в каждом процессе создается пул с фиксированным количеством

потоков. В ходе работы программы пользователь формирует пользовательские задания и передает пулу потоков. Менеджер пула передает задание первому свободному потоку, в котором и выполняется данное задание. В конце программы пул потоков закрывается.

Замечание. В среде ParJava введено ограничение, согласно которому в каждом процессе параллельной Java+MPI+threads программы используются одинаковое количество потоков.

Замечание. В среде ParJava введено ограничение, согласно которому в каждом процессе пул потоков создается в последовательной части основного потока (main) и закрывается в конце выполнения основного потока. В одном процессе нельзя создать больше одного пула потоков.

Пользовательское задание представляет собой объект пользовательского класса наследника от системного класса mpiJava.threads.PJTask. В пользовательском классе должен быть реализован метод run(), содержащий исходный код задания. Обычно метод run() содержит цикл (гнездо циклов) исходной программы, который нужно распараллелить между потоками. Взаимодействие между потоками программы производится с помощью таких механизмов как критические секции, семафоры, атомарные операции над переменными и.т.д.

Таким образом, разработанная библиотека дает возможность пользователю разрабатывать параллельные приложения на языке Java с явными обращениями к MPI и использовать потоки Java в каждом процессе программы.

3. Моделирование Java потоков и функций взаимодействия между потоками в интерпретаторе ParJava

Пусть на входе имеется параллельная MPI программа на языке Java, где в каждом MPI процессе запускается одинаковое количество потоков программы. Необходимо расширить модель параллельной программы в среде ParJava возможностью моделирования потоков Java, а также разработать и реализовать версию интерпретатора ParJava поддерживающую обработку потоков и функции взаимодействия (синхронизации) между потоками из библиотеки mpiJava.threads.

Каждый MPI-процесс моделируемой программы представляется в ее модели с помощью логического процесса (LProc). Логический процесс определен как последовательность действий (примеры действий: выполнение базового блока, выполнение операции обмена и т.п.). Каждое действие имеет определенную продолжительность. В логическом процессе определено понятие модельных часов. Начальное показание модельных часов каждого логического процесса равно нулю. После интерпретации очередного действия к модельным часам соответствующего логического процесса добавляется

значение времени, затраченного на выполнение этого действия (продолжительности). Продолжительность каждого действия, а также значения исследуемых динамических параметров базовых блоков, измеряются заранее на целевой платформе.

Пользовательские потоки моделируются с помощью расширенного логического процесса (ELProc). Каждый объект класса ELProc обладает локальными модельными часами, время которых хранится в переменной time. При установлении нового задания (pr_SetTask) запускается пользовательский поток, и время его модельных часов устанавливается равным времени модельных часов родительского процесса. При интерпретации модели интерпретатор обновляет переменную **time** потока для каждого фрагмента (базовый блок, сбалансированное гнездо циклов и.т.д.) выполненного в рамках данного потока. После завершения работы поток посылает сообщение пулу об окончании и сообщая ему время своего выполнения.

Если в основном потоке после запуска группы новых заданий с помощью метода SetTask используется блокирующий метод WaitForAll, то выполнение основного потока блокируется. Как только все потоки закончили выполнение, максимальное время выполнения запущенных потоков прибавляется к модельным часам основного потока.

Для моделирования функции взаимодействия между потоками в интерпретаторе среды ParJava используются следующие базовые примитивы:

pr_CreateThreadPool – создание пула потоков. Пользовательские задания могут выполняться только потоками из созданного пула потоков.

pr_SetTask – установка нового задания. Выбирается свободный поток из пула и ему дается новое задание на выполнение.

pr_CreateAtomic – создание атомарной переменной.

pr_CompareAndSet – атомарное сравнение и присвоение переменной.

pr_CreateSemaphore – создание семафора.

pr_Lock – блокирующий вызов для входа в регион ограниченного доступа.

pr_TryLock – неблокирующий вызов для входа в регион ограниченного доступа.

pr_Unlock – выход из региона ограниченного доступа.

pr_Copy – копирование сообщения.

Функции из библиотеки времени выполнения для работы с потоками полностью моделируются в интерпретаторе с помощью вышеописанных примитивов.

3.1. Оценка времени выполнения базовых примитивов моделирования потоков Java.

В данном разделе описывается метод оценки времени выполнения функций взаимодействия между потоками. Время выполнения базовых примитивов моделирующих операции над потоками определяется следующим образом:

pr_CreateThreadPool(n) – время создания одного потока обозначим как $\mathbf{time}_{\mathbf{tCreate}}$. В таком случае время выполнения данного примитива определяется как

$$\text{Time(pr\_CreateThreadPool(n))} = n\ \mathbf{time}_{\mathbf{tCreate}} + \mathbf{time}_{\mathbf{tnArray}}$$

где $\mathbf{time}_{\mathbf{tnArray}}$ время необходимое для создания массива для хранения n потоков.

pr_SetTask(task, isBlocking) – пусть $\mathbf{time}_{\mathbf{tStart}}$ время запуска потока, а $\mathbf{time}_{\mathbf{tRun}}$ время выполнения потока с заданием task. В этом случае

$$\text{Time(pr\_SetTask(task, isBlocking))} = \mathbf{time}_{\mathbf{tStart}} + (\text{isBlocking} ?\ \mathbf{time}_{\mathbf{tRun}} : 0)$$

Переменная $\mathbf{time}_{\mathbf{tRun}}$ определяется во время интерпретации и равно разности показаний модельных часов пользовательского потока и родительского потока.

Время работы примитивов **pr_CreateAtomic(atomic,initValue)**, **pr_CreateSemaphore(initN, mode), pr_Unlock(sem,n)** константно и определяется посредством тестов – пусть $\mathbf{time}_{\mathbf{aCreate}}$ время создания атомарной переменной, $\mathbf{time}_{\mathbf{sCreate}}$ время создания семафора, $\mathbf{time}_{\mathbf{release}}$ время разблокировки семафора.

$$\text{Time(pr\_CreateSemaphore)} = \mathbf{time}_{\mathbf{sCreate}}$$

$$\text{Time(pr\_Unlock)} = \mathbf{time}_{\mathbf{release}}$$

$$\text{Time(pr\_CreateAtomic)} = \mathbf{time}_{\mathbf{aCreate}}$$

pr_TryLock(sem,n) - пусть $\mathbf{sem}_{\mathbf{counter}}$ значение счетчика в семафоре. Переменная $\mathbf{sem}_{\mathbf{ts}}$ представляет собой временную метку семафора. Когда поток освобождает регион с ограниченным доступом (примитив pr_Unlock) переменной семафора $\mathbf{sem}_{\mathbf{ts}}$ присваивается показание модельных часов потока.

$$\text{Time(pr\_TryLock(sem,n))} = \mathbf{time}_{\mathbf{cmp}} + (((\mathbf{sem}_{\mathbf{counter}} - n) \geq 0) ?$$

$$\text{pr\_CompareAndSet(atomic, } \mathbf{sem}_{\mathbf{counter}}, (\mathbf{sem}_{\mathbf{counter}} - n) : 0)$$

где $\mathbf{time}_{\mathbf{cmp}}$ время сравнения $(\mathbf{sem}_{\mathbf{counter}} - n) \geq 0$.

pr_Lock(sem,n) – пусть $\mathbf{time}_{\mathbf{lock}}$ время блокировки потока и определяется следующим образом $\mathbf{time}_{\mathbf{lock}} = (\mathbf{time} < \mathbf{sem}_{\mathbf{ts}})\ ?\ (\mathbf{sem}_{\mathbf{ts}} - \mathbf{time}) : 0$. Тогда

$$\text{Time(pr\_Lock(sem,n))} = \text{Time(pr\_TryLock(sem,n))} + (\text{pr\_TryLock(sem,n)} ?$$

$$\mathbf{time}_{\mathbf{lock}} : 0)$$

pr_CompareAndSet(atomic,expect,update) - $\mathbf{time_{aCAS}}$ время сравнения и присвоения переменной. Эта операция производится следующим образом – вначале открывается критическая секция, после чего производиться сравнение и присвоение переменной, и закрытие критической секции. Тогда время выполнения примитива определяется как

$$\text{Time(pr\_CompareAndSet)} = \mathbf{time_{aCAS}} + \text{Time(pr\_Lock(sem,1))} + \text{Time(pr\_Unlock(sem,1))}$$

pr_Copy(n) – пусть $\mathbf{time_{copy}}$ время копирования одного байта и определяется с помощью тестов. В этом случае

$$\text{Time(pr\_Copy())} = n\ \mathbf{time_{copy}}$$

Параметры $\mathbf{time_{tCreate}}$, $\mathbf{time_{tnArray}}$, $\mathbf{time_{tStart}}$, $\mathbf{time_{sCreate}}$, $\mathbf{time_{release}}$, $\mathbf{time_{aCreate}}$, $\mathbf{time_{aCAS}}$, $\mathbf{time_{cmp}}$, $\mathbf{time_{copy}}$ определяются с помощью тестов на узле вычислительной платформы.

Замечание. Значение параметров $\mathbf{time_{tStart}}$, $\mathbf{time_{sCreate}}$, $\mathbf{time_{release}}$, $\mathbf{time_{aCreate}}$, $\mathbf{time_{aCAS}}$, $\mathbf{time_{cmp}}$ сильно меньше чем значение остальных параметров и их вклад в определении времени выполнения коммуникаций незначителен, поэтому в дальнейшем будем считать их равными нулю.

Время выполнения функций из групп создания и управления потоками, методов атомарных операций, методов для работы с критическими секциями элементарно вычисляются, используя оценки времени выполнения базовых примитивов приведенных выше.

Таким образом, имея оценки времени выполнения базовых примитивов определяется время выполнения основных функций взаимодействия между потоками в пользовательской программе.

4. Доводка производительности параллельной программы в среде Java

В данном разделе приводится описание эффектов призванных повысить производительность параллельной программы. Вначале сравнивается время выполнения обмена данными между процессами программы и время выполнения чтения данных между потоками программы. После чего приводится описание нескольких эффектов связанных с реализацией JVM, влияющих на производительность параллельной программы. Утилизация специфических факторов описанных в данном разделе прикладным программистом может привести к потенциальному росту производительности параллельной программы.

4.1. Операция «обмена данными» между процессами/потоками Java

Рассмотрим время выполнения **операции «чтение данных соседнего процесса/потока»** для многопроцессной программы, в котором используются неблокирующие, блокирующие коммуникаций MPI, и для многопоточной программы.

Пусть имеется многопроцессная программа P1 и многопоточная программа P2. Для программы P1 (P2) в каждом процессе (потоке) производятся вычисления и в определенный момент процесс (поток) 0 становится потребителем (p_0) и должен получить доступ к данным процесса (потока) 1 который в этом взаимодействии является производителем (p_1).

В многопроцессной программе P1 при использовании **неблокирующих** коммуникаций время выполнения операции «чтение данных соседнего процесса» определяется следующей формулой:

$$\text{time}_{\text{wait}} = \begin{cases} (t_{ready} - t_0) + time_{comm}, t_{ready} > t_0 \\ (t_0 - t_{ready}) > time_{comm} \,?\, 0 : time_{comm} - (t_0 - t_{ready}), t_{irecv} < t_{ready} < t_0 \\ (t_0 - t_{irecv}) > time_{comm} \,?\, 0 : time_{comm} - (t_0 - t_{recv}), t_{ready} < t_{irecv} \end{cases}$$

где t_{irecv} - момент времени вызова iRecv в процессе p_0, t_0 - момент времени вызова wait в процессе p_0, t_{ready} - момент времени когда в процессе p_1 готовы данные, $\text{time}_{\text{comm}}$ - время необходимое для пересылки данных из процесса-производителя к процессу-потребителю и копирования в буфер памяти в процессе-потребителе, $\text{time}_{\text{wait}}$ – время выполнения функции wait.

При использовании блокирующих коммуникаций:

$$\text{time}_{\text{wait}}=\text{time}_{\text{recv}}=\begin{cases} (t_{ready} - t_0) + time_{comm}, t_{ready} > t_0 \\ time_{comm}, t_{ready} < t_0 \end{cases}$$

где $\text{time}_{\text{recv}}$ – время выполнения функции Recv, t_0 - момент времени вызова Recv в процессе p_0, t_{ready} - момент времени, когда в процессе p_1 готовы данные, $\text{time}_{\text{comm}}$ - время необходимое для пересылки данных из процесса p_1 к процессу p_0 и копирования в буфер памяти в процессе p_0.

В случае с **многопоточной** программой P2 поток производитель p_1 вызывает функцию блокирования lock и начинает вычислять данные. Как только данные вычислены он разблокирует критическую секцию. Поток-потребитель p_0, не может считать данные пока p_1 не разблокирует критическую секцию (p_0 блокируется в lock-е). Время выполнения операции «чтение данных соседнего потока» определяется следующей формулой:

$$\text{time}_{\text{lock}}=\begin{cases} 0; t_{ready} \le t_0 \\ t_{ready} - t_0, t_{ready} > t_0 \end{cases}$$

где t_0 - момент времени вызова lock в потоке-потребителе p_0, t_{ready} - момент времени, когда в потоке-производителе p_1 готовы данные, $time_{lock}$ – время выполнения функции lock.

Отсюда видно, что в любом случае (при использовании блокирующих или неблокирующих коммуникаций и с учетом реализации обмена данными в MPI на основе общей памяти) в многопроцессной программе P1 на операцию «чтение данных соседнего процесса» тратится больше времени, чем в многопоточной программе P2.

Когда процесс вызывает функцию wait (явно при неблокирующей посылке или посредством recv), то при определенных условиях (не готовы данные или производится пересылка) вызов блокируется. Блокирование происходит внутри реализации MPI и в зависимости как реализована операция блокирования в MPI это может возыметь определенные последствия:

1. если lock реализован как lock-sleep, то произойдет переключение контекста и когда данные прибудут, процесс пробудится и опять будет произведен переключение контекста
2. если lock реализован посредством spin-lock, то процесс останется активным и переключений контекстов не произойдет.

В случае с **многопоточной** программой операцию lock определяет пользователь. Если он явно организует spin-lock, то очевидно контекст переключаться не будет. Когда пользователь вызывает функцию lock (например semaphore.acquire()), то какой именно подход блокирования будет выбран зависит от реализации JVM.

Переключение контекста не является бесплатной операцией – управление CPU передается JVM и операционной системе, которые сохраняют данные потока и отправляют в спячку (режим ожидания). Когда поток должен пробудиться, то система (JVM) загружает его фрейм и данные в память. И скорее всего это переключение повлечет за собой определенное количество кэш промахов, что в свою очередь также отразиться на производительности.

4.2. Синхронизация потоков при обращении к данным в общей памяти Java

При использовании общей памяти потоками параллельной программы значительный вес в доводке производительности программы получают эффекты, связанные с синхронизацией доступа потоков к общим данным. В Java синхронизация данных ведет к появлению дополнительных накладных расходов: синхронизация обеспечивается специальными инструкциями барьеров памяти (memory barriers), которые могут сбрасывать кэш, аппаратные буферы, а также повлиять на другие оптимизации компилятора (предотвратить переупорядочивание инструкций). Синхронизация бывает оспариваемая (**contended)** и неоспариваемая (**uncontended)**. «неоспариваемая» синхронизация имеет низкую стоимость (20-250 тактов)[8]. «Неоспариваемая»

синхронизация может обрабатываться внутри JVM, в то время как «оспариваемая» требует вмешательства операционной системы, что увеличивает ее стоимость. Динамический компилятор Java может сократить стоимость синхронизации, заменив «оспариваемую» синхронизацию «неоспариваемой», когда докажет, что за данный барьер никогда не производится спор между потоками (contention). Существует три способа сократить «спор»:

- Сократить продолжительность блокирования
- Сократить частоту запросов блокирования
- При возможности заменить исключающую блокировку механизмами позволяющими улучшить параллелизм

4.3. Эффекты, связанные со сборщиком мусора в Java

В **многопроцессной** программе каждый процесс запускается на отдельном JVM. Необходимая память для работы процесса выделяется системой управления памяти **своего JVM**. В этом случае, для каждого процесса работает **свой сборщик мусора(gc)**. В случае же с **многопоточной** программой память необходимая всем потокам программы выделяется на одном JVM, и это надо учесть при запуске программы в опциях интерпретатора(xmx,…). Помимо этого здесь работает только **один gc**, который обслуживает все потоки программы. Следовательно, давление на gc возрастает пропорционально количеству потоков программы.

В JVM освобождением не нужной памяти занимается сборщик мусора, который время от времени запускается в отдельном потоке и может стать узким местом в параллельной программе. С целю улучшения производительности необходимо уменьшить время работы gc, а в идеальном случае и вовсе свести на ноль. Сборщик мусора перебирает кучу (heap) и выделяет (маркирует) те области, на которые есть живые ссылки. После этапа маркировки производится чистка всех немаркированных областей. Обычно большинство объектов выделенных в куче используются не долго (умирают рано), они хранятся в молодом поколении. Другие объекты могут жить до конца работы программы, и они хранятся в долгоживущем поколении. Процесс сборки мусора можно оптимизировать, используя эффект ранней смерти. С целю уменьшения времени работы gc общая память в куче разбивается на несколько областей:

1. Молодое поколение(young generation) – хранятся молодые объекты.
2. Долгоживущее поколение(tenured) – хранятся долгоживущие объекты.
3. Перманентная область (permanent) – здесь хранятся служебные данные JVM.

Как только в поколении не остается места производиться сборка мусора для данного поколения: минорная сборка(minor collection) для молодого

поколения и мажорная сборка (major collection) для долгоживущего поколения. Большинство выделенных объектов умирают молодыми, следовательно, минорная сборка занимает меньше времени, потому что время сборки прямо пропорционально количеству живых объектов. При создании объекта в программе, производится попытка выделить память в молодом поколении, если этого не удается (молодое поколение переполнено), то производится минорная сборка (minor collection), чистящая молодое поколение. Во время минорных сборок, объекты, которые прожили достаточно долго, переносятся в долгоживущее поколение. Как только долгоживущее поколение переполняется, производится мажорная сборка, которая работает намного медленнее из за большого количества живых объектов в нем.

Для уменьшения влияния сборщика мусора на производительность программы рекомендуется:

1. Большие матрицы и общие данные выделить в самом начале программы.
2. Вычислить объем данных выделенных в первом шаге, и соответствии с этим задать минимальный и максимальный размер кучи (Xmx,Xms).
3. Задать размеры поколений (NewRatio, NewSize, MaxNewSize).

Чем больше размер молодого поколения, тем меньше будет минорных сборок. С другой стороны, чем больше размер молодого поколения, тем меньше размер долгоживущего поколения, что в свою очередь приведет к частым мажорным сборкам. Поэтому размеры поколения нужно выбрать так, что большие матрицы, выделенные в начале программы, попадали сразу в долгоживущее поколение, а так же оценить размер памяти, который будет выделяться по ходу выполнения программы, и задать соответствующий размер молодого поколения так, чтобы не было (по возможности) минорных сборок.

4.4. Эффекты, связанные с буфером локальной памяти потоков в Java

В многопоточной программе выделение памяти потоками производится параллельно. В общем случае все потоки выделяют память в общей куче. Каждый раз при выделении памяти в общей куче(heap) система управления памяти ставит lock, выделяет память и отпускает lock, что может повлиять на производительность общей программы. Поэтому рекомендуется большую матрицу и общие данные выделить в самом начале программы, а локальные данные для каждого потока выделять в локальной области, что позволит избежать lock-ов на общей куче.

Для улучшения производительности при запуске каждого потока JVM создает для него локальный буфер для выделения памяти(TLAB). Во время работы потока при выделении памяти, вначале производится попытка выделить память из области TLAB-а и если эта попытка оказывается неудачной, то

память выделяется в общей куче со всеми вытекающими последствиями. При запуске программы необходимо знать, сколько памяти будет выделено во время работы потока и задать соответствующие корректировки относительно работы с TLAB-ами с помощью опций интерпретатора Java (ResizeTLAB, TLABSize,…).

5. Результаты численных расчетов

Модель, приведенная в данной статье, был реализован в среде ParJava и апробирован на параллельной программе FT из набора NAS Parallel Benchmarks [9,10]. NPB на языке Java был разработан и реализован в университете Coruna [11].

В рамках работы проведенной в данной статье, оригинальная реализация FT была оптимизирована с учетом эффектов описанных в разделе 4 и в результате время выполнения программы уменьшилась в более чем два раза. Помимо этого был реализован многопоточный вариант программы: коммуникации между процессами программы обеспечиваются функциями MPI, а для взаимодействия между потоками в каждом процессе используются функции из библиотекой времени выполнения mpiJava.threads. Проведена серия экспериментов по сравнению параллельной программы основанной только на библиотеке MPI и параллельной программы на основе MPI+threads. Была проведена оценка времени выполнения многопроцессно-многопоточной программы и результаты приведены в данном разделе.

Приведем описание параллельной программы FT. Пусть имеется последовательность $u=\{u_0, u_1,\dots u_{N-1}\}$. Дискретное преобразование Фурье (задача 1D FFT) преобразует последовательность u в другую последовательность U, где

$$U_k = \sum_{n=0}^{N-1} u_n e^{\frac{-2\pi i k n}{N}}$$

В задаче FT дискретное преобразование Фурье применяется на 3D сетке размерностью LxMxN. В этом случае формула преобразование Фурье принимает следующий вид:

$$F_{q,r,s}(u) = \sum_{l=0}^{L-1}\sum_{k=0}^{M-1}\sum_{j=0}^{N-1} u_{j,k,l} e^{\frac{-2\pi i j q}{L}} e^{\frac{-2\pi i k r}{M}} e^{\frac{-2\pi i l s}{N}}$$

Данная задача вычисляется на высокопроизводительной вычислительной платформе с распределенной памятью. В связи с этим реализованы два вида декомпозиции данных 1D декомпозиция и 2D декомпозиция.

В задаче 3D FFT при применении 1D декомпозиции пространство данных разбивается на слои – каждому процессу отдается один слой (см. рис. 1).

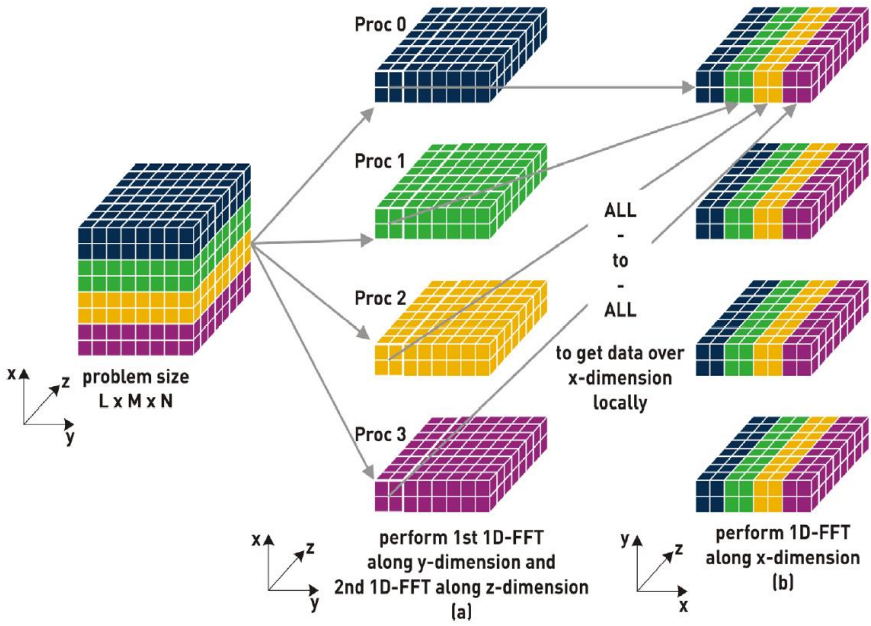


Рис. 1. 1D декомпозиция в решении задачи 3D FFT.

Параллельные вычисления в этом случае производится в 3 этапа

1. 2D FFT(либо 2 раза по 1D FFT) вдоль двух локальных направлениях в рамках процесса (оси z и y).
2. Глобальное транспонирование, что представляет собой коммуникацию AllToAll между всеми процессами. В результате у всех процессов появляются данные для счета вдоль оси x.
3. 1D FFT вдоль 3-го направления.

При использовании 1D декомпозиции используется только одно глобальное транспонирование для получения в локальную память процесса данных необходимых для счета. Недостатком 1D декомпозиции является ограничение масштабируемости (максимального параллелизма) размером наибольшей длины 3D сетки данных. Однако степень параллелизма можно увеличить количеством ядер на каждом узле кластера при использовании комбинированного алгоритм 1D декомпозиции и распараллеливания по потокам в рамках одного узла.

В случае 2D декомпозиции (см рис. 2) вычисления производятся в 5 этапов

1. 1D FFT в одном направлении (ось y).
2. глобальное транспонирование.

3. 1D FFT по второму направлению (ось z).
4. глобальное транспонирование.
5. 1D FFT по третьему направлению (ось x).

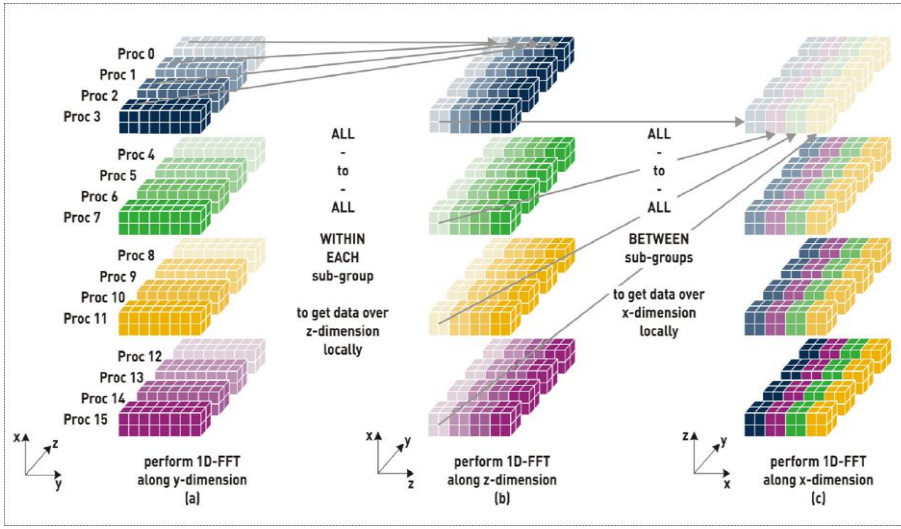


Рис. 2. 2D декомпозиция в решении задачи 3D FFT.

Пусть исходные данные представляют собой куб N^3. В этом случае при 1Ddекомпозиция параллельная программа масштабируется O(N), а при 2D декомпозиция $O(N^2)$.

Программа быстрого преобразования FT тестировалась на кластере МВС-100К (результаты см. рисунок XXX1), на 1410-ти 4-ядерных процессора Intel(R) Xeon(R) CPU X5365 с частотой 3.00GHz (два процессора на каждом узле), с интерфейсной платой HP Mezzanine Infiniband 4x DDR и 8Gb памяти на каждом узле.

На рис. 3 представлены графики зависимости времени выполнения от числа используемых ядер вычислительной платформы для программы быстрого преобразования Фурье (БПФ). Размер рассчитываемой матрицы 512x512x256, объем памяти требуемой в задаче примерно 5Gb, количество внешних итераций 20. Использовалаясь 64-разрядная версия среды Java.

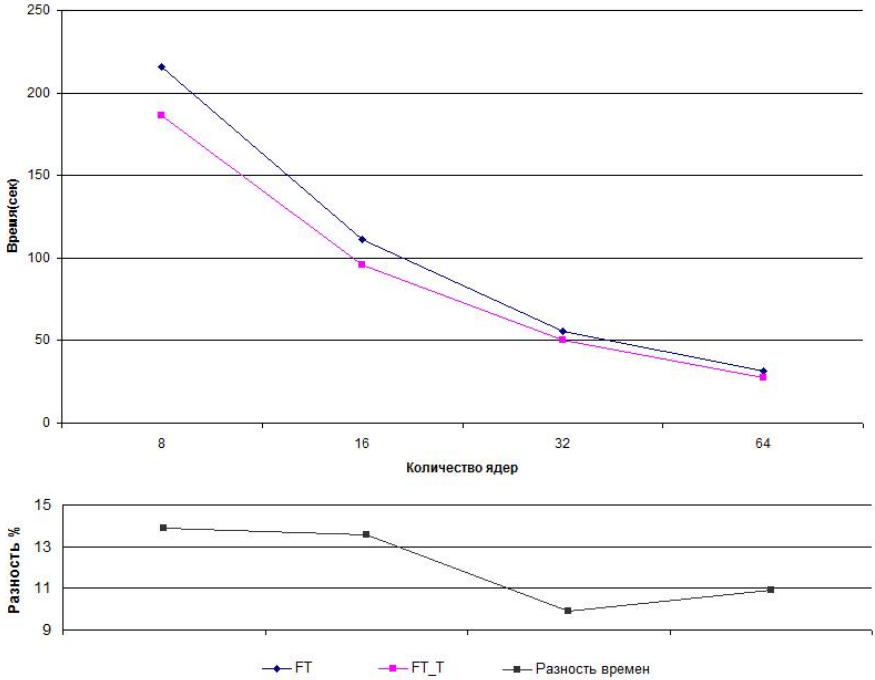


Рис. 3. Зависимость Времени выполнения программы быстрого преобразования Фурье от количества используемых ядер.

На приведенном графике FT представляет собой параллельную программу быстрого преобразования Фурье на языке Java с явными обращениями к MPI. В этом случае применялся 2D декомпозиция, на каждом узле кластера запускается по восемь процессов. FT_T представляет собой параллельную программу быстрого преобразования Фурье, с использованием и интерфейса MPI, и потоков Java. В случае с FT_T применяется 1D декомпозиция – на каждом узле запускается один процесс, внутри которого при обработке гнезд циклов программы порождаются по восемь потоков. «Разность» представляет собой пройентное соотношение разности времен программы FT и FT_T. Алгоритм, используемый в NPB, предполагает, что количество используемых процессов является степенью двойки.

Время выполнения программы FT_T(MPI и потоки Java) 9-14% меньше, чем аналогичной программы FT, где используются только процессы MPI.

Число процессов для FT принимало значения 8 (вычисления проводились на 1 узле), 16 (на 2 узлах), 32 (на 4 узлах) и 64 (на 8 узлах), т.е. на каждом ядре каждого узла был запущен *MPI*-процесс. Для программы FT_T в каждой точке используется по одному процессу на узле, но в каждом процессе используется

по 8 потоков. Таким образов в количество используемых потоков будет 8(на 1 узле), 16 (на 2 узлах), 32(на 4 узлах) и 64(на 8 узлах).

На рис. 4 FT_T представляет собой ускорение программы, измеренное при выполнении программы на вычислительной платформе, а FT_T_Interp - ускорение, предсказанное интерпретатором ParJava на инструментальной машине. При сравнении со временем реального запуска на вычислительной платформе получились существенные погрешности.

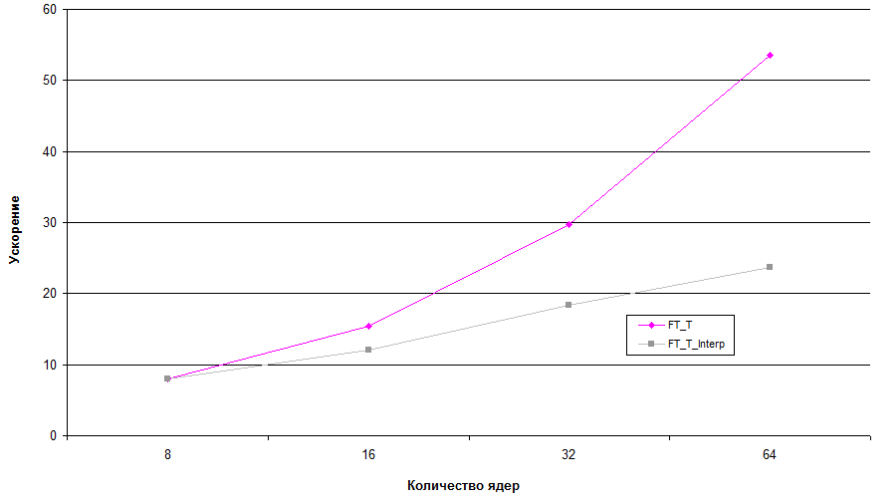


Рис.4. Сравнение предсказанного и фактически полученного ускорения программы FT_T.

Исследования показали, что причина в модели коммуникации метода AlltoAll в интерпретаторе ParJava. Для моделирования коммуникационной функции AlltoAll был реализован алгоритм попарного обмена сообщений с большой длинной. В результате время коммуникаций сократилось больше чем в два раза. На рис. 5 приведено сравнение ускорения по амдалю для программы FT_T и ускорения предсказанного интерпретатором модели после реализации новой версии функции AlltoAll.

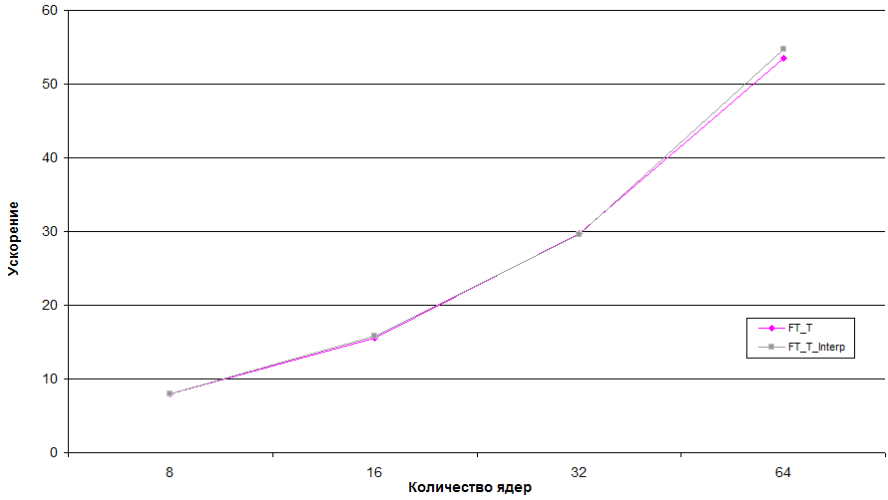


Рис. 5. Ускорение программы БПФ (новая модель коммуникации AlltoAll в интерпретаторе).

Из рис. 5 видно, что интерпретатор модели достаточно точно(3-7%) предсказал ускорение программы при увеличении количества процессов/потоков.

6. Заключение

Расширение модели возможностью работы с потоками было разработано и реализовано в среде ParJava. Была разработана библиотека времени выполнения для работы с потоками и соответствующие модельные функции в интерпретаторе модели в среде ParJava. Разработан метод, оценивающий динамические характеристики (время выполнения) функций взаимодействия между потоками.

Была реализована параллельная программа быстрого преобразования Фурье FT_T на основе параллельной программы FT из тестовых бенчмарков NPB[9,10]. Программа FT_T представляет собой параллельную по данным MPI программу, где в каждом процессе для вычисления основных циклов программы порождаются параллельные потоки (по количеству ядер процессора). Как показали эксперименты программа FT_T за счет использования общих ресурсов и меньших коммуникаций работает быстрее FT на 9-14%.

Однако следует учесть трудности с которыми сталкивается прикладной программист, разрабатывающий параллельное приложение с применением потоков Java (управление памятью, управления локальными буферами

потоков, уменьшение «оспаривания», и.т.д.). В работе приведены основные рекомендации призванные улучшить производительность параллельного приложения с учетом нюансов связанных с JVM при работе с многопоточной программой.

Реализованная модель многопроцессно-многопоточной программы была применена для предсказания параллельного приложения FT_T. Погрешность предсказаний составляет 3-7%.

Список литературы

[1] В.П. Иванников, А.И. Аветисян, С.С. Гайсарян, В.А. Падарян. Оценка динамических характеристик параллельной программы на модели. // «Программирование» 2006, №4, с. 21–37

[2] 12. Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko and Sang Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. Revised version, August 1999.

[3] Иванников В. П., Аветисян А. И., Гайсарян С. С., Акопян М. С. Особенности реализации интерпретатора параллельных программ в среде ParJava. // «Программирование» 2009, №1, с. 10-25

[4] А.И. Аветисян, М.С. Акопян, С.С. Гайсарян. Методы точного измерения времени выполнения гнезд циклов при анализе JavaMPI-программ в среде ParJava. Труды Института системного программирования РАН, том 21, 2011, с. 83-102

[5] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. MPI – The complete Reference, Volume 1, The MPI Core, Second edition. / The MIT Press. 1998

[6] http://docs.oracle.com/javase/tutorial/essential/concurrency/

[7] http://www.ispras.ru/ru/parjava/mpijava.php

[8] Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. Java Concurrency In Practice. Addison Wesley Professional, May 19, 2006, p. 384

[9] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga. THE NAS PARALLEL BENCHMARKS. THE NAS PARALLEL BENCHMARKS. RNR Technical Report RNR-94-007, March 1994

[10] H. Jagode, “Fourier Transforms for the BlueGene/L Communications Network”, Master’s thesis, University of Edinburgh, 2006.

[11] Dami´an A. Mall´on, Guillermo L. Taboada, Juan Touri˜no, and Ram´on Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. // Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09). Weimar, Germany, Feb 2009, pp. 181-190.

Extension of ParJava model for HPC clusters with multicore nodes

M.S. Akopyan
ISP RAS, Moscow, Russia
manuk@ispras.ru

Annotation. At the beginning of 2000 each node of high performance cluster with distributed memory contained processor with single core and each MPI process of parallel application used to utilize all resources of node. At this moment vendors propose microprocessors with multiple cores on chip and few processors on the single board. Using multiple threads in a single node with modern multicore processors allows to increase performance of parallel application due to usage of shared memory and lower overhead. An extension of model for parallel SPMD programs has been developed with ability to use Java threads. The usage of threads in program allows better utilization of the resources of multicore processor. Developed model allows estimate execution time of parallel program with explicit calls to MPI library, where parallel Java threads could be used in each process. However, there are a lot of problems arising when threads have been used in Java environment. This paper contains recommendations called for performance tuning of multiprocessed-multithreaded program concerning to JVM memory management, garbage collector configuration, management of local buffers etc. Java version of parallel application FT (Fast Furier Transformation) from NPB has been adapted for multiprocess-multithreaded environment. Tests on implemented application show 9-14% performance improvement. Model of multiprocess-multithreaded application has been developed. Performance prediction for multiprocess-multithreaded FT shows 3-7% prediction error.

Keywords: parallel computing; SPMD program simulation; execution time estimation; scalability estimation; multi-core; performance tuning of Java program.

References

[1]. Ivannikov V.P., Avetisyan A.I., Gaissaryan S.S., Padaryan V.A. Estimation of dynamical characteristics of a parallel program on a model. Programming and Computer Software. 2006. Volume 32, Issue 4. pp. 203-214. doi: 10.1134/S0361768806040037

[2]. 12. Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko and Sang Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. Revised version, August 1999.

[3]. Ivannikov V.P., Avetisyan A.I., Gaissaryan S.S., Akopyan M.S. Implementation of Parallel Interpreter in the Development Environment ParJava. Programming and Computer Software. 2009. Volume 35, Issue 1. pp. 6-17. doi: 10.1134/S0361768809010034

[4]. Avetisyan A.I., Akopyan M.S, Gaissaryan S.S. Metody tochnogo izmereniya vremeni vypolneniya gnezd tsiklov pri analize JavaMPI-program v srede ParJava [The methods of precise measurement of the loop nests' execution time during JavaMPI-programs analysis in ParJava environment]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 21, pp. 83-102 (in Russian).

[5]. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. MPI – The complete Reference, Volume 1, The MPI Core, Second edition. / The MIT Press. 1998
[6]. http://docs.oracle.com/javase/tutorial/essential/concurrency/
[7]. http://www.ispras.ru/ru/parjava/mpijava.php
[8]. Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. Java Concurrency In Practice. Addison Wesley Professional, May 19, 2006, p. 384
[9]. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga. THE NAS PARALLEL BENCHMARKS. THE NAS PARALLEL BENCHMARKS. RNR Technical Report RNR-94-007, March 1994
[10]. H. Jagode, “Fourier Transforms for the BlueGene/L Communications Network”, Master’s thesis, University of Edinburgh, 2006.
[11]. Dami´an A. Mall´on, Guillermo L. Taboada, Juan Touri˜no, and Ram´on Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09). Weimar, Germany, Feb 2009, pp. 181-190.

Реализация конвейеризации циклов и встраивания присваиваний в трансляторе C-to-HDL[1]

Алексей Меркулов, Андрей Белеванцев
<steelart@ispras.ru>,<abel@ispras.ru>

Аннотаци. Реализация алгоритмов на программируемых логических интегральных схемах с помощью языков описания аппаратуры является сложной задачей. Поэтому, инструмент, позволяющий эффективно транслировать алгоритм с языка высокого уровня в язык описания аппаратуры, был бы очень полезен. В данной статье рассматривается инструмент для трансляции функций языка Си в модули на языке Verilog, процесс трансляции и две реализованных на уровне описания аппаратуры оптимизации: встраивание присваиваний и конвейеризация циклов. Результаты тестирования показывают, что эти оптимизации существенно увеличивают производительность генерируемого кода.

Ключевые слова: ПЛИС, Verilog, автоматическая трансляция, встраивание присваиваний, конвейеризация циклов.

1. Введение

Программируемые логические интегральные схемы (ПЛИС) в последнее время стали всё чаще рассматриваться, как вычислительные ускорители, подобно тому, как используются GPU. Однако, реализация вычислительных алгоритмов для ПЛИС-устройств сопряжено с рядом трудностей. ПЛИС-устройства конфигурируются при помощи языков описания аппаратуры (hardware description language, HDL). Таким образом, разработчик должен знать язык описания аппаратуры и основы схемотехники.

Разработка и отладка описания аппаратуры являются намного более сложными задачами, чем при разработке программ на языке высокого уровня. Повышенная сложность разработки вызвана целым рядом причин. Для высокоуровневых программ характерна последовательная модель исполнения инструкций, параллелизм возникает либо на уровне инструкций с сохранением семантики последовательного исполнения, либо его параллелизм задаётся на уровне процессов, внутри которых тем не менее сохраняется

[1] Работа выполнена при финансовой поддержке Минобрнауки РФ, контракт № 07.514.11.4001.

последовательная семантика инструкций. Однако для описаний аппаратуры применяется принципиально параллельная модель исполнения. При такой модели исполнения становится крайне сложным отслеживать зависимости и влияния различных компонентов аппаратуры друг на друга. Для отладки описания аппаратуры используются симуляторы языков описания аппаратуры. С их помощью можно отладить отдельные компоненты системы, однако промоделировать и предсказать поведение всей системы в целом далеко не всегда удаётся. Существуют средства отладки уже реализованных на ПЛИС описаний аппаратуры. Однако эти средства предоставляют ограниченные возможности по сравнению со средствами отладки программ.

Исходя из названных выше причин, инструмент, позволяющий эффективно транслировать алгоритм с языка программирования в язык описания аппаратуры, был бы крайне полезным при разработке описаний аппаратуры.

В данной статье рассматривается трансляция с подмножества языка Си в язык описания аппаратуры Verilog. В процессе работы был разработан транслятор C-to-HDL, основанный на концепциях транслятора C-to-Verilog[1]. В процессе работы транслятор C-to-Verilog был полностью переписан и были добавлены такие оптимизации, как встраивание присваиваний и конвейеризация циклов на уровне HDL-присваиваний. Результаты тестирования показывают, что данные оптимизации значительно увеличивают производительность генерируемого кода.

В настоящей статье в разделе 2 предлагается обзор существующих открытых решений в области трансляции с языка высокого уровня в язык описания аппаратуры. Раздел 3 описывает реализованный в данной работе транслятор и оптимизацию встраивания присваиваний. Раздел 4 содержит описание конвейеризации циклов на уровне HDL-присваиваний. Раздел 5 описывает результаты тестирования, а раздел 6 завершает статью.

2. Обзор существующих решений

В настоящее время существует несколько систем, производящих трансляцию языка программирования в описание аппаратуры. Обычно эти системы реализуются как транслятор с языка программирования Си в язык описания аппаратуры. Рассмотрим два популярных открытых транслятора: ROCCC[2] и C-to-Verilog.

2.1. ROCCC

ROCCC (Riverside Optimizing Compiler for Configurable Computing) является открытым транслятором с подмножества языка Си в VHDL. Главным преимуществом транслятора является эффективность генерируемого VHDL кода. К сожалению, эффективность достигается за счёт наложения строгих ограничений на исходный код. В частности, тело функции должно состоять из строго вложенного гнезда циклов (другие циклы разрешаются только в случае,

если их можно полностью развернуть); сумма (или другая функция) от нескольких индукционных переменных не может быть использована, как индекс массива. Также, запрещается использовать циклы `while`, `do...while`, адресную арифметику, логические сдвиги на неконстантное значение, рекурсивные функции. В качестве внутреннего представления ROCCC использует комбинацию из представлений SUIF[3] и LLVM[4]. Стоит отметить, что исходные коды актуальной версии транслятора ROCCC доступны только по запросу. К сожалению, получить их для дальнейшего анализа не удалось.

2.2. C-to-Verilog

C-to-Verilog является открытым транслятором с языка программирования Си в язык описания аппаратуры Verilog. Единицами трансляции являются функции, которые транслируются в модули. C-to-Verilog является кодогенератором компилятора LLVM. В отличие от транслятора ROCCC, C-to-Verilog не накладывает сильных ограничений на исходный код функций. Тем не менее, не разрешается использовать типы данных с плавающей точкой (float, double), глобальные переменные, многомерные массивы и рекурсивные функции. Однако, эффективность генерируемого кода у C-to-Verilog является относительно низкой по сравнению с ROCCC. C-to-Verilog выполняет лишь несколько простых оптимизаций, таких как базовое встраивание присваиваний на уровне HDL-представления и программная конвейеризация на уровне инструкций LLVM.

3. Реализация

В качестве основы для транслятора был выбран транслятор C-to-Verilog как имеющий небольшое количество ограничений и простую инфраструктуру, основанную на компиляторе LLVM. Однако требовалось увеличить эффективность генерируемого кода.

К сожалению, в процессе работы выяснилось, что инфраструктура C-to-Verilog недостаточно гибка для реализации необходимых оптимизаций. Поэтому было решено создать новый транслятор, C-to-HDL, основанный на реализованных в C-to-Verilog идеях. На данный момент C-to-HDL поддерживает в качестве целевого языка только Verilog, однако добавить другой целевой язык не составит большой проблемы.

C-to-HDL, так же как и C-to-Verilog, реализован как кодогенератор компилятора LLVM, позволяя таким образом транслировать не только с языка программирования Си, но и с любого другого языка программирования, поддерживаемого LLVM. Трансляция с входного языка программирования в промежуточное представление (байт-код LLVM) и ряд высокоуровневых оптимизаций осуществляются стандартными средствами LLVM. Таким образом, на вход C-to-HDL получает байт-код LLVM.

Будем понимать под *присваиванием* неблокирующее присваивание в терминах языка Verilog. Каждая инструкция LLVM транслируется в набор присваиваний. φ-функции тоже транслируются в присваивания, однако отличным от остальных инструкций способом.C-to-HDL использует смешанное представление уровня инструкций LLVM и уровня HDL-присваиваний, чтобы удобно и эффективно выполнять планирование и оптимизацию встраивания присваиваний, поскольку инструкции LLVM являются минимальными единицами для перечисленных преобразований (присваивания одной инструкции не могут рассматриваться как независимые друг от друга), но преобразования сами по себе выполняются на уровне HDL-присваиваний.

3.1. Интерфейс и структура генерируемых модулей

Единицей трансляции C-to-HDL является функция. Считается, что каждый массив, являющийся параметром функции, соответствует отдельному блоку памяти. Число портов для каждого блока памяти может настраиваться. Локальные массивы и динамически выделяемую память использовать не разрешается, поскольку они не могут быть непосредственно оттранслированы в язык описания аппаратуры. В будущем это ограничение может быть ослаблено поддержкой локальных массивов фиксированного размера, поскольку они тривиально транслируются в набор регистров. Си-структуры и типы данных с плавающей точкой также не поддерживаются в текущей версии.

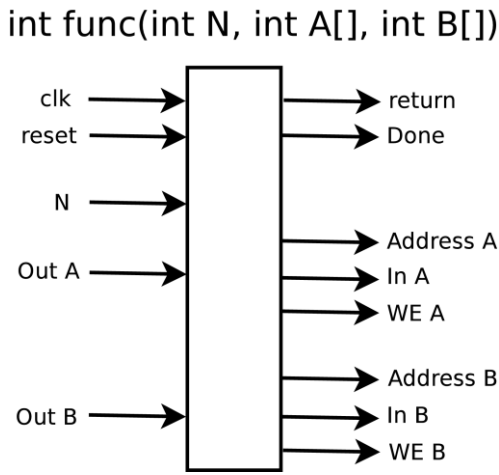


Рисунок 1. Пример интерфейса генерируемого модуля.

Доступ к массивам транслируется в доступ к блокам памяти. Поэтому если в теле транслируемой функции используются указатели, то их использование

должно быть достаточно простым, чтобы в каждой точке функции можно было бы определить, на какой массив и с каким смещением ссылается указатель. Другими словами, любая использующаяся адресная арифметика должна быть настолько простой, чтобы можно было превратить её в непосредственный доступ к массивам.

Генерируемые модули могут включать подмодули для выполнения сложных арифметических операций, таких как умножение, деление и других. Количество подмодулей для каждого типа сложных арифметических операций может конфигурироваться. Основную часть генерируемого модуля составляет конечный автомат, который реализует заложенный в функции алгоритм. За текущее состояние автомата отвечает специальный регистр. Каждое состояние автомата соответствует набору присваиваний, которые выполняются за один такт времени. Во всех состояниях, за исключением последнего, регистру текущего состояния присваивается значение, соответствующее следующему состоянию. Так осуществляется переход из одного состояния в другое.

Конечный автомат конструируется в соответствии с графом потока управления функции. Каждый базовый блок транслируется в последовательность состояний автомата. Присваивания независимых инструкций могут выполняться параллельно в одном состоянии автомата. Если эвристика встраивания позволяет, то цепочка зависимых присваиваний может быть преобразована в одно присваивание и выполнена за один такт. Эвристики встраивания будут описаны в разделе 3.4.

Некоторые инструкции не могут быть исполнены за один такт, например инструкции чтения значения из памяти, записи значения в память или сложные арифметические операции. Таким инструкциям требуется для исполнения несколько тактов, и они используют некоторый ресурс (интерфейс блока памяти или арифметический подмодуль). Эти инструкции транслируются в присваивания нужных операндов ресурсу (на первом такте исполнения инструкции) и присваиваний, с помощью которых считывается результат выполнения ресурса (на последнем цикле исполнения инструкции).

В текущей версии C-to-HDL можно выделить следующие стадии трансляции из байт-кода LLVM в язык описания аппаратуры:

- Создание HDL-присваиваний для инструкций LLVM.
- Планирование инструкций (обычное или в случае конвейеризации).
- Генерация версий переменных в конвейеризированных циклах.
- Распределение регистров.
- Окончательная генерация конечного автомата.
- Вывод конечного автомата.

Последние 3 фазы трансляции далее не являются предметом рассмотрения настоящей статьи. Отметим, что текущим алгоритмом распределения

регистров является непосредственная взаимно-однозначная трансляция виртуальных регистров в реальные.

3.2. Генерация HDL-присваиваний

Для каждой инструкции LLVM, за исключением φ-функций, генерируется соответствующий набор HDL-присваиваний. Простые инструкции транслируются в одно соответствующее присваивание, выполняемое за один такт. Сложные арифметические операции, так же, как и инструкции чтения из памяти, требуют несколько тактов на исполнение и транслируются в два множества присваиваний. Присваивания из первого множества помещаются на первый такт исполнения инструкции, и они назначают операнды инструкции для соответствующего ресурса: адрес и режим доступа к блоку памяти в случае инструкции чтения из памяти или операнды в случае сложной арифметической операции. Второе множество состоит только из одной операции, которая сохраняет в виртуальный регистр значение из памяти или же результат сложной арифметической операции. Инструкции записи значения в память транслируются в два присваивания: сохраняемого значения и режима доступа (на запись). Инструкция перехода транслируется в одно присваивание регистру текущего состояния следующего состояния.

```
%2 = load i32* %1, align 1
соответствует операциям:
0: mem_A_mode <= 0;
0: mem_A_addr <= ltmp_1_2;
2: ltmp_2_1 <= mem_A_out;
```

(a) Чтение из памяти

```
%3 = mul i32 %2, %0
соответствует операциям:
0: mul32_ina <= ltmp_2_1;
0: mul32_inb <= ltmp_0_1;
4: ltmp_3_1 <= mul32_out;
```

(b) Умножение

```
store i32 %9, i32* %10, align 1
соответствует операциям:
0: mem_A_in <= ltmp_9_1;
0: mem_A_addr <= ltmp_10_2;
0: mem_A_mode <= 1;
```

(c) Запись в память

```
%6 = add i32 %3, %5
соответствует операции:
0: ltmp_6_1 <= (ltmp_3_1 + ltmp_5_1);
```

(d) Простая инструкция

```
br i1 %11, label %bb1, label %bb
соответствует операции:
0: eip <= (ltmp_11_3 ? bb1_start : bb_start);
```

(e) Ветвление

Рисунок 2. Пример трансляции инструкций LLVM в HDL-присваивания.

На рисунке 2 представлены примеры трансляции инструкций. Номер перед присваиванием указывает, на каком такте оно должно быть выполнено, считая

от начала выполнения инструкции. В данном примере подразумевается, что инструкция чтения выполняется за два такта, а инструкция умножения – за четыре такта.

3.3. Планирование инструкций

C-to-HDL в качестве регионов для планирования использует базовые блоки. На данный момент используется простой алгоритм для планирования в неконвейерном случае. Тем не менее, на многих примерах достигается лучшее возможное планирование. Алгоритм заключается в том, что для каждой инструкции *I* базового блока в порядке их следования в базовом блоке определяется наиболее ранний такт, на который инструкция *I* может быть помещена. Затем инструкция *I* помещается на этот такт.

Единицами планирования выступают инструкции, однако результатом планирования являются HDL-присваивания, размещённые в состояниях конечного автомата. Поэтому планирование на уровне HDL-присваиваний имеет меньше ограничений, чем обычное планирование инструкций. В частности, простые инструкции реализуются прямо из базовых элементов ПЛИС, и потому их количество ограничено только размерами кристалла. Таким образом, можно полагать, что HDL-планирование не имеет ограничение на число инструкций, выполняемых параллельно, имеется лишь ограничение на количество одновременно задействованных ресурсов для работы с памятью или для выполнения сложных арифметических операций.

Рассмотрим алгоритм планирования. В первую очередь вычисляются зависимости между инструкциями внутри базового блока (φ-функции не рассматриваются). Затем, для каждой инструкции определяется такт, на который её следует поместить. Рассмотрим подробнее определение этого такта. Если инструкция *I* не имела зависимостей, то тогда её можно поместить на такт 0. В противном случае вычисляется такт *P*, на котором все зависимости инструкции *I* будут *почти* удовлетворены. Слово *почти* означает, что все зависимости для инструкции *I* будут удовлетворены на такте *P*+1, но на такте *P* всё ещё будет выполняться некоторая инструкция, от которой зависит инструкция *I*. Если эвристика встраивания позволяет, то инструкция *I* может быть помещена на такт *P* и вычисление зависимостей по аргументам может быть подставлено вместо аргументов инструкции I. В противном случае, инструкция *I* должна быть помещена на такт *P*+1. Например, если инструкция `c=a+b` размещена на такте *P* и требуется запланировать инструкцию `e=d&c` (пусть `d` вычисляется раньше такта *P*), то, если эвристика встраивания позволяет, присваивание `e=d&(a+b)` может быть помещена на такт *P*. При вычислении такта, на который должна быть размещена инструкция, следует учитывать так же занятость использующихся ей ресурсов.

φ-функции транслируются в HDL-присваивания отличным от других инструкций способом. Для каждой φ-функции заводится виртуальный

регистр. Если φ-функция определяется в базовом блоке ВВ, то к началу выполнения ВВ требуется, чтобы регистр, отвечающий φ-функции, имел правильное значение. Для этого в каждого предшественнике РВВ базового блока ВВ на последнем такте его выполнения виртуальному регистру φ-функции присваивается значение, которое принимает φ-функция, если управление идёт из базового блока РВВ в ВВ.

3.4. Встраивание присваиваний

Рассмотрим инструкцию *I* базового блока ВВ. Пусть для инструкции *I* все зависимости почти удовлетворены на такте *P*, а также инструкция *I* имеет аргументы, вычисление которых заканчивается на такте *P*. Если эвристика встраивания позволяет (и требующиеся для инструкции ресурсы не заняты), то инструкция *I* может быть помещена на такт *P*, и присваивания на такте *P*, которые вычисляли операнды для инструкции *I*, могут быть встроены в присваивания инструкции *I*.

Рассмотрим различные эвристики встраивания. Сразу можно назвать две крайних эвристики: встраивание можно производить всегда или никогда. Если запретить встраивание, то цепочки зависимых вычислений будут занимать излишне много тактов. Если же встраивания разрешать всегда (*эвристика полного встраивания*), тогда генерируемые правые выражения присваиваний могут оказаться слишком длинными, что повлечёт деградацию частоты, на которой может работать ПЛИС.

В качестве альтернативы можно предложить *простую эвристику встраивания*. При использовании этой эвристики, операнды инструкции могут быть в неё встроены только тогда, когда получающиеся присваивания не увеличат сложность выражений в правой части. Инструкции битового сдвига с константным вторым операндам и другие битовые операции с любым константным операндом следует встраивать всегда, поскольку они транслируются в выбор соответствующих битов. Присваивания константы или виртуального регистра некоторому виртуальному регистру тоже может быть всегда встроено. Однако присваивания значения, полученного через "провод" (из интерфейса ресурса) не должны встраиваться, потому что провод уже сам по себе содержит некоторую задержку сигнала.

φ-функции транслируются в назначения соответствующему виртуальному регистру нужного значения. Поэтому аргументы φ-функции всегда могут быть встроены в соответствующее присвоение. Такое встраивание производится при любой эвристики встраивания.

4. Конвейеризация циклов

Как правило, алгоритмы тратят больше всего времени во внутренних циклах. Поэтому наибольшее внимание с точки зрения оптимизации должно быть уделено именно внутренним циклам. Конвейеризация циклов является

распространённой техникой оптимизации внутренних циклов. В отличие от программной конвейеризации циклов[5], которая была реализована в C-to-Verilog в виде оптимизации modulo scheduling, конвейеризация на уровне HDL-присваиваний имеет ряд преимуществ. В частности, конвейерное планирование на уровне HDL-присваиваний имеет те же преимущества, что и описанное выше обычное планирование базовых блоков на уровне HDL-присваиваний.

Рассмотрим подробнее алгоритм конвейеризации. При конвейеризации цикла все итерации исходного цикла имеют одинаковое планирование. Однако следующая итерация исходного цикла начинает исполняться не тогда, когда закончится предыдущая, а раньше – через количество тактов k, считая от начала исполнения предыдущей итерации исходного цикла. Это число k (количество тактов между началами соседних итераций исходного цикла) является *интервалом инициации* (initiation interval, II). В результате каждая итерация конвейеризированного цикла исполняет сразу несколько итераций исходного цикла. На итерацию в конвейеризированном цикле уходит количество тактов, равное интервалу инициации. Таким образом, ускорение при использовании оптимизации конвейеризации равно отношению количества тактов на итерацию в исходном цикле к интервалу инициации.

Для конвейеризации цикла прежде всего требуется вычислить минимально возможный интервал инициации k. Затем проводится планирование цикла для конвейерного исполнения. Это планирование отличается от обычного планирования, описанного в разделе 3.3. После планирования требуется выполнить переименование регистров.

4.1. Вычисление интервала инициации

Планирование на уровне HDL-присваиваний имеет следующий набор факторов, влияющих на интервал инициации:

1. **Число обращений к памяти и число портов памяти (для каждого массива).** На каждом такте каждый порт памяти может использовать только одна инструкция чтения или записи значения в память.
2. **Число сложных арифметических операций и количество арифметических подмодулей (для каждого типа арифметических операций).**
3. **Максимальная длина кросс-итерационных зависимостей.** При расчёте этого ограничения должна приниматься во внимание эвристика встраивания. Кросс-итерационные зависимости должны быть удовлетворены в течение интервала инициации.

Текущая реализация конвейеризации циклов применяется только к внутренним циклам с рядом ограничений. Во-первых, тело цикла должно представлять из себя только один базовый блок. Во-вторых, между φ-

функциями не должно быть циклических зависимостей. То есть, если φ-функция Ф1 использует (рекурсивно через операнды) φ-функцию Ф2, то φ-функция Ф2 не должна использовать φ-функцию Ф1 и наоборот. При соблюдении этого ограничения между φ-функциями можно установить частичный порядок. Этот частичный порядок играет важную роль в процессе планирования.

Назовём *графом кросс-итерационной зависимости* φ-функции Ф базового блока ВВ наибольший связный направленный ациклический граф со следующими свойствами: всякая инструкция базового блока ВВ, использующая (прямо или рекурсивно через операнды) φ-функцию Ф, является вершиной в этом графе; единственным истоком этого графа является инструкция, результат выполнения которой присваивается φ-функцию Ф на следующей итерации (то есть, эта инструкция является аргументом φ-функцию Ф); стоками графа могут быть только инструкции, для которых φ-функция Ф является аргументом; дуги в этом графе ведут от инструкции к её операндам. Иными словами, граф кросс-итерационной зависимости представляет из себя граф из инструкций, который показывает, как φ-функция Ф обновляет своё значение для следующей итерации. Третье ограничение в текущем алгоритме конвейеризации заключается в том, что для каждой φ-функции в теле цикла граф её кросс-итерационной зависимости должен представлять из себя просто линейную последовательность инструкций. Такую последовательность инструкций будем называть *кросс-итерационной цепочкой*.

В текущей реализации поддерживается конвейеризация циклов, в которых присутствуют как чтения, так и записи в один и тот же массив. Конвейеризация в этом случае будет работать корректно, если в цикле не происходит чтений после записи в одну и ту же ячейку массива. В противном случае чтение может вернуть старое некорректное значение, поскольку инструкция записи вследствие конвейерного исполнения может не успеть выполниться и обновить значение. Текущая версия алгоритма не поддерживает определение и восстановление после таких коллизий. Следует отметить, что реализованный в C-to-Verilog алгоритм программной конвейеризации не поддерживает конвейеризацию циклов, содержащих записи и чтения из одного массива.

Рассмотрим алгоритм вычисления интервала инициации. Три упомянутых выше ограничения обрабатываются независимо друг от друга: ограничение на доступ к памяти, ограничение на сложные арифметические операции и ограничение на кросс-итерационные зависимости. Вычисление первых двух ограничений сводится к простому подсчёту используемых ресурсов и не требует большего описания. Остановимся подробней на вычислении ограничения связанного с кросс-итерационными зависимостями. Сначала требуется найти кросс-итерационные цепочки для всех φ-функций. Затем кросс-итерационные цепочки предварительно планируются с учётом

эвристики встраивания и результатом этого предварительного планирования является минимальное количество тактов для удовлетворения кросс-итерационных зависимостей. Максимум по всем φ-функциям количества тактов для удовлетворения их кросс-итерационных зависимостей и является этим ограничением.

4.2. Планирование при конвейеризации цикла

При конвейерном планировании осуществляется планирование итерации исходного цикла таким образом, чтобы учитывать последующее конвейерное исполнение. Конвейерное планирование выполняется в терминах тактов, считая от начала выполнения итерации исходного цикла (в терминах *виртуальных тактов*). Пусть вычисленный интервал инициации равен `k`, а количество виртуальных тактов равно `n`. Для удобства будем считать, что `n` делится на `k`. Планированию требуется учитывать ряд дополнительных ограничений, таких ,что присваивания на виртуальных тактах `i` и `i+k` будут исполняться на одном реальном такте (но от разных итераций). Следовательно, требуется, чтобы в случае, если некоторая стадия ресурса используется на виртуальном такте `i`, то эта же стадия этого же ресурса не должна использоваться ни одной инструкцией на виртуальном такте `i + m*k` (`m` - целое, не равное нулю и `0 <= i + m*k < n`).

Инструкции базового блока (за исключением φ-функций) могут быть разделены на два множества: множество инструкций, которые являются частью какой-либо кросс-итерационной цепочки, и множество F остальных инструкций. Как упоминалось выше, на множестве φ-функций можно ввести частичный порядок и составить из φ-функций список L, в котором этот порядок выполняется. Таким образом, если φ-функция Ф2 зависит от Ф1, то в списке L φ-функция Ф1 будет следовать раньше Ф2.

Алгоритм планирования состоит из чередования двух фаз. Первая фаза планирует инструкции из множества F инструкций, которые ещё не были запланированы и для которых удовлетворены все зависимости. Это планирование происходит таким же образом, как было описано в разделе 3.3. Вторая фаза планирует кросс-итерационную цепочку для очередной φ-функции из списка L. Это чередование продолжается до тех пор, пока все φ-функции из списка L не будут запланированы. Затем требуется ещё раз повторить первую фазу, чтобы запланировать оставшиеся инструкции (например, инструкцию перехода).

Поскольку зависимость кросс-итерационной цепочки должна быть удовлетворена в течение интервала инициации, необходимо планировать кросс-итерационную цепочку как единое целое, учитывая предварительное планирование. Именно поэтому необходимо разделить планирование на две фазы. Присваивание виртуальному регистру φ-функции нового значения происходит на последнем такте выполнения кросс-итерационной цепочки этой φ-функции.

Поскольку в процессе конвейерного исполнения цикла значение любого виртуального регистра r, назначенного внутри цикла на такте t, становится неактуальным через k тактов (следующая итерация запишет новое значение в этот же регистр через k тактов), в случае, если регистр r используется позднее, чем через k тактов от своего назначения, требуется создать новый виртуальный регистр r', которому присваивается значение регистра r на такте t+k. Таким образом, новый регистр будет актуален с такта t+k+1 до такта t+2*k. В этом диапазоне использования регистра r заменяются на использования регистра r'. Если же имеются использования регистра r позднее такта t+2*k, то потребуется создать регистр r'' и так далее. То же самое верно и для виртуальных регистров φ-функций цикла, за исключением того, что они считаются актуальными в течение предыдущих k тактов, считая от своего назначения в цикле, поскольку их назначение является по сути обновлением значения уже для следующей итерации.

Инструкция перехода должна находиться в конце конвейеризированного цикла, поэтому её требуется помещать на виртуальный такт t так, чтобы t+1 делилось на k. Если инструкцию перехода можно выдать раньше, то требуется выравнить её по концу конвейеризированного цикла.

При планировании инструкций, использующих ресурсы, требуется учитывать конвейерное исполнение цикла. Если некоторый ресурс оказался занят на такте i, то он будет также занят и на тактах i + m*k (m - целое и 0 <= i + m*k < n).

Все инструкции записи в память должны быть помещены на последнюю итерацию конвейеризированного цикла, то есть на такты [n-t, n-1]. В этом случае можно обойтись без генерации эпилога, поскольку все инструкции, кроме инструкций записи в память, не имеют побочных эффектов, и они могут быть выполнены для любых входных данных, так как их результат не будет использован в случае использования данных за границами цикла.

φ-функции базовых блоков, в которые может перейти управление после выполнения цикла (за исключением его самого), планируются на последний виртуальный такт таким же образом, как было рассказано в разделе 3.3. Собственные φ-функции цикла планируются вместе с их кросс-итерационными цепочками, как было описано выше в данном разделе.

5. Тестирование

Для тестирования было выбрано три теста: умножение матриц, вейвлет-преобразование и алгоритм кластеризации. Тестирования проводилось при помощи симулятора Icarus Verilog[6]. Рисунок 3 показывает результаты тестирования. За основу была взята производительность тестов, оттранслированных неизменённым C-to-Verilog. График показывает ускорение тестов, в зависимости от включенных оптимизаций. C-to-Verilog использует простое встраивание присваиваний и поэтому производительность тестов, транслированных с помощью C-to-HDL без оптимизаций слегка медленней, чем транслированных через C-to-Verilog.

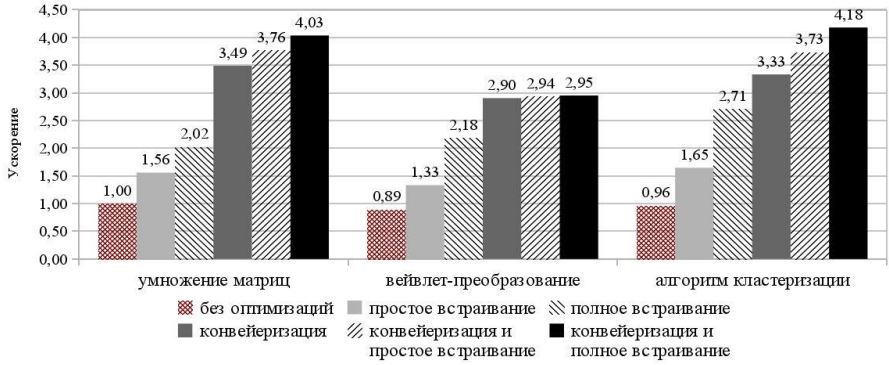


Рисунок 3. Результаты тестирования.

На диаграмме видно, что конвейеризация существенно повышает производительность тестов даже без использования оптимизации встраивания. Наилучшая производительность достигается при использовании конвейеризации и эвристики полного встраивания. Однако, как упоминалось раньше, эвристика полного встраивания может повлечь деградацию частоты работы ПЛИС. В этом случае может быть использована эвристика простого встраивания. Как видно из диаграммы, эвристика простого встраивания незначительно проигрывает эвристике полного встраивания с точки зрения количества затраченных на выполнение тактов. В то же время при трансляции с помощью эвристики простого встраивания выражения в присваиваниях будут получаться простыми и максимально возможная частота работы ПЛИС будет выше, чем в случае эвристики полного встраивания.

6. Заключение

В качестве результата работы был создан транслятор C-to-HDL. Он основывается на трансляторе C-to-Verilog, однако имеет ряд улучшений.

Главным преимуществом C-to-HDL являются реализованные эффективные оптимизации конвейеризации циклов и встраивания присваиваний. Из результатов тестирования видно, что реализованные оптимизации существенно увеличивают производительность генерируемых модулей. Ускорение, получаемое за счёт оптимизации конвейеризации циклов, в первую очередь зависит от вида этих циклов, вследствие чего на синтетических тестах можно добиться ускорения, ограниченного только размерами ПЛИС-устройства.

В будущем планируется добавить другие оптимизации (например, распределение регистров при помощи раскраски графа), планируется улучшить эвристики встраивания и реализовать детектирование и восстановление после коллизий чтения/записи в конвейеризированных циклах, в которых имеются чтения и записи в один и тот же массив. Также планируется изучить возможность трансляции с языка высокого уровня непосредственно в схему модуля аппаратуры. Такая низкоуровневая трансляция позволит непосредственно оценивать ограничения целевой архитектуры, что даст возможность делать точные оценки при встраивании и оптимизировать в целом не только с точки зрения производительности, но и с точки зрения энергопотребления и количества занимаемых ресурсов.

Список литературы

[1] Rotem N. and Asher Y. B. C to Verilog. Automating circuit design. http://c-to-verilog.com/
[2] Riverside Optimizing Compiler for Configurable Computing. http://www.jacquardcomputing.com/roccc/
[3] The LLVM Compiler Infrastructure. http://llvm.org/
[4] The SUIF Compiler System. http://suif.stanford.edu/suif/
[5] Lam M. S. Software pipelining: an effective scheduling technique for vliw machines http://doi.acm.org/10.1145/989393.989420
[6] Icarus Verilog. http://iverilog.icarus.com/

Implementation of Loop Pipelining and Assignment Inlining in the C-to-HDL Translator

Alexey Merkulov <steelart@ispras.ru> ISP RAS, Moscow, Russia
Andrey Belevantsev <abel@ispras.ru> ISP RAS, Moscow, Russia

Abstract. Implementing algorithms for field-programmable gate arrays using a hardware description language is a complex task. Therefore, it would be useful to have a tool that can efficiently translate an algorithm from a high-level language to a hardware description language. In this paper we consider the C-to-HDL which can translate C functions into Verilog modules, the translation process, and two important optimizations implemented on hardware description level: assignment inlining and loop pipelining. The basic structure and ABI interface of C-to-HDL follows that of the C-to-Verilog translator which inspired us for creating our tool. We also use LLVM infrastructure for translating LLVM bitcode representation into the Verilog code. Simple arithmetic operations are executed within a single cycle, while for complex arithmetic operations and loads/stores from/to memory, first, a set of assignments loading instruction operands, memory address, and memory access mode (read or write) is generated and placed on the first cycle of executing the instructions, and second, the final assignment transferring the operation result to a virtual register is generated.

The following optimizations are implemented and greatly improve the execution performance. Instruction scheduling is performed, taking into account that the number of instructions executed in parallel is only limited by the FPGA free chip space; the memory operations have limits given by the number of memory channels and blocks on the FPGA. If possible, assignments of temporary registers are inlined to make the more complex operations, but without producing too long dependence chains and unwanted limiting of the FPGA frequency. Finally, software pipelining following the usual modulo scheduling scheme is also performed with the same resource constraint limit relaxations as described for instruction scheduling.

Experimental results demonstrate that these optimizations significantly improve (up to 4x) the performance of generated code.

Keywords: FPGA, Verilog, automatic translation, assignment inlining, loop pipelining.

References

[1]. Rotem N. and Asher Y. B. C to Verilog. Automating circuit design. http://c-to-verilog.com/
[2]. Riverside Optimizing Compiler for Configurable Computing, http://www.jacquardcomputing.com/roccc
[3]. The LLVM Compiler Infrastructure, http://llvm.org
[4]. The SUIF Compiler System, http://suif.stanford.edu/suif
[5]. Lam M. S. Software pipelining: an effective scheduling technique for VLIW machines. SIGPLAN Not. 39, 4 (April 2004), 244-256. DOI=10.1145/989393.989420
[6]. Icarus Verilog, http://iverilog.icarus.com

Динамический анализ обфусцированных приложений с диспетчеризацией или виртуализацией кода [1]

М.Г. Бакулин, С.С. Гайсарян, Ш.Ф. Курмангалеев, И.Н. Ледовских, В.А. Падарян, С.М. Щевьева
{bakulinm, ssg, kursh, il, vartan, shchsveta}@ispras.ru

Аннотация. В данной статье приводятся результаты экспериментального исследования по восстановлению графа потока управления, запутанного специализированным компилятором на основе LLVM. Средства запутывания и распутывания бинарного кода независимо разрабатывались двумя коллективами ИСП РАН. Помимо того, для количественной оценки стойкости запутывания были получены метрики сложности кода модельных примеров.

Ключевые слова. Динамический анализ, виртуальная машина, диспетчер, метрики кода.

1. Введение

В настоящее время, специалисты, занимающиеся обратной инженерией бинарного кода, регулярно сталкиваются со сложными системами защиты от анализа. Среди них все чаще используются методы запутывания бинарного кода, способные значительно затормозить процесс восстановления алгоритмов и форматов данных. Такой защитой оснащают не только вредоносное ПО, например, включенный в состав Visual Studio компонент Dotfuscator[2] позволяет рядовому разработчику защитить свои алгоритмы инструментом, интегрированным с базовыми средствами разработки.

Распространение средств запутывания кода делает все более важным техническое оснащение специалистов по обратной инженерии[1]. Ключевую роль в работе играют программные инструменты, способные качественно снизить сложность анализа, позволяющие его провести в разумные временные рамки.

[1] Работа поддержана грантом Президента Российской Федерации для государственной поддержки молодых российских ученых-кандидатов наук МК-1281.2012.9 и грантами РФФИ 12-01-31417 мол_а, 11-07-00450-а.

[2] http://msdn.microsoft.com/en-us/library/ms227240.aspx

Существует два основных подхода к анализу кода – статический и динамический. При динамическом подходе некоторые виды защиты (например, упаковка или шифрование, а также самомодификация кода) влияют на сложность анализа в меньшей степени, чем при статическом. С другой стороны, динамический анализ не может обеспечить полноту покрытия анализируемого приложения. В силу того, что у обоих подходов имеются свои преимущества и недостатки, часто применяется комбинированный метод анализа. В данной статье в качестве инструмента восстановления алгоритма рассматривается система TrEx [2] и метод её практического применения при анализе запутанного кода.

2. Система динамического анализа TrEx

В Институте системного программирования РАН разрабатывается система анализа бинарного кода TrEx. Основным источником данных выступает набор трасс, полученных с общей временной точки при выполнении исследуемой программой типичных сценариев работы. Под трассой понимается последовательность инструкций, выполненных процессором во время выполнения на нём интересующей аналитика программы, а так же значения всех регистров перед выполнением каждой инструкции. Трассы снимаются при помощи полносистемных эмуляторов, таких, например, как эмулятор QEMU[3].

Среда TrEx предоставляет различные возможности для упрощения анализа: разметку шагов трассы в соответствии с тем, какой процесс/нить выполняется на данном шаге, доступ к информации о системных вызовах и вызовах библиотечных функций, динамический прямой и обратный слайсинг трассы[4], построение различных графов анализируемой программы (зависимостей, вызовов, потока управления), восстановление формата данных[5, 6], а также многое другое.

В случае запутанного кода построение, например, графа потока управления может оказаться малополезным для распознавания алгоритма, поскольку различные преобразования запутывания управления не только увеличивают количество вершин и дуг графа, но и существенно меняют его общий вид. Наиболее показательные примеры таких преобразований – диспетчер и виртуализация кода. Для преодоления этих видов запутывания разработан компонент системы TrEx, способный восстанавливать граф потока управления кода, который был защищён наложением виртуальной машины. При таком методе защиты часть кода приложения транслируется в байт-код сгенерированной виртуальной машины, после чего защищаемый фрагмент заменяется интерпретатором этого байт-кода[7]. Многие интерпретаторы виртуальных машин работают по такой схеме: 1) считывание кода операции из памяти; 2) вычисление адреса обработчика (блока интерпретации) этого кода; 3) передача управления (диспетчеризация) и исполнение блока интерпретации данного кода операции; 4) переход к считыванию следующего

кода операции. Как правило, у такого эмулятора имеется счетчик команд виртуальной машины (VPC – Virtual Program Counter), который определяет, из какой области памяти необходимо считывать следующий код операции. Метод восстановления графа основывается на том, какие значения принимает этот счетчик команд виртуальной машины во время исполнения.

Если известны все шаги трассы, соответствующие считыванию кода операции из памяти и для каждого из них известно значение VPC, можно попытаться восстановить граф, используя следующий метод. Если инструкция – считывание кода операции, то в строящийся граф добавляется (если её в нем ещё нет) вершина с номером, равным значению счетчика команд, а также добавляется ребро $a \rightarrow b$, где a – значение счетчика команд при предыдущем считывании кода операции, а b – значение на текущем шаге. После этого граф требуется упростить, заменив длинные цепочки вершин на одну. Делается это так: если у вершины ровно один предшественник и один последователь, при этом у её предшественника только один последователь, вершина и инцидентные ей рёбра удаляются, и добавляется ребро из её предшественника в её последователя. Полученный граф выводится в формате .dot и может быть нарисован при помощи инструмента graphviz[8]

Предложенный способ хорошо себя показал при восстановлении графа потока управления тестовых приложений, защищенных демонстрационными версиями обфускаторов VMProtect[3], CodeVirtualizer[4], Safengine[5], и The Enigma Protector[6] [9].

3. Построение обфусцирующего компилятора на основе инфраструктуры LLVM

Для защиты бинарного кода от анализа используется множество различных методов, один из них – запутывающие преобразования. Такие преобразования обычно вносятся при помощи автоматических обфускаторов, которые принимают на вход исходный код программы или бинарный файл, а на выходе предоставляют исполняемый файл с запутанной программой.

Оптимальным выбором, позволяющим реализовать стойкие варианты запутывания программ, является создание обфусцирующего компилятора на базе одной из существующих компиляторных инфраструктур[10]. С одной стороны, это позволяет производить запутывание программы, имея полную информацию о ней на всех этапах компиляции, с другой – позволяет сосредоточиться на разработке алгоритмов защиты, а не на создании требуемой инфраструктуры. Преобразования могут производиться во время

[3] http://vmpsoft.com/products/vmprotect
[4] http://www.oreans.com/codevirtualizer.php
[5] http://enigmaprotector.com/ru/about.html
[6] http://safengine.com/en-us

обработки промежуточного представления компилятора на машинно-независимом уровне, что позволяет обеспечить поддержку нескольких архитектур

В Институте системного программирования был разработан обфусцирующий компилятор на основе инфраструктуры LLVM [11]. При разработке преобразований учитывались критерии эффективности:

- Запутывающее преобразование должно затрагивать и поток управления и поток данных запутываемой программы;
- Стойкость преобразования должна основываться на алгоритмически сложных задачах, например, требовать от атакующего применения анализа указателей для точного восстановления потоков данных защищенной программы;
- При разработке преобразования нужно учитывать особенности работы средств анализа, например, для автоматических декомпиляторов следует насытить граф потока управления несводимыми участками.

Разработанные методы усложнения программного кода:

- Преобразование, перемещающее локальные переменные в глобальную область видимости;
- Защита константных строк, используемых программой;
- Вставка в код фиктивных циклов;
- Приведение графа потока управления к плоскому виду с применением алгоритма диспетчеризации;
- Переплетение нескольких функций в одну с заменой всех вызовов отдельных функций на вызов одной общей;
- Сокрытие вызовов функций. Для защищаемой функции создается функция-переходник, внутри которой содержится несколько вызовов различных функций. Вызов нужной функции определяется с помощью предиката;
- Запутывание графа потока управления – создание несводимых участков в графе;
- Замена вызовов одной и той же функции на вызовы ее копий.

При помощи среды TrEx было проведено исследование разработанных методов защиты кода для оценки эффективности полученного обфусктора.

4. Восстановление графа потока управления

Анализ кода, полученного от запутывающего компилятора, проходил по схеме слепого эксперимента: аналитики использовали обфскатор как «черный ящик», не располагая информацией о том, какие запутывающие преобразования будут проделываться над кодом.

Для изучения особенностей реализации методов запутывания разработанным обфускатором был использовано один модельный алгоритм (Рис. 1), тремя различными способами переведенный в форму исполняемого кода.

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define n 10
04 void bubbleSort(int numbers[], int array_size) {
05       int i, j, temp;
06       for (i = 0; i < array_size; i++) {
07             for (j = 0; j < array_size - 1; j++) {
08                   if (numbers[j] > numbers[j + 1])
09 {
10                         temp = numbers[j + 1];
11                         numbers[j + 1] =
12 numbers[j];
13                         numbers[j] = temp;
14                   }
15             }
16       }
17 }
18 int main() {
19       int a[n];
20       printf("Start\n");
21       for (int i = 0; i < n; i++) {
22             a[i] = rand()*rand();
23             printf("%d ", a[i]);
24       }
25       printf("\n");
26       bubbleSort(a, n);
27       for (int i = 0; i < n; i++) {
28             printf("%d ", a[i]);
29       }
30       printf("Finish\n");
         return 0
   }
```

Рис. 1. Листинг программы сортировки пузырьком.

Первый тестовый пример был получен прототипной версией запутывающего компилятора, второй – с помощью его же итоговой версии. Третий пример был получен итоговой версией компилятора, но на модифицированной версии кода. Изменения никак не меняли сам алгоритм и его граф потока управления в частности, но препятствовали применению некоторых

общеупотребительных оптимизаций. Для всех случаев программа компилировалась с параметрами, включающими на максимальную защиту.

Полученный исполняемый файл первого примера был запущен при помощи эмулятора QEMU, в результате была снята трасса программы. В процессе трассировки программа вывела на печать следующую информацию:

```
Start
1323961443 1092212534 1392800464 -818133696 1041837016 -
628591679 -1034175621 609956508 1082926972 -1134093940
-1134093940 -1034175621 -818133696 -628591679 609956508
1041837016 1082926972 1092212534 1323961443 1392800464
Finish
```

Первым шагом при исследовании трассы обычно является нахождение инструкций «маяков», которые относятся к изучаемому приложению[12]. Для этого существует несколько подходов. В данном случае был использован поиск в трассе определенного состояния процессора: один из регистров общего назначения содержит числа, выводящиеся на экран приложением. Был найден первый шаг трассы, на котором значение регистра стало равно 1041837016 (одно из чисел, выведенных на печать). Номер этого шага равен `0x98FA1C`[7], номер процесса, соответствующего этому шагу, равен `0x18A8000`, адрес инструкции `0x413B2C`. Также в непосредственной близости (на шаге `0x98F8F4`) был обнаружен вызов функции `rand`, которая, как известно из исходного кода, используется для вычисления значений элементов массива в процессе инициализации. На основании полученной информации был сделан вывод, что данный шаг и процесс соответствует коду исследуемого приложения. После этого была выделена подтрасса, в которой содержатся только те шаги исходной трассы, которые относятся к тому же процессу и тому же адресному пространству, что и шаг `0x98FA1C`. В такой трассе содержатся только те инструкции, которые относятся либо непосредственно к исполняемому файлу изучаемого приложения, либо к различным библиотечным функциям (таким, как `printf`, `rand`), которые были вызваны из исследуемого приложения.

Для дальнейшего уточнения положения функции сортировки пузырьком в трассе были использованы вызовы функции печати на экран. В случае с данным приложением для печати использовалась функция `printf` из библиотеки cygwin1.dll. Адрес начала этой функции равен `0x610C4ADF`. Согласно графу вызовов, в трассе присутствует 23 вызова этой функции,

[7] Для нахождения такого шага использовался модуль поиска состояния, встроенный в систему TrEx, который находит первый шаг, для которого выполняется заданное условие. В данном случае условие было `EAX == 1041837016 || EBX == 1041837016 || ECX == 1041837016 || EDX == 1041837016`.

поэтому можно предположить, что первый вызов относится к печати сообщения “Start” (строка 18 в исходном коде), следующие 10 вызовов соответствуют печати чисел в неотсортированном массиве (строка 21), следующий вызов соответствует печати символа перевода строки (строка 23), ещё 10 вызовов соответствуют печати отсортированного массива (строка 26) и, наконец, последний вызов осуществляет печать сообщения “Finish” (строка 28). Поэтому была создана ещё одна подтрасса, в которую вошли только шаги от двенадцатого до тринадцатого вызова функции `printf`. В полученной трассе содержится примерно 250 тысяч шагов, в то время как в исходной трассе их больше 20 миллионов.

Обычно, аналитик не располагает исходным кодом программы, поэтому для нахождения необходимого вызова системной функции может быть использован и другой способ. Для каждого вызова функции, например, `printf`, восстанавливаются параметры, с которым она была вызвана. Известно, что функция `printf` принимает несколько параметров, и все параметры лежат на стеке. Если был произведён вызов функции `printf("%d", a[i])`, то на шаге, соответствующем этому вызову, по адресу `ESP + 4` в памяти будет находиться адрес форматной строки, а по адресу `ESP + 8` будет находиться значение переменной `a[i]`, которое выводится на печать. Пусть необходимо восстановить фактические параметры вызова функции, произошедшего в основной трассе на шаге с номером `0xAA632C` (этот шаг соответствует одиннадцатому вызову `printf` в трассе). Для этого используется имеющийся в системе TrEx компонент Buffer Restore, который на основе обращений к участку памяти может установить его значение. Простейший пример такого обращения: если после инструкции `MOV EAX, DWORD PTR[0x226B2C]` значение регистра `EAX` стало равно `0x42735D`, то, очевидно, по адресу `0x226B2C` в памяти находится число `0x42735D`. Алгоритм восстановления значения, хранящегося в памяти по адресу `ESP + 4`, вернул результат `0x499007`. После этого был запущен алгоритм восстановления буфера по адресу `0x499007`, была получена строка `"%d "`, то есть данный вызов относится к печати числа. Само число, которое выводится на печать, расположено в памяти по адресу `ESP + 8`. При помощи алгоритма восстановления буфера было установлено, что по этому адресу в памяти находится число `0xBC67198C` (или -1134093940 в десятичной системе). Таким образом, одиннадцатый вызов функции `printf` в трассе соответствует печати числа -1134093940. Аналогичным образом можно восстановить информацию и о других вызовах функции `printf` в трассе.

В результате фильтрации была получена подтрасса, в которой содержится в основном только код функции, которую необходимо проанализировать. После этого был запущен поиск шага выделенной подтрассы, на котором какой-либо из регистров общего назначения принял значение, равное одному из выведенных на печать чисел. Значение 1092212534 встретилось на одном из

регистров на шаге 0xF0D4. В непосредственной близости на шаге 0xF0D1 находится инструкция перехода по динамически вычисляемому адресу JMP ECX. Такие инструкции – характерная черта диспетчеров и эмуляторов виртуальных машин. В программах, запутанных другими виртуализаторами, возможно использование и других подобных инструкций, например JMP DWORD PTR [EAX*4 + EDI], или последовательность инструкций PUSH ECX, RET и тому подобные. После нахождения такой инструкции перехода необходимо определить, как был вычислен адрес перехода и определить, что является счётчиком команд виртуальной машины. Участок трассы, соответствующий вычислению адреса перехода, приведён на рисунке 2.

```
00404846    MOV   DWORD PTR SS:[EBP - 0000183Ch], ESI
0040484C    JA    0040480Ch
0040484E    MOV   EAX, DWORD PTR SS:[EBP - 0000183Ch]
00404854    MOV   ECX, DWORD PTR [EAX * 4 + 00499140h]
0040485B    JMP   ECX
```

Рис. 2. Участок трассы первого примера. Вычисление адреса перехода.

Можно видеть, что адрес перехода равен значению ячейки памяти по адресу ESI*4 + 0x499140. Таким образом, можно предположить, что счетчиком команд виртуальной машины служит регистр ESI. Для проверки этого предположения был запущен алгоритм восстановления графа потока управления виртуализованного кода. На вход алгоритм принимает условие, которое соответствует считыванию кода операции из памяти (в данном случае, EIP == 0x404854[8]) и выражение, которое соответствует значению счётчика команд виртуальной машины при данном считывании (ESI*4 + 0x499140). В результате получен граф, изображённый на рисунке 3.

[8] В некоторых случаях требуется чуть более сложное условие. Например, для программ, защищенных при помощи Safengine, условие имеет вид EIP == a && EBP == b, где a и b – константы, определяемые вручную по трассе аналогично тому, как был найден адрес 0x404854.

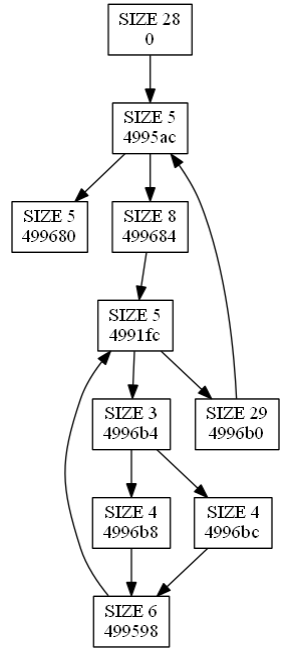


Рис. 3. Граф потока управления, полученный по трассе первого приложения.

Каждой вершине в графе соответствует два числа: размер и адрес. При построении графа длинные цепочки вершин заменялись на одну вершину. Наличие таких цепочек позволяет заключить о разбиении исходных базовых блоков на более мелкие при применении запутывания «диспетчер». Первое число соответствует количеству вершин в цепочке, которую замещает данная вершина, а второе соответствует адресу, из которого считывается адрес первого перехода. Для первого базового блока адрес считается равным нулю. На этом рисунке отчетливо видны два цикла: внешний и внутренний, во внутреннем цикле существует какое-то условие, в зависимости от которого выполняется один или другой базовый блок. Именно такой граф потока управления характерен для сортировки пузырьком, что позволяет предположить об успешности восстановления. Благодаря информации, вынесенной в вершины графа, можно приближённо разметить трассу в соответствии с тем, к какому базовому блоку относится тот или иной участок трассы. В случае данного исследуемого приложения для этого необходимо создать текстовые комментарии к позициям трассы, для которых выполняется условие `EIP == 0x404854 && ESI*4 + 0x499140 == A`, где `A` – адрес, соответствующий какому-либо базовому блоку восстановленного графа. Тогда участок трассы между двумя комментариями будет соответствовать одному базовому блоку. При этом, конечно, на этом участке

будут встречаться не только инструкции, относящиеся к алгоритму, который был защищён запутыванием, но также и управляющие инструкции виртуальной машины или диспетчера.

Во втором примере исходный Си-код был защищен итоговой версией обфускатора. Для нахождения участка трассы, содержащего код сортировки, был использован тот же метод, что и в предыдущем случае. Полученная подтрасса содержит примерно 140 тысяч шагов. Точно так же в подтрассе был найден шаг, на котором один из регистров содержит значение 1092212534, и рядом с этим шагом был найден переход по динамически вычисляемому адресу. Участок трассы, на котором происходит вычисление адреса перехода, изображён на рисунке 4.

```
0041C95C    MOV   EAX, DWORD PTR SS:[EBP -
00002B10h]
0041C962    XOR   EAX, DWORD PTR SS:[EBP -
00002B20h]
0041C968    MOV   EDX, DWORD PTR SS:[EBP -
00002B18h]
0041C96E    XOR   EAX, EDX
0041C970    MOV   ECX, DWORD PTR SS:[EBP -
00002B1Ch]
0041C976    XOR   ECX, EDX
0041C978    XOR   EDX, DWORD PTR SS:[EBP -
00002B14h]
0041C97E    MOV   DWORD PTR SS:[EBP - 00002B18h],
EDX
0041C984    ADD   ECX, EAX
0041C986    MOV   DWORD PTR SS:[EBP - 00002B0Ch],
ECX
0041C98C    CMP   ECX, 000003F1h
0041C992    JA    0041C95Ch
0041C994    JMP   DWORD PTR [ECX * 4 + 005CAF5Ch]
```

Рис. 4. Участок трассы второго приложения. Вычисление адреса перехода.

Видно, что отсутствует четко выраженная конструкция, реализующая виртуальный счётчик команд. Была произведена попытка автоматически восстановить граф со следующими параметрами: адрес инструкции считывания кода операции `0x41C994`, регистр со значением счётчика команд – `ECX`. В результате был получен граф, изображённый на рисунке 5.

Рис. 5. Граф потока управления, полученный по трассе второго примера

Такой граф явно не соответствует ожидаемому результату. Были проведены дополнительные исследования, с целью выяснить, что же в действительности используется в качестве счётчика команд виртуальной машины в данном случае. Однако, несмотря на достаточно долгое изучение, выявить дополнительный код, реализующий виртуальный счетчик команд, никак не удавалось. Тогда было сделано предположение, что оптимизирующий компилятор обнаружил в коде константу, которая задавала размер массива, выполнил распространение констант по коду и развернул внутренний цикл сортировки[9]. Действительно, можно видеть, что в графе во внутреннем цикле имеется 9 ветвлений. Для проверки предположения, что некорректный граф получается из-за оптимизаций компилятора, исходный код был изменён таким образом, чтобы размер массива задавался пользователем с клавиатуры. Полученный код был скомпилирован итоговой версией запутывающего компилятора, после чего снята трасса третьего примера. В результате выделения участка трассы, соответствующего функции сортировки, была получена подтрасса размером примерно 217 тысяч шагов.

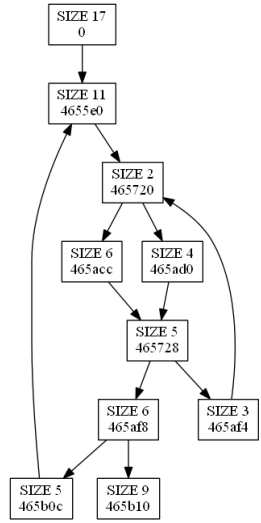


Рис. 6. Граф потока управления, полученный по трассе третьего примера

[9] Уже после снятия трассы третьего примера был скомпилирован код исходной программы без применения запутывающих преобразований, и в графе потока управления полученной функции точно так же присутствовали 9 ветвлений. Отсутствие развёрнутого цикла в графе первого примера связано, по-видимому, с другими настройками компиляции в прототипной версии запутывающего компилятора.

Код вычисления адреса перехода в третьем приложении получился такой же, как во второй версии с точностью до констант. Был применён алгоритм восстановления графа потока управления, результат изображён на рисунке 5. Как можно видеть, граф восстановлен достаточно точно: видно два цикла (внешний и внутренний), и во внутреннем цикле исполняется один из двух базовых блоков в зависимости от выполнения или не выполнения условия, но при этом цикл с предусловием был, по-видимому, заменён на цикл с постусловием.

5. Оценка сложности сгенерированного кода на основе метрик

Вычисление метрик программного кода позволяет не только оценивать различные свойства программ, прогнозировать объем работ, характеризовать проектные решения, сложность и надежность программного обеспечения в процессе разработки и сопровождения. Использование метрик целесообразно и при решении задач анализа приложений – в частности, применительно к бинарному коду. В процессе исследования бинарного кода требуется уметь классифицировать приложения по сложности, оценивать трудоемкость анализа как всей программы, так и ее отдельных частей (модулей, функций), а также находить в коде участки, представляющие первоочередной интерес для аналитика и потому, как правило, защищенные различными методами запутывания. Эти задачи предъявляют несколько иные требования к выбору метрик кода, чем процесс разработки и модификации приложений. В настоящее время в рамках среды TrEx создаётся компонент для вычисления различных метрических характеристик анализируемого кода. В ходе выполнения описываемой работы были выполнены первые эксперименты по получению и использованию метрик сложности бинарного кода в среде TrEx.

Выделяют несколько групп метрик сложности кода: количественные метрики, метрики сложности потока управления, метрики сложности данных, комбинированные, гибридные и объектно-ориентированные метрики. В анализе двоичного кода можно использовать лишь часть известных метрик из числа количественных, метрик сложности потоков управления и данных, а также получаемых на их основе гибридных мер. Количественные метрики, изначально используемые для оценки трудозатрат по проектам, основаны на подсчёте различных конструкций исходного или бинарного кода. Они легко могут быть адаптированы для анализа трасс программ. Количественные характеристики программ обычно рассматриваются в первую очередь, ввиду простоты их получения – для вычисления этих метрик не требуется предварительный анализ кода или трасс, построение графов потока управления, потока данных, вызовов и т.п.

В первых экспериментах использовались количественные метрики, позволяющие оценить насыщенность программы управляющими конструкциями – метрика Джилба и BCD-метрика. Полученные оценки

корректировались посредством аналога метрики LOC (Lines Of Code) – для запутанного бинарного кода в качестве такого аналога LOC был взят размер исполняемого файла. Метрики Джилба [13] показывают сложность программного обеспечения на основе насыщенности программы условными операторами или операторами цикла. Вычисляются два значения: абсолютная сложность CL (количество управляющих операторов) и относительная сложность по Джилбу cl=CL/n, где n – общее число операторов программы. BCD-метрика является модификацией ABC-метрики [14], которая основана на подсчете присваиваний (**A**ssignment), передач управления за пределы области видимости, т.е. вызовов функций (**B**ranch), и логических проверок (**C**ondition). Мера записывается тройкой значений, например, ABC = <7, 4, 2>, но для оценки сложности программы вычисляется одно число, как квадратный корень из суммы квадратов A, B, C. Модификация этой метрики, названная BCD (**B**ranch and **C**all **D**ensity) сводилась к вычислению лишь двух значений – B (количество инструкций условных и безусловных переходов) и C (количество инструкций вызова). Затем вычисляются плотности этих инструкций BD=B/n и CD=C/n. Результирующее значение для характеристики сложности программы получается как квадратный корень из суммы квадратов BD и CD.

По трассам в среде TrEx были получены значения метрик для незащищенной программы сортировки пузырьком и трёх вариантов обфускации на основе инфраструктуры LLVM. Результаты сведены в таблицу 1; для сравнения приводятся также характеристики этого же приложения, защищенного демо-версией обфускатора Safengine.

Можно видеть, что относительная сложность анализируемых тестовых примеров как по Джилбу, так и на основе BCD-метрики, после наложения запутывающих преобразований не только не увеличилась, но даже стала заметно меньше (за исключением значения BCD-меры в случае Safengine). Данный факт объясняется тем, что все примененные обфускаторы применяют к программе ряд запутывающих преобразований помимо запутывания потока управления посредством диспетчера. В частности, добавляется значительное количество непрозрачных предикатов, недостижимого и мёртвого кода; применяются также различные виды запутывания данных. Эти преобразования увеличивают количество инструкций в коде программы, что приводит к уменьшению относительных оценок сложности потока управления на основе количественных метрик.

Таблица 1. Относительная сложность кода

Программа	**cl**	**BCD**
Без защиты	0,213607	0,190239
Защищена Safengine	0,206598	0,193034

Пример 1	0,141131	0,131036
Пример 2	0,18671	0,176082
Пример 3	0,171154	0,159457

Для получения адекватной оценки сложности восстановления потока управления было принято решение использовать комбинированные количественные метрики, представляющие собой произведение размера исполняемого файла в килобайтах (аналог метрики LOC для бинарного кода) на значения метрик Джилба и BCD.

Такая оценка, как можно видеть из таблицы 2, более точно соответствует реальным затратам времени на анализ и восстановление потока управления использованного нами запутанной программы. Особенно наглядно это проявляется для оценок примеров № 1 и №2; пример №3, для которого исходный текст программы был изменен с целью исключения ряда оптимизирующих преобразований, закономерно получил более низкую оценку сложности, несмотря на то, что значения относительных мер для него ближе к их значениям для примера №2. Код с защитой Safengine имеет меньшие комбинированные оценки сложности при больших относительных; это объясняется тем, что демонстрационная версия данного протектора имеет ряд ограничений, в частности, по запутыванию потока данных. В целом, результаты данного эксперимента свидетельствуют о возможности предварительной оценки сложности анализа защищенного кода посредством простых количественных метрик и их комбинаций.

Таблица 2. Сложность кода с поправкой на размер

Программа	**Размер, Кб**	**LOC*cl**	**LOC*BCD**
Без защиты	134	28.6233	25.492
Защищена Safengine	194	40.08	37.4486
Пример 1	630	88.9125	82.55
Пример 2	1893	353.442	305.6
Пример 3	414	70.858	66.015

6. Заключение

Предложенный метод восстановления графа потока управления виртуализованного кода показал свою эффективность, однако имеет и недостатки. Один из них напрямую связан с природой динамического анализа: если при запуске программы одна из веток условного перехода ни разу не

выполнялась, то в восстановленном графе не будет соответствующего ветвления. Чтобы нивелировать этот недостаток предполагается дополнить метод следующим образом: если какой-либо базовый блок имеет больше одного последователя, то код операции (не счётчик команд!), соответствующий последнему исполненному переходу по вычислимому адресу в этом базовом блоке, помечается как код условного перехода, и все другие блоки, в которых был считан такой же код, тоже будут помечаться как блоки, имеющие возможный условный переход. Однако для определения цели этого невыполнившегося перехода будет необходимо полностью восстановить алгоритм работы данной виртуальной машины.

Следующий недостаток связан с тем, что для корректного восстановления графа необходимо вручную найти нужные инструкции перехода и счётчик команд виртуальной машины. В трассе может быть несколько таких инструкций с различными адресами. Например, у каждой функции приложения может быть свой собственный интерпретатор команд виртуальной машины, который использует свой собственный счётчик команд, и если неправильно определить, какая из этих инструкций перехода по вычисляемому адресу относится к изучаемой функции, можно получить некорректный граф. Возможна и другая ситуация: для передачи управления во всех функциях используется одна общая инструкция перехода, и в результате будет получен граф, представляющий смесь графа вызовов и потока управления. Дополнительную сложность при этом создаёт то, что для определения ошибочности восстановленного графа аналитик может ориентироваться только на своё чутьё и ожидания того, какой граф должен быть получен. В будущем планируется добавить модуль, собирающий статистику обо всех переходах по вычисляемым адресам и предоставляющий её пользователю для облегчения поиска нужного перехода.

Отдельно стоит заметить, что можно восстановить только граф потока управления той программы, которая была подана на вход запутывателю. Таким образом, если были проведены предварительные дополнительные запутывания потока управления (независимо от того, намеренные это усложнения или просто оптимизирующие преобразования компилятора), получить исходный граф представляется невозможным. Для борьбы с некоторыми из запутываний (в частности, с разворачиванием циклов и клонированием кода) необходимо уметь составлять модели поведения полученных базовых блоков и сравнивать полученные модели между собой, но это достаточно сложная задача для автоматического анализа.

Не стоит забывать, что восстановленный граф потока управления может только помочь аналитику в восстановлении алгоритма исследуемой программы. Аналитику требуется ещё установить, что делает каждый из базовых блоков, а в случае с кодом, запутанным посредством комбинации различных методов, это может оказаться достаточно сложной задачей. Исследование этих задач и разработка соответствующего инструментария

является направлением дальнейшей работы в рамках развития среды динамического анализа TrEx.

Список литературы

[1] А.Ю.Тихонов, А.И. Аветисян. Развитие taint-анализа для решения задачи поиска программных закладок // Труды Института Системного Программирования. — 2012. — Т. 22. — С. 9–24.

[2] Падарян В. А., Гетьман А. И., Соловьёв М. А. Программная среда для динамического анализа бинарного кода // Труды Института Системного Программирования. — 2009. — Т. 16. — С. 51–72.

[3] К. Батузов, П. Довгалюк, В. Кошелев, В. Падарян. Два способа организации механизма полносистемного детерминированного воспроизведения в симуляторе QEMU // Труды Института Системного Программирования. — 2012. — Т. 22. — С. 77–94.

[4] Korel B., Laski J. Dynamic program slicing // Information Processing Letters, Vol. 29, Issue 3. — 1988. — P. 155–163.

[5] А.И. Гетьман, Ю.В. Маркин, В.А. Падарян, Е.И. Щетинин. Восстановление формата данных // Труды Института Системного Программирования. — 2010. — Т. 19. — С. 195–214.

[6] А.И. Аветисян, А.И. Гетьман. Восстановление структуры бинарных данных по трассам программ // Труды Института Системного Программирования. — 2012. — Т. 22. — С. 95–118.

[7] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, Wenke Lee. Automatic reverse engineering of malware emulators // Proceedings of the 2009 30th IEEE Symposium on Security and Privacy. — SP ’09. — Washington, DC, USA : IEEE Computer Society, 2009. — P. 94–109.

[8] J. Ellson, E.R. Gansner, E. Koutsofios et al. Graphviz and dynagraph – static and dynamic graph drawing tools // Graph Drawing Software / Ed. by M. Junger, P. Mutzel. — Berlin/Heidelberg : Springer-Verlag, 2004. — Mathematics and Visualization. — P. 127–148.

[9] И.Н.Ледовских, М.Г.Бакулин. Подход к восстановлению потока управления запутанной программы // Труды Института Системного Программирования. — 2012. — Т. 22. — С. 155–167.

[10] Курмангалеев Ш.Ф., Корчагин В.П., Матевосян Р.А. Описание подхода к разработке обфусцирующего компилятора // Труды Института Системного Программирования. — 2012. — Т. 23. 10 страниц. Принято к печати

[11] Курмангалеев Ш.Ф., Корчагин В.П., Савченко В.В., Саргсян С.С. Построение обфусцирующего компилятора на основе инфраструктуры LLVM // Труды Института Системного Программирования. — 2012. — Т. 23. 15 страниц. Принято к печати.

[12] Тихонов А. Ю., Аветисян А. И., Падарян В. А. Методика извлечения алгоритма из бинарного кода на основе динамического анализа // Проблемы информационной безопасности. Компьютерные системы. — 2008. — Т. 3. — С. 66–71.

[13] Милютин А., Метрики кода программного обеспечения http://www.viva64.com/ru/a/0045/ . Дата публикации: 20.07.2009.

[14] Hassan Raza Bhatti, Automatic Measurement of Source Code Complexity // Master’s Thesis, Lulea University of Technology, Lulea, Sweden, 2011

Dynamic analysis of virtualization- and dispatching-obfuscated applications

M.G. Bakulin, S.S. Gaissaryan, Sh.F. Kurmangaleev,
I.N. Ledovskikh, V.A. Padaryan, S.M. Shchevyeva
ISP RAS, Moscow, Russia
{bakulinm, ssg, kursh, il, vartan, shchsveta}@ispras.ru

Abstract. Obfuscation algorithms are now widely used to prevent software reverse engineering. Binary code virtualization is one of the most powerful obfuscations technics. Another obfuscation method known as “dispatching” can be used to transform application control flow similarly to virtual machine insertion. Our research was aimed at reconstruction of control flow graph in case of both code virtualization and dispatching. To achieve this goal, we implemented de-obfuscation tool which keeps track of virtual program counter used by virtual machine emulator and reconstructs the application control flow. This paper describes experimental results of test application de-obfuscation via dynamic analysis. Both obfuscating and de-obfuscating tools were independently developed by two different teams of ISP RAS – the LLVM-based obfuscating compiler and the software environment for dynamic analysis of binary code. The paper briefly introduces both software tools and then describes results of experimental research on recovering of control flow graph of obfuscated application. Application was initially protected by specialized obfuscating LLVM-based compiler. Next, TrEx environment was used to analyze program execution trace, to find the dispatcher-protected part of application and to recover its control flow. Additionally, some software code complexity metrics for test applications were calculated to estimate obfuscation resilience provided by different versions of obfuscating compiler.

Keywords: Dynamic analysis, virtual machine, dispatcher, code metrics

References

[1]. Tikhonov A.YU., Avetisyan A.I. Razvitie taint-analiza dlya resheniya zadachi poiska programmnykh zakladok [Development of taint-analysis methods to solve the problem of searching of undeclared fatures]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 20, pp. 9–24 (in Russian).

[2]. Padaryan V. A., Get'man A. I., Solov'yov M. A. Programmnaya sreda dlya dinamicheskogo analiza binarnogo koda [Software environment for dynamic analysis of binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], 2009, vol. 16, pp. 51–72 (in Russian).

[3]. Batuzov K., Dovgalyuk P., Koshelev V., Padaryan V. Dva sposoba organizatsii mekhanizma polnosistemnogo determinirovannogo vosproizvedeniya v simulyatore QEMU [Two approaches to full-system deterministic replay in QEMU]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 77–94 (in Russian).

[4]. Korel B., Laski J. Dynamic program slicing. Information Processing Letters, 1988, vol. 29, Issue 3, pp. 155–163. doi: 10.1016/0020-0190(88)90054-3

[5]. Get'man A.I., Markin YU.V., Padaryan V.A., SHHetinin E.I. Vosstanovlenie formata dannykh [Format recovery] Trudy ISP RAN [The Proceedings of ISP RAS], 2010, vol. 19, pp. 195–214 (in Russian).

[6]. Avetisyan A.I., Get'man A.I. Vosstanovlenie struktury binarnykh dannykh po trassam program [Recovery of binary data structures from program traces] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 95–118 (in Russian).
[7]. Sharif M., Lanzi A., Giffin J., Lee W. Automatic reverse engineering of malware emulators Proceedings of the 2009 30th IEEE Symposium on Security and Privacy — SP '09. — Washington, DC, USA : IEEE Computer Society, 2009, pp. 94–109. doi: 10.1109/SP.2009.27
[8]. Ellson J., Gansner E.R., Koutsofios E., et al. Graphviz and dynagraph – static and dynamic graph drawing tools. Graph Drawing Software (ed. by M. Junger, P. Mutzel), Berlin/Heidelberg: Springer-Verlag, 2004. — Mathematics and Visualization. pp. 127–148. doi: 10.1007/978-3-642-18638-7_6
[9]. Ledovskikh I.N., Bakulin M.G.. Podkhod k vosstanovleniyu potoka upravleniya zaputannoj programmy [An approach to reconstruction of control flow of an obfuscated program] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 153–168 (in Russian).
[10]. Kurmangaleev SH.F., Korchagin V.P., Matevosyan R.A. Opisanie podkhoda k razrabotke obfustsiruyushhego kompilyatora [Description of the Approach to Development of the Obfuscating Compiler] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 23, pp. 67–76 (in Russian).
[11]. Kurmangaleev SH.F., Korchagin V.P., Savchenko V.V., Sargsyan S.S. Postroenie obfustsiruyushhego kompilyatora na osnove infrastruktury LLVM [Building Obfuscation Compiler Based on LLVM Infrastructure] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 23, pp. 77–92 (in Russian).
[12]. Tikhonov A. YU., Avetisyan A. I., Padaryan V. A. Metodika izvlecheniya algoritma iz binarnogo koda na osnove dinamicheskogo analiza [Methodology of Exploring of an Algorithm from Binary Code by Dynamic Analysis] Problemy informatsionnoj bezopasnosti. Komp'yuternye sistemy [Problems of Computer Security. Computer Systems], 2008, vol. 3, pp. 66–71 (in Russian).
[13]. Milyutin A., Metriki koda programmnogo obespecheniya [Software code metrics] http://www.viva64.com/ru/a/0045/ [http://www.viva64.com/en/a/0045/], 20.07.2009.
[14]. Hassan R. B. Automatic Measurement of Source Code Complexity. Master's Thesis, Lulea University of Technology, Lulea, Sweden, 2011.

Описание подхода к разработке обфусцирующего компилятора

Курмангалеев Ш. Ф., Корчагин В. П., Матевосян Р.А.,
kursh@ispras.ru,korchagin@ispras.ru, hripsime@ispras.ru

Аннотация. В данной статье приводится обзор запутывающих преобразований программ, сформулированы критерии эффективности методов обфускации. Также предлагается подход к реализации обфусцирующего компилятора на основе инфраструктуры LLVM. Особенность подхода заключается в одновременном применении преобразований, маскирующих различные аспекты работы запутываемого приложения, что обеспечивает стойкую защиту от статического анализа.

Ключевые слова: llvm; обфускация.

1. Введение

В настоящее время актуальна задача защиты программ, как от статического, так и от динамического анализа кода. Доступность качественных средств для анализа кода и большой выбор подключаемых модулей, в автоматическом режиме обходящих многие приемы противодействия анализу, понижают планку требований к квалификации аналитика. Что ведет к повышению требований к защите программ. Необходимо использовать либо методы противодействия анализу неизвестные широкому кругу лиц, либо использовать преобразования достаточно трудоемкие для анализа. Оптимальным выбором, позволяющим реализовать максимально стойкие варианты запутывания программ, является создание обфусцирующего компилятора[1] на базе одной из существующих компиляторных инфраструктур. С одной стороны это позволит производить запутывание программы, имея полную информацию о ней на всех этапах компиляции, а с другой позволит сосредоточиться на разработке защиты, а не на создании требуемой инфраструктуры. Для реализации прототипа была выбрана инфраструктура LLVM [2], позволяющая получать для программы промежуточные и бинарные коды, а также из промежуточного представления генерировать код на языке C.

2. Оценка эффектности методов обфускации

Поскольку известные методы защиты от тестирования отладчиками разного типа имеют ряд существенных недостатков, таких как платформозависимость, наличие средств автоматического обхода известных методов, было принято решение отказаться от реализации таких методов**.** В работах [3] и [4] рассматриваются критерии эффективности методов обфускации основанные на метриках сложности программ. Опишем используемые метрики:

Метрикой называется отображение, ставящее в соответствие каждой программе некоторое число.

Метрика LC размера процедуры [5] — простейшая метрика сложности кода. Чем больше процедура, тем выше априорная оценка её сложности. Размер каждой процедуры будет измерен в инструкциях промежуточного представления.

Метрика YC сложности циклической структуры определяется как мощность транзитивного замыкания отношения достижимости в графе потока управления процедуры.

Максимальная мощность транзитивного замыкания равна $(n-1)^2 = n^2 - 2n + 1$, где n — количество базовых блоков в графе потока управления. Из терминального базового блока не достижим ни один базовый блок, из начального базового блока достижимы все базовые блоки, кроме самого *начального базового блока*, а из любого оставшегося базового блока достижимы все базовые блоки, кроме *начального*.

Если в графе потока управления любая вершина достижима из *начального базового блока*, и *конечный базовый блок* достижим из любой вершины, то минимальная сложность циклической структуры графа из n вершин равна $3n - 6$. Эти два соотношения определяют диапазон, в котором может находиться значение этой метрики. Пусть M — мощность транзитивного замыкания для некоторой процедуры. Тогда для удобства

$$YC = \frac{M - (3n - 6)}{(n-1)^2 - (3n - 6)}$$

Следовательно, $0 < YC < 1$.

Метрика DC сложности потока данных процедуры [6] — позволяет оценить сложность зависимостей по данным в процедуре. Значение метрики DC

вычисляется как количество дуг в графе, достигающих определений процедуры.

Метрика MC усложнения программы при запутывании вычисляется по формуле:

$$MC = \frac{YC(\text{послезащиты})}{YC(\text{дозащиты})} + \frac{DC(\text{после})}{DC(\text{дозащиты})}$$

Чем больше метрика усложнения программы, тем сложнее для понимания становится замаскированная программа в силу увеличения числа информационных и управляющих связей. Так же Чернов приводит сводную таблицу метрики *MC* для различных методов обфускации (табл. 1).

Преобразование	*MC*
Открытая вставка процедур	2.63
Выделение процедур	2.79
Непрозрачные предикаты TTI	14.43
Недостижимый код	3.63
Мёртвый код	3.92
Дублирующий код	2.77
Внесение тождеств	4.85
Внесение несводимости	3.21
Переплетение процедур	1.45
Клонирование базовых блоков	4.89
Развёртка циклов	3.14
«Диспетчер»	6.14
Локализация переменных	2.71
Расширение области действия переменных	2.29
Повышение косвенности	13.00

Табл.1. *Сводная таблица характеристик маскирующих преобразований*

Видно, что наилучшими параметрами обладают преобразования, повышения косвенности, вставки непрозрачных предикатов, и введения диспетчера. Очевидно, что устойчивость к анализу возрастет, при применении нескольких запутывающих преобразований. Но также можно повысить сложность самих преобразований путем связывания потоков данных, вводимых преобразованием, и потоков данных исходной программы, и устранения слабых мест преобразований. Например, для переплетения процедур таким слабым местом является единственная точка диспетчеризации по дополнительному параметру для выбора рабочей процедуры.

3. Сводимость графа потока управления

В современных языках программирования содержится стандартный набор управляющих структур – if-then-else, for, while,do-while,switch и пр. Каждая из этих структур вносит в граф потока управления свой специфический, шаблонный подграф. Примеры таких подграфов для структур do-while и if-then-else показаны на рис. 1:

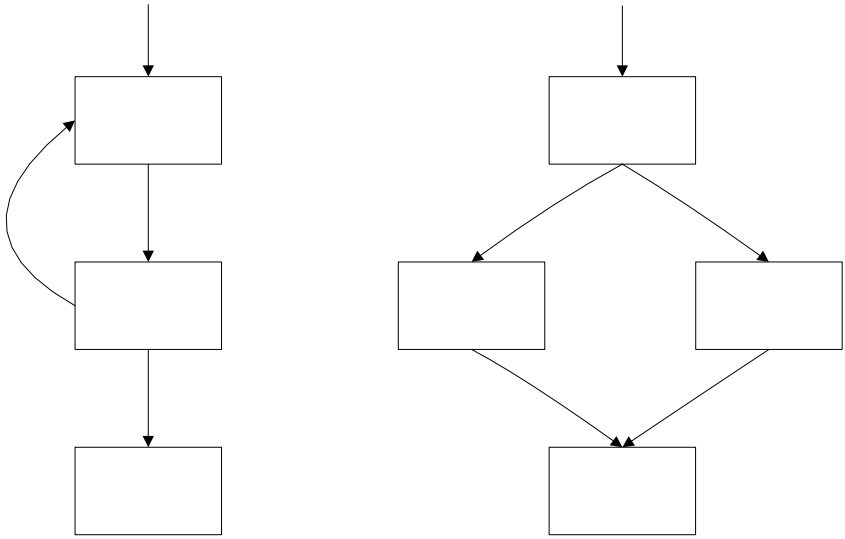


Рис. 1. Слева шаблон do-while, справа шаблон if-then-else.

Рассмотрим произвольный граф потока управления. Имея в наличии шаблоны для управляющих структур, имеется возможность применять их к графу следующим образом: в графе находится тот или иной шаблон, сворачивается в

один узел и вставляется вместо шаблона, к получившемуся графу также применяется свертка. Граф может выродиться в одну вершину, в этом случае будем говорить, что граф сводимый. В противном случае граф несводим. Пример несводимого участка показан на рис. 2 слева. Этот участок можно привести к сводимому, как показано на рис. 2 справа.

Приведение несводимого участка к сводимому увеличивает размер исходного графа потока управления.

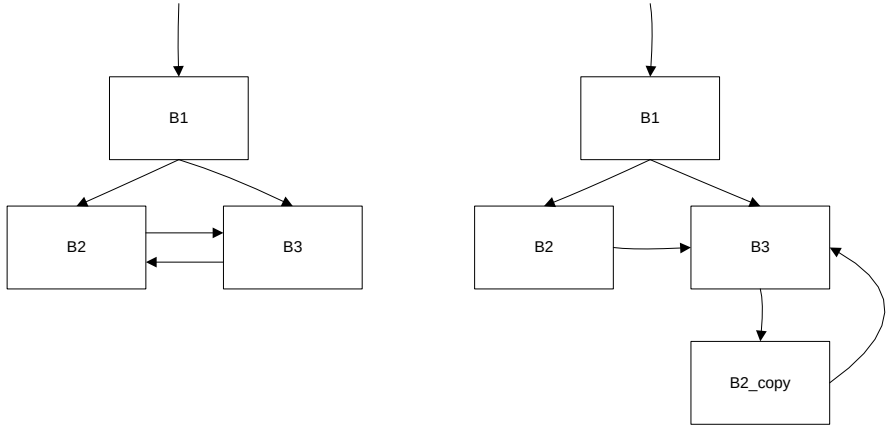


Рис. 2. Слева несводимый граф, справа сводимый вариант.

4. Декомпиляция. Проблемы декомпиляции

Декомпилятор – программа, которая воссоздает код на языке программирования высокого уровня, исходя из исполняемого модуля, который в свою очередь был сгенерирован компилятором. Таким образом, можно сказать, что декомпилятор является противоположностью компилятора. Как во время компиляции, так и во время декомпиляции код проходит несколько этапов анализа. Одним из таких этапов является генерация графа потока управления (CFG) и анализ потока управления на его основе. Для воссоздания управляющих структур высшего уровня декомпилятор ищет в CFG знакомые ему шаблоны (напр. рис. 1). При нахождении несводимых участков в графе, декомпилятор будет пытаться привести их к сводимым, как это сделалось с графом из рис. 2. В работе [7] указаны два сложно поддающихся анализу случая с циклами (рис. 3).

Даже если декомпилятору удастся сделать такое преобразование, воссоздание начальной версии кода станет невозможным.

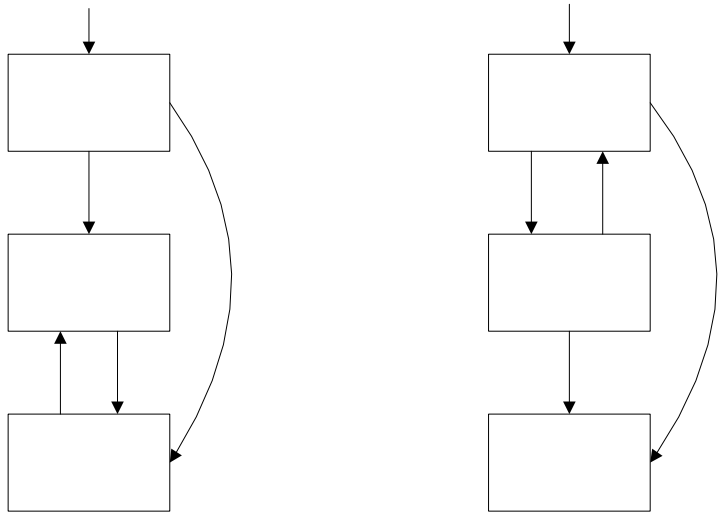


Рис. 3. Слева случай с несводимым графом, справа случай с ребром, выходящим из цикла.

Первый случай на рис. 3 идентичен треугольнику на рис. 2 слева. А на рис. 3 справа дело обстоит наоборот – ребро не входит в цикл, а выходит из него.

Таким образом, можно заключить, что генерация трудных для декомпилятора случаев, повысит стойкость защиты.

5. Определение критериев эффективности методов обфускации

В работе Щелкунова [5] предлагается ряд неформальных правил, которым должны удовлетворять запутывающие преобразования:

- Маскировать граф потока управления подпрограммы;
- Граф потока управления подпрограммы должен быть неприводимым;
- Определения фальшивых переменных не должны уничтожаться в конце подпрограммы;
- Определения фальшивых переменных не должны уничтожаться до того, пока эти переменные не были хоть раз использованы;
- «Мертвый» код не должен сильно отличаться от реально выполняемого кода. Все правила, приведенные выше, должны относиться, в том числе, и к нему.

Использование вышеприведенных правил позволит существенно усложнить процесс автоматической деобфускации подпрограммы. Более того, благодаря

использованию глобального контекста, достаточно сложно будет понять, что именно делает подпрограмма без детального анализа подпрограмм, которые с ней взаимодействуют.

Проведем сравнительный анализ преимуществ и недостатков методов обфускации по группам, поскольку все преобразования одной группы имеют сходные достоинства и недостатки:

- Маскировка текста. Преобразования такого типа не оказывают влияния на стойкость к анализу бинарного кода программы, поэтому в данной работе они не рассматриваются;
- Маскировка управляющей структуры. Поскольку преобразования затрагивают только граф потока управления, то при анализе потока данных зачастую появляется возможность обратить такое преобразование;
- Маскировка данных. Позволяет существенно усложнить анализ программы, поскольку усложняют анализ, как с помощью автоматических средств анализа, так и непосредственно аналитик не может сделать заключение о функционале, не обладая информацией о потоке данных касающегося анализируемого фрагмента. Но зачастую к себе привлекают внимание, повторяющиеся действия по шифрованию/дешифрованию или нехарактерная работа с данными. После восстановления схемы работы, анализ не представляет большой проблемы;
- Преобразования, затрагивающие и граф потока управления, и поток данных, компенсируют недостатки обоих подходов и позволяют добиться максимальной устойчивости. Но более трудоемки в разработке и реализации.

Можно сформулировать следующие критерии эффективности:

- Маскирующее преобразование должно затрагивать и поток управления, и поток данных запутываемой программы;
- Стойкость преобразования должна основываться на алгоритмически сложных задачах, например, требовать от атакующего применения анализа указателей, для точного восстановления потоков данных защищенной программы;
- При разработке преобразования нужно учитывать особенности работы средств анализа, например для автоматических декомпиляторов следует насытить граф потока управления несводимыми участками.

6. Предлагаемые методы запутывания

На базе проведенных исследований разработаны методы усложнения программного кода с оценкой роста используемой памяти и оценкой падения быстродействия обфусцированного программного кода.

Методы усложнения программного кода:

- Преобразование, перемещающее локальные переменные в глобальную область видимости;
- Защита константных строк, используемых программой;

- Вставка в код фиктивных циклов;
- Приведение графа потока управления к плоскому виду с применением алгоритма диспетчеризации;
- Переплетение нескольких функций в одну с заменой всех вызовов отдельных функций на вызов одной общей;
- Сокрытие вызовов функций. Для защищаемой функции создается функция-переходник, внутри которой содержится несколько вызовов различных функций. Вызов нужной функции определяется с помощью трудного предиката;
- Запутывание графа потока управления – создание несводимых участков в графе;
- Замена вызовов одной и той же функции на вызовы ее копий.

Для реализации алгоритмов запутывания был выбран подход создания обфусцирующего компилятора на базе имеющихся решений с открытым кодом. Такой выбор обусловлен следующими причинами:

- Многие алгоритмы обфускации требуют наличия информации характерной для компиляторов;
- Встраивание защиты во время компиляции позволяет увеличить ее стойкость и скорость разработки защиты;
- Во время компиляции в расположении имеется максимальная информация о программе, в отличие от случая защиты исполняемого файла.

В настоящий момент параллельно проводятся работы по тестированию стойкости предложенных преобразований [8].

7. Требования к инфраструктуре

Сформулируем основные требования к компиляторной инфраструктуре. Компиляторная инфраструктура, на базе которой будет разрабатываться запутывающий компилятор должна удовлетворять следующему набору требований:

- Обеспечивать компиляцию исходных кодов на C/C++ под Windows и Linux;
- Иметь открытые исходные коды;
- Иметь документацию и поддержку сообщества;
- Расширяемость;
- Возможность влиять на генерируемый код на любом этапе компиляции, от препроцессора до кодогенерации;
- Возможность получить различную информацию об обрабатываемой программе на любой стадии компиляции, требуемую для реализации алгоритмов запутывания кода;
- Возможность, получить код на языке "C"из внутреннего представления компилятора.

- LLVM - компиляторная инфраструктура с открытыми исходными кодами. Перечислим ее основные особенности:
- Реализована на C++;
- Модульная и расширяемая архитектура;
- Имеет как статический, так и динамический компилятор.
- Поддерживает несколько фронтэндов:
- C, C++, Objective-C (Clang, GCC/dragonegg);
- Ruby (Rubinius, MacRuby);
- Python (unladen swallow);
- Поддерживает множество целевых архитектур: ARM, Alpha, Intel x86, Microblaze, MIPS, PowerPC, SPARC и т. д.

Промежуточное представление (LLVM IR) играет центральную роль в процессе компиляции. Все оптимизации реализованы как компиляторные проходы преобразования“LLVM IR to LLVM IR”.

Анализ кода, может быть реализован как отдельный проход, а его результаты могут разделять несколько проходов трансформирующих код.

Все машинно-зависимые оптимизации происходят в отдельном генераторе кода для каждой машины.

8. Заключение

В данной статье был проведен обзор запутывающих преобразований программ, были приведены критерии эффективности методов обфускации. Был предложен подход к реализации обфусцирующего компилятора.

Описанный в работе подход к построению обфусцирующего компилятора применяется при разработке обфусцирующего компилятора в ИСП РАН. Реализация представляет собой набор компиляторных проходов, запускаемых после работы стандартных оптимизирующих проходов компилятора LLVM. Особенности реализации разработанных алгоритмов будут раскрыты в последующих публикациях.

Список литературы

[1] Christian Collberg; JasvirNagra Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection Addison-Wesley Professional Pub. Date: July 24, 2009 Print ISBN-10: 0-321-54925-2, 748 стр.

[2] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. — Master’s thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.

[3] А.В. Чернов. Анализ запутывающих преобразований программ. Труды ИСП РАН, том 3, 2002, стр. 7-38

[4] Д. А. Щелкунов. Применение запутывающих преобразований и полиморфных технологий для автоматической защиты исполняемых файлов от исследования и модификации. Труды международной конференции РусКрипто. Апрель 2008 г.

[5] M. H. Halstead. Elements of Software Science. Elsevier North-Holland, 1977.

[6] E. I. Oviedo. Control flow, data flow, and program complexity. In Proceedings of IEEE COMPSAC, 1980, pp. 146-152.
[7] Simon Moll. Decompilation of LLVM IR. B.Sc. Thesis, Saarland University, 2011.
[8] М.Г. Бакулин, С.С. Гайсарян, Ш.Ф. Курмангалеев, И.Н. Ледовских, В.А. Падарян, С.М. Щевьева Динамический анализ обфусцированных приложений с диспетчеризацией или виртуализацией кода. Сдано в печать: Труды ИСП РАН, том 23, 2012, 17 стр.

Building an obfuscation compiler based on LLVM infrastructure

Kurmangaleev S.F. kursh@ispras.ru
ISP RAS, Moscow, Russia
Korchagin V. P. korchagin@ispras.ru
ISP RAS, Moscow, Russia
Savchenko V.V. sinmipt@ispras.ru
ISP RAS, Moscow, Russia
Sargsyan S.S. sevaksargsyan@ispras.ru
ISP RAS, Moscow, Russia

Annotation. The paper describes the LLVM-based obfuscating compiler developed in ISP RAS. The main goal was to implement the obfuscating transformations as standalone compiler passes, so they could be used either separately or together with optimization passes. The proposed transformations are based on well-known obfuscation algorithms (including constant string protection, fake cycle insertion, control flow graph flattening, functions merge, function call encapsulation, control flow graph structure obfuscation, opaque predicate insertion and other) and are specifically improved to resist better to static analysis deobfuscation techniques. For that purpose several effectiveness measures were performed. For example, flattening and function merge algorithms were significantly improved in the direction of static analysis protection. In addition to complete description of implemented methods, the application performance decrease estimation and the increase of application memory consumption estimation are presented. Also, the possibility of source code information recovery is estimated. The implemented obfuscating transformations can be applied together to the given application to provide the holistic protection from the static analysis deobfuscation attacks. Results achieved with these transformations match the quality of the best obfuscating tools that are present on the market.

Keywords: LLVM, obfuscation

References

[1]. [1] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization.— Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, 61 pages

[2]. [2] D. A. Shhelkunov. Primenenie zaputyvayushhikh preobrazovanij i polimorfnykh tekhnologij dlya avtomaticheskoj zashhity ispolnyaemykh fajlov ot issledovaniya i modifikatsii. [Applying obfuscation transformations and polymorphic technologies for automatic protection executable files from analysis and modification]. Trudy mezhdunarodnoj konferentsii RusKripto. [Proceedings of international conference RusCrypto]. April 2008 (in Russian).

[3]. [3] A.V. Chernov. Analiz zaputyvayushhikh preobrazovanij programm. [Analysis obfuscating program transformations] Trudy ISP RAN [The Proceedings of ISP RAS], 2002, vol.3, pp. 7-38 (in Russian).

[4]. [4] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report. University of Virginia, Charlottesville, VA, USA., 18 pages
[5]. [5] Ilya N. Ledovskikh, Maxim G. Bakulin. Podkhod k vosstanovleniyu potoka upravleniya zaputannoj programmy. [An Approach to Reconstruction Control Flow of the Obfuscated Program] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 153-168 (in Russian).
[6]. [6] N.P. Varnovskij, A.V. Shokurov. Gomomorfnoe shifrovanie. [Homomorphic encryption]. Trudy ISP RAN [The Proceedings of ISP RAS], 2007. Vol. 12, pp. 27-36. (in Russian).
[7]. [7] Christian Collberg. Jasvir Nagra Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional. Pub. Date: July 24, 2009. Print ISBN-10: 0-321-54925-2.
[8]. [8] N.P. Varnovskij, V.A. Zakharov, N.N. Kuzyurin, A.V. Chernov, A.V. Shokurov. Ob osobennostyakh primeneniya metodov obfuskatsii programm dlya informatsionnoj zashhity mikroehlektronnykh skhem. [About usage features of the program obfuscation techniques applying to informational microelectronic circuits security]. Trudy ISP RAN [The Proceedings of ISP RAS], 2006, vol. 11, pp. 27-60 (in Russian).
[9]. [9] A.V. Chernov Ob odnom metode maskirovki programm [About one method program masking], Trudy ISP RAN [The Proceedings of ISP RAS], 2003, vol.4, pp. 85-119 (in Russian).
[10]. [10] M.G. Bakulin, S.S. Gaissaryan, Sh.F. Kurmangaleev, I.N. Ledovskikh, V.A. Padaryan, S.M. Shchevyeva. Dinamicheskij analiz obfustsirovannykh prilozhenij s dispetcherizatsiej ili virtualizatsiej koda. [Dynamic analysis of virtualization- or dispatching-obfuscated applications]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012,vol. 23, pp. 49-66. (in Russian).

Построение обфусцирующего компилятора на основе инфраструктуры LLVM

Курмангалеев Ш. Ф., Корчагин В. П., Савченко В. В., Саргсян С.С.
korchagin@ispras.ru, kursh@ispras.ru sinmipt@ispras.ru
sevaksargsyan@ispras.ru

Аннотация. В статье описываются маскирующие преобразования, реализованные в ходе разработки обфусцирующего компилятора в ИСП РАН, приводится оценка понижения быстродействия и увеличения объема потребляемой приложением памяти, а также оценка возможности восстановления информации об исходном коде. Реализованные маскирующие преобразования могут быть одновременно применены к запутываемому приложению, что увеличивает степень защиты приложения и обеспечивают стойкую защиту от статического анализа.

Ключевые слова: llvm; обфускация.

1. Введение

Оптимальным выбором, позволяющим реализовать стойкие варианты запутывания программ, является создание обфусцирующего компилятора на базе одной из существующих компиляторных инфраструктур. С одной стороны, это позволит производить запутывание программы, имея полную информацию о ней на всех этапах компиляции, с другой – позволит сосредоточиться на разработке алгоритмов защиты, а не на создании требуемой инфраструктуры. Для реализации была выбрана компиляторная инфраструктура LLVM[1], в качестве компилятора переднего плана используется Clang (читается как клэнг).

Целью работы является построение обфусцирующего компилятора для защиты программы от обратной инженерии с помощью инструментов, использующих статический анализ кода. Все разработанные преобразования представляют собой отдельные компиляторные проходы, запускаемые поочередно, после окончания работы оптимизирующих проходов. Преобразования производятся во время обработки промежуточного представления LLVM на машинно-независимом уровне, что, с одной стороны, позволяет получать запутанное промежуточное представление, которое в дальнейшем можно преобразовать в код на языке Си с помощью стандартных

инструментов LLVM. С другой стороны, такой подход обеспечивает поддержку нескольких архитектур при условии совпадения порядка байтов и минимального различия в ABI.

При разработке преобразований учитывались критерии эффективности:

- Маскирующее преобразование должно затрагивать и поток управления, и поток данных запутываемой программы;
- Стойкость преобразования должна основываться на алгоритмически сложных задачах, например, требовать от атакующего применения анализа указателей для точного восстановления потоков данных защищенной программы [2];
- При разработке преобразования нужно учитывать особенности работы средств анализа [3], например, для автоматических декомпиляторов следует насытить граф потока управления несводимыми участками.

Разработанные методы усложнения программного кода:

- Преобразование, перемещающее локальные переменные в глобальную область видимости;
- Защита константных строк, используемых программой;
- Вставка в код фиктивных циклов;
- Приведение графа потока управления к плоскому виду с применением алгоритма диспетчеризации;
- Переплетение нескольких функций в одну с заменой всех вызовов отдельных функций на вызов одной общей;
- Сокрытие вызовов функций. Для защищаемой функции создается функция-переходник, внутри которой содержится несколько вызовов различных функций. Вызов нужной функции определяется с помощью трудного предиката;
- Запутывание графа потока управления – создание несводимых участков в графе;
- Замена вызовов одной и той же функции на вызовы ее копий.

2. Описание маскирующих преобразований

2.1. Преобразование, маскирующее вызов функций

Для защищаемого вызова создается функция-переходник, внутри которой содержится несколько вызовов функций. Внутри переходника вызов нужной функции диспетчеризуется по значению трудного предиката. Реализовано два варианта преобразования: только для вызовов внешних функций и для вызовов всех функций.

Для каждого вызова функции производится его замена на вызов функции-переходника. Для избегания чрезмерной вложенности вызовов, переходники на переходники не создаются. Для выбора функций, которые будут размещены внутри переходника, была введена мера "близости функций".

Значение меры - коэффициент Жаккарда для множеств типов аргументов двух функций. Половина функций в переходнике выбираются, как самые "похожие" по введенной мере, и другая половина - "непохожие".

Аргументы, передаваемые в функцию-переходник запутываются с помощью битовой операции "исключающее ИЛИ". Между всеми аргументами делается операция "исключающего ИЛИ", обозначим результат операции за S. Затем к каждому аргументу применяется операция "исключающего ИЛИ" с S, и в таком виде аргумент передается в функцию. Также для распутывания передается само значение S. Внутри функции-переходника сначала происходит распутывание аргументов. Производится операция "исключающего ИЛИ" между каждым аргументом и значением S. В результате аргумент получает свое исходное значение.

После распутывания аргументов происходит вычисление непрозрачного предиката, по результату которого происходит диспетчеризация вызова функций. Диспетчеризация вызовов функций производится с помощью большого switch блока. Каждое значение в нем сгенерировано случайным образом и соответствует какой-либо функции. Все вызовы перемешиваются в случайном порядке внутри функции-переходника. Последний аргумент функции-переходника служит для определения функции, которая будет вызвана. Этот аргумент передается в непрозрачный предикат. Так, как значение предиката известно, на этапе компиляции, то можно подобрать такое значение аргумента, которое соответствовало бы нужной функции.

2.2. Маскирующее преобразование «диспетчер»

Идея маскирующего преобразования «диспетчер» заключается в преобразовании графа потока управления таким образом, что статический анализ переходов между базовыми блоками становится трудной задачей [4].

Данное преобразование состоит в том, что базовым блокам присваиваются номера. В начало функции вставляют блок «диспетчер» - аналог switch в языке С. В конец каждого блока дописывается, код устанавливающий номер следующего блока для выполнения и передающий управление на диспетчер. Диспетчер же на основе переданной ему информации принимает решение куда дальше передать управление.

Терминальные инструкции маскируемых базовых блоков удаляются, а содержащие их базовые блоки модифицируются таким образом, чтобы передать управление диспетчеру с номером базового блока, соответствующего нужной ветви перехода. Для каждого блока делается до 5 копий, которые так же добавляются в диспетчер. Помимо этого, производится усреднение размера базовых блоков, инструкции “call” оцениваются как несколько инструкций, так как передача параметров в промежуточном представлении LLVM производится в той же команде, а на реальных архитектурах по команде на аргумент. Для сокрытия переменной диспетчеризации предпринято следующее:

Значение переменной диспетчеризации вычисляется по формуле I = X1 XOR Z; а следующее значение Z по формуле $Z_{след}$ = X2 XOR $Z_{текущее}$; Z, X1 и X2 выбираются случайным образом для блока предшествующего диспетчеру, и X2 генерируется случайным образом для каждого блока исходной функции во время его обработки.

В каждом блоке выбирается одна переменная подходящего типа, с которой посредством операции "исключающего ИЛИ" (XOR) происходит сцепление переменной диспетчеризации. Такое преобразование не позволит выделить переменную диспетчеризации с помощью алгоритма обратного слайсинга [5], так как в полученную таким образом трассу окажутся, вовлечены живые переменные, вычисляемые в программе.

Вид графа потока управления после преобразования примерно следующий:

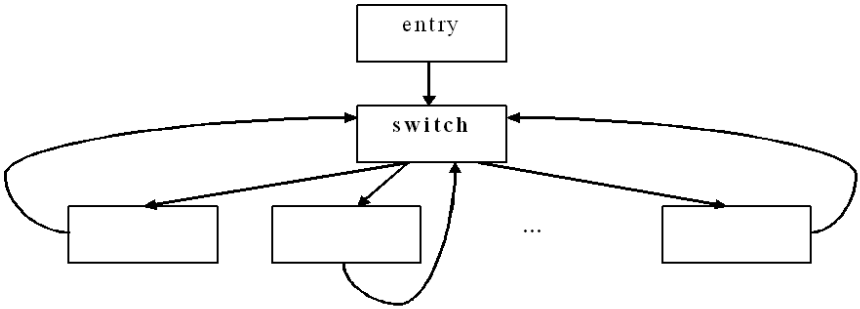


Рис. 1. Вид графа потока управления после запутывания.

2.3. Преобразование, маскирующее строковые константы

Зачастую во время статического анализа, строковые константы, хранящиеся в открытом виде, могут дать аналитику дополнительную информацию о функционировании программы [6]. Или помочь найти интересующий код, по строкам, выводимым во время интересующего его события. Преобразование, маскирующее строковые константы, предназначено для сокрытия информации о строках, во время статического анализа программы.

Защита выполняется следующим образом: вначале все константные строки, кроме тех, что содержатся в агрегатных типах (массивы, контейнеры из стандартной библиотеки), шифруются, в модуль добавляются шифрующая и дешифрующая функции. Перед каждым использованием той или иной строки вставляется вызов функции дешифратора, а после – шифрующей функции. Это справедливо для строк, для которых не выполняются операции с указателями. Если же такие операции имеют место, то для корректной работы запутывающего алгоритма необходим анализ указателей. В таких случаях

обратного шифрования строки не производится. Шифрование строк после использования требуется для того, что бы во время работы программы все строки не находились в памяти расшифрованными, поскольку в этом случае можно сделать снимок памяти процесса, после дешифрования строк. Шифрование строк производится с помощью операции исключающего "или" со случайным ключом.

2.4. Маскирующее преобразование, приводящее граф потока управления к несводимому

Колберг [7] описывает алгоритм, который приводит граф потока управления к несводимому. Алгоритм заключается в том, что для каждого цикла добавляется «фиктивное» ребро из заголовка цикла в его тело. Добавление такого ребра осуществляется с помощью вставки непрозрачных предикатов.

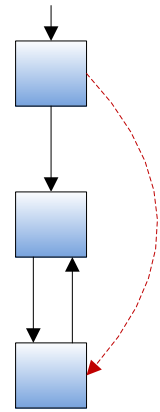


Рис.2. Добавление «фиктивного» ребра в цикл.

Таким образом, трансформируя граф потока управления, мы создаем сложный для анализа случай, препятствующий качественной декомпиляции.

2.5. Переплетение циклов

Алгоритм запутывания для всех циклов функции добавляет «фиктивные» ребра из одного цикла в другой. Как видно из рис. 2, метод создает ребра как входящие в цикл, так и выходящие из него.

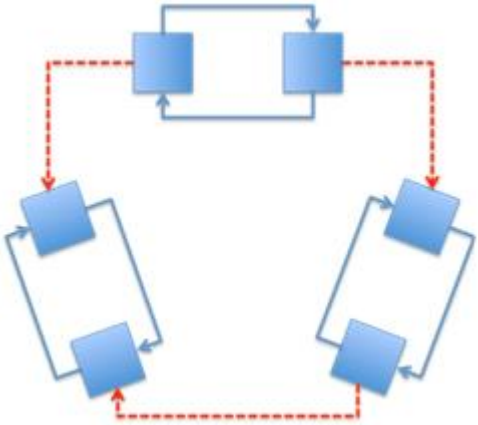


Рис.3. Переплетение 3 циклов, пунктиром показаны «фиктивные» ребра.

Минус такой трансформации состоит в том, что она эффективно запутывает только код больших функций, которые имеют сложные структуры, содержащие несколько циклов. Помимо этого применяется следующий подход для запутывания всего графа потока управления: из множества блоков функции выбирается N блоков и между ними случайно добавляются ребра. Фиктивные переходы защищаются непрозрачными предикатами.

2.6. Маскирующее преобразование, осуществляющее переплетение функций

Классический подход к переплетению функций, обладает малой стойкостью. Он предполагает объединение сигнатур функций и наличие параметра, по которому происходит диспетчеризация[8]. Восстановить исходный код переплетенных таким образом функций не составляет особого труда. Предложена модификация упомянутого алгоритма таким образом, чтобы помимо диспетчеризующего условия переплетаемые функции имели точки пересечения потоков управления и потоков данных, чтобы применение алгоритма обратного слайсинга не позволяло найти единственную точку, в которой производится выбор рабочей функции.

Переплетение происходит следующим образом:

1. Объединяются сигнатуры двух функций, генерируется дополнительный параметр, по которому в процессе выполнения будет производиться выбор функции.

2. Если функции возвращают целочисленное значение, то для реального возврата значения из переплетенной функции используются 2 глобальные перемененные, а сама функция возвращает неиспользуемое значение. Если оригинальные функции возвращают указатели, то тип возвращаемого

значения переплетенной функции становится указателем на void. Преобразование проиллюстрировано на рисунках

3. В новой функции, полученной на основе переплетения двух функций, произвольно выбираются два базовых блока (один из первой функции второй из второй). Для которых производится преобразование зацепления дуг [9] рис. 4. В генерируемом общем базовом блоке производятся вычисления с глобальными переменными. Для затруднения анализа потоков данных эти переменные используются для вычислений в и других функциях модуля. Таким образом, у двух переплетенных функций всегда будут общие вычисления. Результат вычислений, используются в качестве возвращаемого значения и переплетенной функции и в глобальную переменную, что не позволит исключить добавленные вычисления как мертвый код, результат которых нигде не используется.

4. После генерации переплетенной функции, все места вызова оригинальных функций, заменяются на вызов переплетенной функции с генерацией дополнительных параметров и изменением обработки возвращаемого значения.

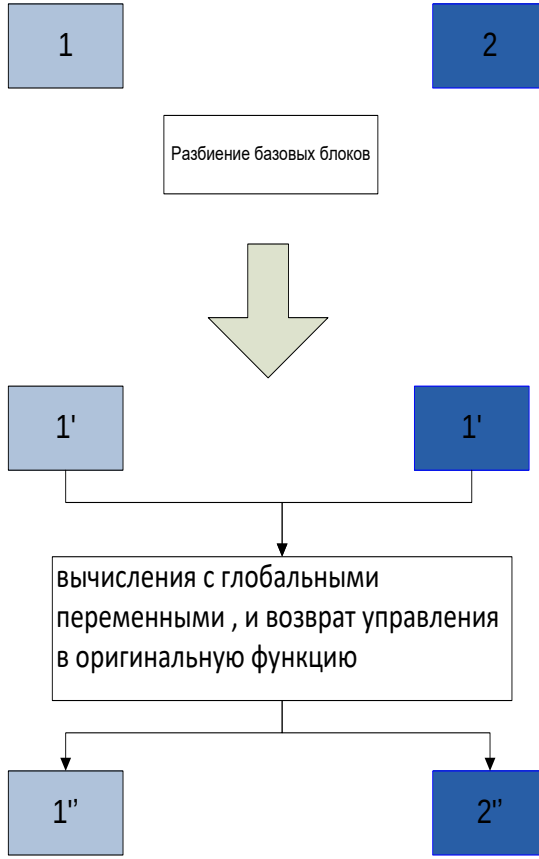


Рис. 4. Преобразование зацепления дуг.

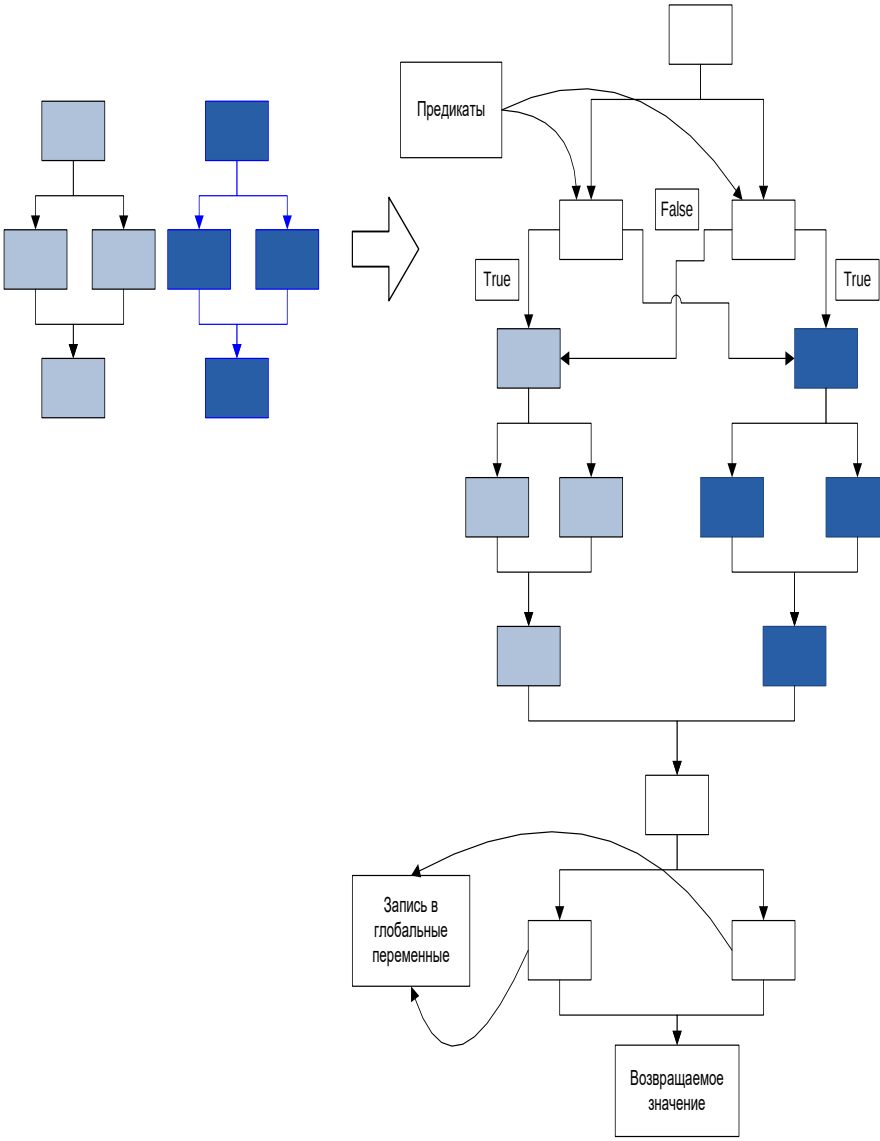


Рис. 5. Переплетение функций.

2.7. Маскирующее преобразование, направленное на увеличение сложности анализа потока данных в программе

Идея преобразования состоит в следующем:

1. Перенос локальных переменных в глобальную область видимости;
2. Использование и изменение перенесенных переменных в разных функциях.

В общем случае нельзя изменять значения переменных, вынесенных из других функций в произвольном месте программы, так как это может привести к неправильному выполнению компилируемой программы. Поэтому строится граф вызовов для всех функций в модуле (рис. 5), анализируя который для каждой функции вычисляется множество переменных модификация которых, не нарушит работоспособность программы. Такими переменными будут переменные, вынесенные из функций, расположенных на разных путях в дереве вызовов. После формирования множеств подходящих переменных, осуществляется добавление мусорного кода, использующего для вычислений «безопасные» переменные. Также найденные переменные используются в предикатах. Функции, передаваемые по адресу в другие функции, не обрабатываются, так как они могут использоваться в многопоточном коде и обращение к одной глобальной переменной может вызвать сбой в работе программы.

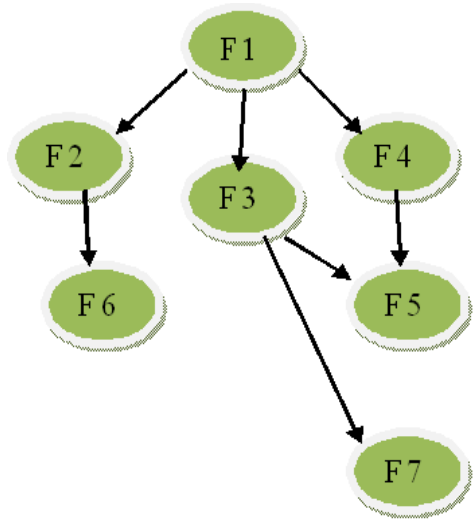


Рис.6. Пример дерева вызовов.

2.8. Маскирующее преобразование, осуществляющее разбиение целочисленных констант

Часто в коде в явном виде встречаются константы характерные для определенных алгоритмов, например константа 0x67452301 для MD5. Поиск констант позволяет определить используемый алгоритм, что упрощает анализ программы. Для противодействия предложен алгоритм разбиения констант. Разбиваются только константы больше единицы. Для разбиения случайным образом выбирается число меньшее исходного, которое будет выступать в качестве первого слагаемого, второе слагаемое получается автоматически.

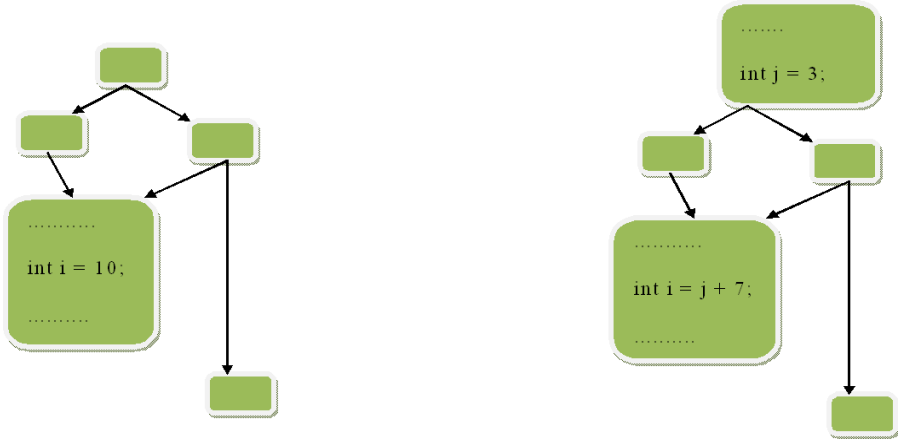


Рис.7. Пример работы.

2.9. Маскирующее преобразование, осуществляющее размножение тел функций

Для каждого использования функции внутри программного модуля, производится создание своего экземпляра вызываемой функции.

Например:

```
#include<stdio.h>
int sum (int a, int b) {
return a+b;
}
int main () {
printf("Sum 2 + 2 = %d \n", sum (2, 2) );
printf("Sum 2 + 3 = %d \n", sum (2, 3) );
return 0;
}
```

Для каждого вызова функции **sum** будет сгенерирована копия тела функции, затем каждый вызов оригинальной функции будет исправлен на вызов соответствующей копии. Такое преобразование увеличивает размер кода программы и время, требуемое для его автоматического анализа. Также, будучи применено совместно с другими запутывающими преобразованиями, зависящими от генератора случайных чисел, они утратят полную идентичность, что повысит сложность анализа, так как потребуется проанализировать каждую копию.

2.10. Маскирующее преобразование, осуществляющее вставку ложных циклов

В коде запутываемой программы происходит поиск участков кода, по структуре напоминающих одну итерацию цикла. Далее в начало участка или в его конец (в зависимости от разновидности вставляемого цикла) вставляется базовый блок с условным переходом в противоположный конец выделенного участка кода. Условный переход содержит в себе непрозрачный предикат, который и маскирует лишь одно исполнение цикла. В качестве подходящего участка кода рассматривается участок с одним входом и выходом.

2.11. Формирование непрозрачных предикатов

Разработано API для использования непрозрачных предикатов в запутывающих преобразованиях. Предикатом является базовый блок или несколько базовых блоков, имеющих один общий терминальный базовый блок. Терминальный базовый блок предиката заканчивается инструкцией условного перехода, которая всегда передает управление только по одной ветке. Причем, основываясь на информации, доступной на этапе компиляции, известно, по какому пути произойдет переход.

Реализовано три типа непрозрачных предикатов:

1. Истинность диофантова уравнения x ^ 2 - n * y ^ 2 = 1. Если параметр n не является точным квадратом, то это уравнение Пелля. При вставке этого предиката, случайным образом выбирается, будет ли он всегда иметь истинное значение, либо ложное. Так, как на этапе компиляции известно, некоторое количество решений данного уравнения, а также для константных значений переменных можно легко проверить истинность уравнения, то можно подобрать значения переменных для выбранного значения предиката.

2. Предикат, основанный на модульной арифметике: (x ^ 3 - x) mod 3 = 0. Это выражение всегда истинно. Значение переменной x для вычисления значения предиката выбирается случайным образом среди целочисленных глобальных переменных. Если таких глобальных переменных нет, то для вычисления предиката используется случайная целочисленная константа.

3. Целочисленное уравнение: 7 * y ^ 2 - 1 = x ^ 2. Это выражение всегда ложно. По аналогии с предыдущим предикатом значения переменных x и y

случайным образом выбираются среди глобальных целочисленных переменных.

При необходимости использовать непрозрачный предикат при запутывании программы, из предложенных предикатов выбирается один случайный.

3. Оценка понижения быстродействия и потребляемой памяти

Произведена оценка увеличения объема и уменьшения быстродействия запутанной подпрограммы. Количество примененных преобразований зависит от входной программы, а замедление программы сильно зависит от быстродействия машины, на которой она будет исполняться, то представляется невозможным для произвольной программы, дать точную оценку замедления. Поэтому в целях практического использования используются коэффициенты, полученные опытным путем, с использованием достаточно большой кодовой базы.

В практических целях было произведен замер замедления на тестах из пакета OpenSSL 1.0.1. и посчитаны метрики на разных уровнях оптимизаций.

Замедление для одного запутывающего преобразования составляет 1,2-5,5 раз, потребление памяти увеличивается в 1,05-2,5 раз.

4. Оценка возможности восстановления информации об исходном коде

Компиляция программы - это процесс с потерями, поэтому точно восстановить исходный код программы на языке высокого уровня в общем случае невозможно. Обфускация программы накладывает дополнительные ограничения на процесс восстановления, так как перед восстановлением самого исходного кода, аналитику следует восстановить граф потока управления программы, расшифровать зашифрованные константы и данные, избавиться от мертвого кода, и провести другие преобразования, обратные запутывающим.

Стоит отметить, что отладка обфусцированного кода представляет собой сложную задачу даже для автора запутывающего преобразования, при наличии исходного кода защищаемого приложения и полной информации о трансформации, произошедшей с программой.

Совместное применение нескольких опций позволит увеличить сложность пропорционально произведению увеличения сложностей каждого преобразования в отдельности. Для примерной оценки сложности анализа, был проведен эксперимент. К программе Sqlite были применены следующие преобразования переплетения функций, перевода переменных в глобальную область видимости, преобразования диспетчеризации и сокрытия вызовов функций. Размер кода приложения увеличился с 2.9 МБ до 15 Мб.

Потребление памяти дизассемблером Ida Pro возросло в ~10 раз. Во время обработки защищенного файла возникло исключение в одной из библиотек Ida Pro, время анализа по сравнению с оригинальным кодом возросло примерно в 10 раз, затем произошло исключение в библиотеках дизассемблера. Также было произведено исследование с помощью инструмента комбинированного анализа, полученные результаты [10] свидетельствуют о том, что обеспечиваемый уровень защиты сравним с уровнем, обеспечиваемым коммерческими разработками.

5. Заключение

В данной статье было приведено описание маскирующих преобразований, реализованных во время разработки обфусцирующего компилятора в ИСП РАН, приведена оценка понижения быстродействия и увеличения объема потребляемой приложением памяти, а также оценка возможности восстановления информации об исходном коде.

Учитывая увеличение сложности при применении маскирующих преобразований, можно сделать вывод, что задача декомпиляции и полного статического анализа потребует намного больше ресурсов, чем декомпиляция и анализ незапутанной программы. Это позволяет говорить о том, что разработанные преобразования обеспечивают хороший уровень защиты.

Список литературы

[1]. Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization.— Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, 61 pages

[2]. Д. А. Щелкунов. Применение запутывающих преобразований и полиморфных технологий для автоматической защиты исполняемых файлов от исследования и модификации. Труды международной конференции РусКрипто. Апрель 2008 г.

[3]. А.В. Чернов. Анализ запутывающих преобразований программ. Труды ИСП РАН, том 3, 2002, стр. 7-38.

[4]. Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report. University of Virginia, Charlottesville, VA, USA., 18 pages

[5]. И.Н. Ледовских, М.Г. Бакулин. Подход к восстановлению потока управления запутанной программы. Труды ИСП РАН, том 22, 2012, стр. 153-168.

[6]. Н.П. Варновский, А.В. Шокуров. Гомоморфное шифрование. Труды ИСП РАН, том 12, 2006, стр. 27-36.

[7]. Christian Collberg. Jasvir Nagra Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional. Pub. Date: July 24, 2009. Print ISBN-10: 0-321-54925-2.

[8]. Н.П. Варновский, В.А. Захаров, Н.Н. Кузюрин, А.В. Чернов, А.В. Шокуров. Об особенностях применения методов обфускации программ для информационной защиты микроэлектронных схем. Труды ИСП РАН, том 11, 2006, стр. 27-60.

[9]. А. В. Чернов. Об одном методе маскировки программ. Труды ИСП РАН, том 4, 2003, стр. 85-119.
[10]. М.Г. Бакулин, С.С. Гайсарян, Ш.Ф. Курмангалеев, И.Н. Ледовских, В.А. Падарян, С.М. Щевьева Динамический анализ обфусцированных приложений с диспетчеризацией или виртуализацией кода. Сдано в печать: Труды ИСП РАН, том 23, 2012, 17 стр.

Building an obfuscation compiler based on LLVM infrastructure

Kurmangaleev S.F. kursh@ispras.ru
ISP RAS, Moscow, Russia
Korchagin V. P. korchagin@ispras.ru
ISP RAS, Moscow, Russia
Savchenko V.V. sinmipt@ispras.ru
ISP RAS, Moscow, Russia
Sargsyan S.S. sevaksargsyan@ispras.ru
ISP RAS, Moscow, Russia

Annotation. The paper describes the obfuscating transformations, which were implemented while developing an LLVM-based obfuscating compiler in ISP RAS. The proposed transformations are based on well-known obfuscation algorithms and are specifically improved to resist better to static analysis deobfuscation techniques. The application performance decrease estimation and the increase of application memory consumption estimation are presented. Also, the possibility of source code information recovery is estimated. The implemented obfuscating transformations can be applied together to the given application to provide the strong protection from the static analysis deobfuscation attacks.

Keywords: LLVM, obfuscation

References

[1]. Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization.— Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, 61 pages

[2]. D. A. Shhelkunov. Primenenie zaputyvayushhikh preobrazovanij i polimorfnykh tekhnologij dlya avtomaticheskoj zashhity ispolnyaemykh fajlov ot issledovaniya i modifikatsii. [Applying obfuscation transformations and polymorphic technologies for automatic protection executable files from analysis and modification]. Trudy mezhdunarodnoj konferentsii RusKripto. [Proceedings of international conference RusCrypto]. April 2008 (in Russian).

[3]. A.V. Chernov. Analiz zaputyvayushhikh preobrazovanij programm. [Analysis obfuscating program transformations] Trudy ISP RAN [The Proceedings of ISP RAS], 2002, vol.3, pp. 7-38 (in Russian).

[4]. Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report. University of Virginia, Charlottesville, VA, USA., 18 pages

[5]. Ilya N. Ledovskikh, Maxim G. Bakulin. Podkhod k vosstanovleniyu potoka upravleniya zaputannoj programmy. [An Approach to Reconstruction Control Flow of the Obfuscated Program] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 153-168 (in Russian).

[6]. N.P. Varnovskij, A.V. Shokurov. Gomomorfnoe shifrovanie. [Homomorphic encryption]. Trudy ISP RAN [The Proceedings of ISP RAS], 2007. Vol. 12, pp. 27-36. (in Russian).
[7]. Christian Collberg. Jasvir Nagra Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional. Pub. Date: July 24, 2009. Print ISBN-10: 0-321-54925-2.
[8]. N.P. Varnovskij, V.A. Zakharov, N.N. Kuzyurin, A.V. Chernov, A.V. Shokurov. Ob osobennostyakh primeneniya metodov obfuskatsii programm dlya informatsionnoj zashhity mikroehlektronnykh skhem. [About usage features of the program obfuscation techniques applying to informational microelectronic circuits security]. Trudy ISP RAN [The Proceedings of ISP RAS], 2006, vol. 11, pp. 27-60 (in Russian).
[9]. A.V. Chernov Ob odnom metode maskirovki programm [About one method program masking], Trudy ISP RAN [The Proceedings of ISP RAS], 2003, vol.4, pp. 85-119 (in Russian).
[10]. M.G. Bakulin, S.S. Gaissaryan, Sh.F. Kurmangaleev, I.N. Ledovskikh, V.A. Padaryan, S.M. Shchevyeva. Dinamicheskij analiz obfustsirovannykh prilozhenij s dispetcherizatsiej ili virtualizatsiej koda. [Dynamic analysis of virtualization- or dispatching-obfuscated applications]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012,vol. 23, pp. 49-66. (in Russian).

Повышение уровня представления трасс выполнения программ[1]

Назаров А. Г., Климушенкова М. А., Довгалюк П. М., Макаров В. А.,
Anton.Nazarov@ispras.ru, <Maria.Klimushenkova@ispras.ru>,
<Pavel.Dovgaluk@ispras.ru>, <Vladimir_Makarov@ispras.ru>

Аннотация. Использование динамического анализа для понимания поведения программы является распространенной практикой в наши дни. Одним из видов динамического анализа является анализ трасс выполнения программ. В то же время, трассы выполнения программ, состоящие из последовательности выполненных процессором инструкций, слишком велики для непосредственного восприятия человеком. Поэтому актуальной является задача повышения уровня представления таких трасс. В данной статье предлагается метод повышения уровня представления, предназначенный для построения модели алгоритма по трассе выполнения программы.

Ключевые слова. динамический анализ; восстановление переменных; восстановление интерфейсов; декомпиляция.

1. Введение

Динамический анализ – это анализ данных, полученных от выполняющейся программы. Он позволяет выявить те свойства программы, которые недоступны при анализе ее исходных текстов из-за их сложности, намеренной запутанности или позднего связывания. В случае, если исходные тексты программы недоступны, применяется динамический анализ бинарного кода.

Один из видов динамического анализа – анализ трасс выполнения программ. В случае выполнения анализа бинарного кода трасса представляет собой последовательность машинных инструкций в порядке их выполнения в программе. Но для того, чтобы понять работу алгоритмов, попавших в трассу, необходимо применять методы выделения из нее высокоуровневых сущностей и их фильтрации. Повышение уровня представления и фильтрация уменьшают объем обозреваемых данных и упрощают анализ алгоритма.

Динамический анализ трасс выполнения программ имеет как достоинства, так и недостатки, по сравнению со статическим анализом исполняемого кода. Недостаток заключается в том, что трасса не содержит инструкций из тех участков кода, которые не выполнялись во время снятия трассы. Достоинство

[1] Работа выполнена при поддержке РФФИ (грант 11-07-00353)

же трасс выполнения в том, что они содержат актуальные пути выполнения программы, а также информацию о значениях регистров.

Повышение уровня представления программы от бинарного кода до С-подобного языка называется декомпиляцией. Существует ряд реализаций, выполняющих декомпиляцию бинарного кода, например [[1]]. Однако, данные подходы не применялись в случае использования динамического анализа.

В данной статье предлагается метод повышения уровня представления трасс выполнения программ, основанный на выделении из них функций и структур данных. Предлагаемый подход использует уже достигнутые результаты по извлечению алгоритма из бинарного кода, описанные в [[2]]. Предполагается создание средства для анализа трасс выполнения программ, позволяющего просматривать их не в виде машинного кода, а в виде конструкций и переменных на С-подобном языке. Пользуясь таким представлением, можно строить высокоуровневую модель алгоритма.

2. Восстановление модели алгоритма

В данном разделе описывается общий подход к решению задачи восстановления модели алгоритма по трассам, и описываются задачи, решаемые авторами в рамках данного подхода. Под **моделью алгоритма** в конечном итоге понимается статическое высокоуровневое представление алгоритма, полученное в результате анализа трассы или множества трасс исполнения программы.

Реализация алгоритмов данного подхода выполняется в виде модулей для системы динамического анализа бинарного кода TrEx [[2]]. Общий комбинированный (статический и динамический) подход к анализу ПО с помощью данной системы и интерактивного дизассемблера IDA Pro описан в статье [[3]].

Общая схема предлагаемого подхода изображена в виде диаграммы потоков данных на рис. 1. Элементы диаграммы, нарисованные пунктирными линиями, находятся на стадии разработки и на момент написания статьи, не имеют реализации в системе TrEx.

2.1. Общее описание процесса восстановления алгоритма

На рис. 1 изображена последовательность выполняемых алгоритмов (вместе с их входными и выходными данными), составляющих предлагаемый процесс восстановления модели алгоритма. Чтобы описать эту последовательность понадобится использовать ряд терминов. Следующие понятия были подробно рассмотрены в других работах, касающихся среды TrEx [[2]].

Трасса (Trace) – трасса исполнения анализируемой программы, содержащая последовательность выполненных процессором инструкций и значения регистров на каждом таком шаге. Для увеличения покрытия кода

анализируемой программы может сниматься несколько трасс с различными входными данными.

Логическая модель функции (Function model) – формальная низкоуровневая модель функции (логический блок), которая может содержать списки входных и выходных логических параметров, описание зависимостей типа «вход-выход» между логическими параметрами, относительный адрес точки входа в функцию и относительные адреса возможных точек выхода функции. В процессе восстановления модели алгоритма необходимо выделять в трассе функции, имеющие отношение к исследуемому алгоритму, и описывать их формальные модели.

Логический параметр формальной модели (Logic parameter) – низкоуровневый элемент адресного пространства являющийся входом и/или выходом функции, описываемой соответствующей формальной моделью. В процессе восстановления модели алгоритма необходимо восстанавливать зависимости по данным между логическими параметрами и детализировать их, чтобы определить способ преобразования входов в выходы для заданной модели, то есть функцию преобразования значений входных параметров в значения выходных параметров.

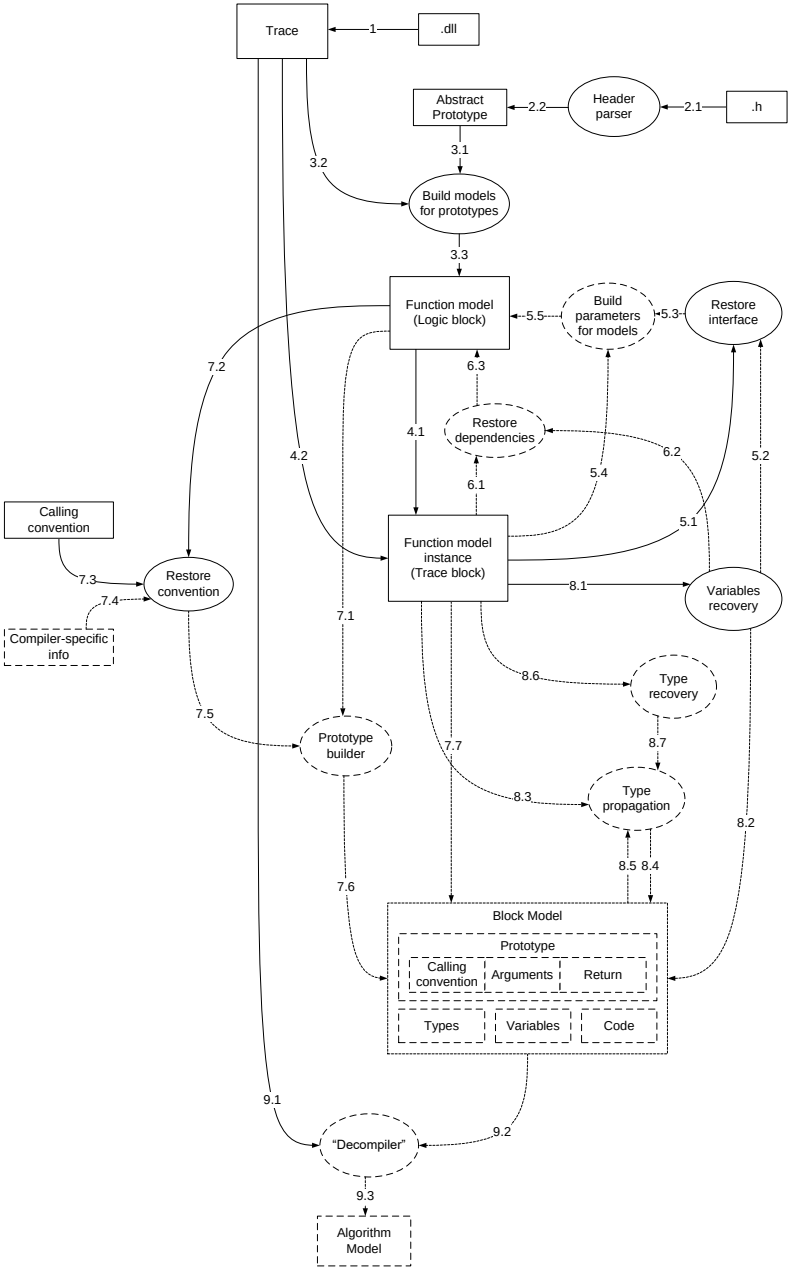


Рис. 1. Диаграмма потоков данных при автоматическом восстановлении модели алгоритма по трассе.

Блок трассы (Function model instance) – реализация логической модели функции, которая соответствует вызову данной функции в трассе и содержит списки реализаций входных и выходных логических параметров с восстановленными (по возможности) значениями при входе и выходе из блока соответственно, описание фактических зависимостей между входами и выходами, а также адрес начала и адрес конца блока в соответствующей трассе. В процессе восстановления модели алгоритма формальные модели строятся при наличии их реализаций (то есть последовательностей инструкций, соответствующих вызовам функции) в трассе (от блоков трассы для вызовов функций к формальным моделям функций). По соответствующим блокам трассы строятся логические параметры и восстанавливаются зависимости по данным между этими параметрами, а также по возможности восстанавливаются их значения.

Следующие понятия ранее в системе TrEx не использовались и вводятся впервые:

Абстрактный прототип (Abstract prototype) – высокоуровневый, абстрактный с точки зрения трассы, прототип функции, полученный в результате разбора заголовочного файла на языке С. Абстрактный прототип функции содержит: имя функции, тип возвращаемого значения, идентификатор используемого соглашения о вызовах (ключевое слово), типизированный список аргументов функции. В процессе восстановления модели алгоритма абстрактные прототипы известных библиотечных функций, вызовы которых присутствуют в трассе, извлекаются из соответствующих заголовочных файлов и являются источником достоверной информации о типах данных высокоуровневых аргументов функций.

Абстрактный аргумент функции (Abstract argument) – абстрактный с точки зрения трассы аргумент функции, полученный в результате разбора заголовочного файла и принадлежащий абстрактному прототипу функции. Абстрактный аргумент функции состоит из имени и типа.

Модель функционального блока (Block model) – высокоуровневое представление функционального блока (функции), содержащее в себе информацию о конкретном функциональном блоке анализируемого алгоритма, которая необходима для построения модели алгоритма в целом. Содержит в себе информацию из абстрактного прототипа (имя функции, типы данных аргументов, и др.), карту размещения абстрактных аргументов на логические параметры формальной модели, восстановленную информацию об используемых переменных (локальных и глобальных) и типах данных, а также высокоуровневое представление зависимостей между аргументами (входами функционального блока) и результатом (выходами функционального блока). В процессе восстановления модели алгоритма строятся модели всех функциональных блоков, являющихся частями исследуемого алгоритма, а также устанавливается характер взаимодействия между отдельными блоками.

Модель алгоритма (Algorithm model) – представление алгоритма в терминах языка программирования высокого уровня. Является конечным результатом анализа трассы и может быть передана пользователям, являющимися специалистами в предметной области, к которой относится работа алгоритма.

Также на схеме фигурирует **информация, специфичная для конкретного компилятора/платформы** (например, размеры встроенных типов данных). Она используется вместе с описаниями соглашений о вызовах в процессе разметки абстрактных аргументов функции на логические параметры формальной модели этой же функции. В рамках описываемого в данной статье подхода компилятор указывается пользователем вручную (используются только компиляторы языков C/C++). В статье [[4]] предлагается подход к идентификации компилятора по бинарному файлу программы, основывающийся на методах кластеризации идиоматических байтовых структур, получаемых на выходе различных компиляторов. При правильном развитии данного подхода его можно применить для анализа трасс выполнения программ.

На диаграмме (рис. 1) обозначены следующие алгоритмы:

- алгоритм идентификации интерфейса блока трассы (**Restore interface**), описанный ранее в [[5]];
- алгоритм восстановления соглашения о вызовах (**Restore convention**);
- алгоритм разбора заголовочных файлов для извлечения объявлений функций (**Header parser**), описанный более подробно в разделе 3;
- алгоритм построения формальных моделей для функций, которым поставлены в соответствие прототипы (**Build models for prototypes**);
- алгоритм построения логических параметров для формальных моделей (**Build parameters for models**);
- алгоритм восстановления зависимостей между логическими параметрами формальных моделей (**Restore dependencies**);
- алгоритм построения прототипа функции, то есть размещения аргументов абстрактного прототипа на низкоуровневые параметры (**Prototype builder**);
- алгоритм восстановления переменных (**Variables recovery**). Его задачей является выделение всех элементов памяти (ячеек памяти и регистров), которые используются в функции. Среди них выделяются входные и выходные элементы. Входными являются элементы, которые используются без инициализации внутри функции, к ним будут относиться параметры вызова, статические и динамические переменные. Выходными являются элементы, которые получили свое значение внутри функции, и используются вне функции, к

ним будут относиться возвращаемое значение функции, статические и динамические переменные. Более подробно алгоритм описан в разделе 4;

- алгоритм восстановления базовых типов данных (**Type recovery**). Размеры регистров/ячеек памяти, на которых размещаются переменные, а также семантика инструкций ассемблера накладывают ограничения на типы своих операндов. Анализируя эти ограничения, восстанавливаются типы некоторых переменных. Этот анализ позволяет получить информацию о типах переменных тех функций, для которых не известны прототипы;
- алгоритм распространения типов данных (**Type propagation**). Типы распространяются внутри функции и между функциями в трассе. Информация о типах передается между функциями с помощью параметров, возвращаемых значений и глобальных переменных;
- обобщённый алгоритм автоматического восстановления модели алгоритма по моделям функциональных блоков (**«Decompiler»**) представлен на диаграмме как некоторый промежуточный алгоритм построения модели алгоритма по моделям функциональных блоков, входящих в этот алгоритм (в процессе дальнейшей работы этот алгоритм будет разрабатываться более детально).

Решение общей задачи восстановления модели алгоритмов состоит из нескольких этапов. На диаграмме потоков данных потоки данных пронумерованы в соответствии с номерами этапов, которые описываются далее.

1. Извлечение символьной информации из бинарных файлов библиотек и привязка символов к соответствующим функциям в трассе.

2. Извлечение объявлений библиотечных функций из заголовочных файлов, построение абстрактных прототипов и их привязка к функциям в трассе (по имени из прототипа и соответствующему символу). При наличии перегруженных функций (актуально для C++) возникает проблема декорирования имён функций (name mangling). Для решения этой проблемы необходимо разработать и реализовать алгоритм выполнения операции обратной декорированию имени функции (name demangling). В результате выполнения такого алгоритма по декорированному имени функции (символу) можно получить объявление функции, по которому можно построить корректный абстрактный прототип и привязать его к соответствующей функции в трассе.

3. Построение логических моделей для функций в трассе, которым поставлен в соответствие абстрактный прототип. На данном этапе

модели не содержат информации о логических параметрах и о зависимостях между логическими параметрами.

4. Построение блоков трассы для логических моделей, построенных на предыдущем этапе, по вызовам соответствующих функций.

5. Для каждой логической модели функции выполняется восстановление интерфейса каждого блока трассы, соответствующего этой модели, и строятся её логические параметры.

6. Для каждой формальной модели выполняется восстановление зависимостей между её логическими параметрами.

7. Для каждой формальной модели при необходимости выполняется восстановление соглашения о вызовах (если не указано в абстрактном прототипе или не указано вручную пользователем), строится карта соответствия между аргументами абстрактного прототипа и логическими параметрами формальной модели функции и сохраняется в соответствующей модели функционального блока (**Block model**).

8. Восстановление переменных и типов данных для переменных, распространение типов (восстановленных и полученных из абстрактных прототипов функций) для функций, формальные модели которых построены на этапе 3. Все полученные данные по функциональным блокам сохраняются в соответствующих **Block model**.

9. Дальнейший анализ построенных моделей функциональных блоков для построения модели интересующего алгоритма.

3. Загрузка прототипов библиотечных функций

При снятии трассы выполнения некоторой анализируемой программы в трассу, кроме пользовательского кода, попадают вызовы функций различных библиотек (модулей). Описание таких функций, как правило, известно, то есть существует возможность получить их полные прототипы (то есть число параметров и их типы). Основная идея данного подхода заключается в подключении достоверной информации о типах данных, получаемой из описаний библиотечных функций, к трассе. В дальнейшем такую информацию можно распространить, используя методы распространения семантики данных (например, описанные в [[6]]). Таким образом, достоверная информация о типах данных может быть получена и в точках соприкосновения с кодом исследуемого алгоритма (например, вызовы библиотечных функций в пользовательском коде), что позволит выполнять анализ интересующих алгоритмов более высоком уровне представления кода.

3.1. Ограничения

В рамках предлагаемого подхода анализируются вызовы функций в трассе, соответствующие библиотечным функциям, то есть функции, для которых в

трассе присутствует символьная информация (имя функции или декорированное имя функции). Здесь речь идёт о функциях, чьи прототипы описаны в заголовочных файлах на языке C, и экспортируемых различными статическими или динамическими библиотеками. Соответственно система типов данных, используемая в разрабатываемых алгоритмах, основывается на системе типов языковой группы C.

Некоторые задачи (например, задача определения соглашения о вызовах для функции в трассе при отсутствии информации о соглашении в соответствующем заголовочном файле) на момент написания статьи не имеют автоматического решения, поэтому для их решения требуется интерактивный режим работы алгоритмов, позволяющий определять некоторые параметры вручную.

3.2. Реализация

Разбор заголовочных файлов – одна из задач, реализация которых требуется для внедрения предлагаемого подхода в среду TrEx. Абстрактные прототипы функций, извлеченные из заголовочных файлов, используются в качестве достоверного источника информации о типах данных параметров и возвращаемых значений.

Для преобразования заголовочных файлов из текстового формата в формат абстрактного синтаксического дерева используется свободная библиотека синтаксического анализа clang [[7]].

Полученное абстрактное синтаксическое дерево содержит в себе информацию о прототипах библиотечных функций: их имена, названия используемых соглашений о вызовах, названия и типы параметров, тип возвращаемого значения. Затем эта информация преобразуется в абстрактный прототип и сохраняется во внутреннем хранилище данных системы динамического анализа TrEx.

Таким образом, была выполнена реализация алгоритма загрузки прототипов функций. Она создает абстрактные прототипы, содержащие имена функций, а также имена и типы параметров. Реализация поддерживает работу с базовыми типами параметров и возвращаемых значений. В дальнейшем планируется расширение модуля для извлечения информации и о структурах данных, а также указателях на них.

4. Восстановление переменных

Алгоритм восстановления переменных был реализован в виде модуля к системе динамического анализа бинарного кода TrEx. Основными задачами модуля восстановления переменных являются:

- Разделение ограниченного числа регистров и повторно использующихся ячеек памяти на переменные
- Определение времени жизни переменных

- Соединение информации о переменных из разных вызовов одной и той же функции
- Определение области памяти, в которой находится переменная

Каждый процессор имеет определенный набор регистров и ограниченный объем памяти, а количество высокоуровневых переменных в программе не ограниченно [[8]], [[9]], [[10]]. Вследствие этого в одном регистре/ячейке памяти в разные моменты времени могут быть расположены разные переменные.

Прежде чем восстанавливать типы, необходимо выделить из низкоуровневых объектов (регистров/ячеек памяти) высокоуровневые объекты, представляющие собой переменные. В большинстве инструментов, которые занимаются восстановлением высокоуровневых объектов данных и их типов с помощью методов статического и динамического анализа, принципы такого разделения очень схожи. Первоначально переменные определяются в SSA форме [[11]], [[12]]. Это приводит к созданию большого количества переменных, с которыми сложно работать, поэтому их нужно объединить в логические объекты. В описываемом модуле переменные объединяются на основании утверждения, что модификация значения (изменение значения с участием старого значения) элемента памяти не является новым определением и не приводит к созданию новой переменной.

Другим подходом [[13]] является выделение переменной как компоненты связности графа определение-использование, построенного для всех используемых элементов. Такой подход преследует те же цели, что и описанный выше, то есть определение диапазона инструкций, когда ячейка памяти или регистр представляют собой одну логическую сущность.

Во время компиляции переменные языка Си могут быть размещены в статической памяти, на стеке, на регистрах и в динамической памяти. Куда в память компилятор разместит переменную зависит от ее времени жизни. Переменные с глобальным временем жизни, то есть глобальные переменные и локальные переменные со спецификатором static, размещаются в статической памяти. Память для них остается зарезервированной до конца выполнения программы. Переменные с локальным временем жизни, то есть локальные переменные и параметры функций, размещаются на стеке или на регистрах. Параметры могут передаваться на регистрах или в стеке. Для того, чтобы явно отделить параметры вызова и возвращаемое значение от глобальных переменных и сохраняемых регистров, нужно знать соглашение о вызовах, используемое компилятором.

В разработанном алгоритме под временем жизни переменной понимается диапазон шагов от создания переменной до ее последнего использования/изменения. Переменная создается в случае присваивания нового значения элементу памяти (регистру или ячейке памяти). Изменение,

то есть формирование нового значения с участием старого, не приводит к созданию новой переменной. Входные элементы функции определяются как элементы, использующиеся в функции, время жизни которых началось до вызова функции. Время жизни выходных элементов начинается до инструкции возврата из функции и заканчивается после.

В конструкциях ветвления выполнение программы может пойти по разным веткам. В каждой такой ветке одна и та же ячейка памяти или регистр может получать новое значение, и как следствие на основе этого будет созданы новые переменные. При соединении этих путей переменные объединяются.

В трассе может быть несколько проходов одной функции, а в разных вызовах могут быть различные пути выполнения. После анализа отдельных вызовов, вся информация объединяется, трассы вызовов накладываются друг на друга [[14]]. Чтобы объединить информацию о переменных нужно разделить память по типам. У статической переменной в каждом вызове будет один и тот же адрес. По одному адресу всегда находится одна переменная, но у нее могут быть разные типы, на пример, в случае с union. Стековые переменные будут иметь разный адрес, но постоянное смещение. У динамических переменных при каждом заходе может быть разный адрес, и по одному адресу могут располагаться разные переменные.

Модуль восстановления переменных предполагает плоскую модель памяти, где каждому процессу выделяется свое виртуальное адресное пространство, в которое загружается код приложения, секции данных, все необходимые библиотеки, а также выделяется области под стек и динамическую память. Загруженные в память библиотеки имеют свою статическую память, но пользуются общим стеком и кучей динамической памяти, хотя могут иметь и свой менеджер памяти.

Секции статических данных представляют собой непрерывные области памяти постоянного размера, которые содержат инициализированные и неинициализированные данные.

Область, отведенная под динамическую память, не является непрерывной, ее размер изменяется в зависимости от потребностей исполняемой программы. Так как динамическая память организована по принципу кучи, участки выделяемые менеджером памяти непоследовательны, а так же могут принадлежать разным областям (например, выше и ниже стека).

Стек представляет собой непрерывную область памяти, у которой есть начальный адрес ("дно" стека). Область стека не имеет определенного размера, она может увеличиваться или уменьшаться в зависимости от потребностей программы. Стек как правило "растет вниз", т.е. адрес следующей выделенной ячейки меньше адреса предыдущей, но может "расти вверх". Область стека может увеличиваться, пока есть свободное место, которое может быть занято растущей областью динамической памяти. В разные моменты времени одни и те же участки памяти могут занимать как стек, так и динамическая память [[8]], [[9]], [[10]].

Сложно отделить статическую память от динамической. Поскольку область статических данных является секцией загружаемого модуля, то можно по адресу загрузки модуля и его размеру определить принадлежит ли рассматриваемая ячейка статической памяти модуля. Где менеджер памяти может выделить место для динамических переменных понять сложно, поэтому необходимо находить функции управления памятью и выполнять анализ их использования.

Во время выполнения программы стек занимает область памяти от "дна" стека до текущего значения указателя стека. Потоки разделяют общее адресное пространство процесса, но каждый имеет свой стек. Так же режим ядра и режим пользователя имеют свой стек. В трассе можно определить наибольшее значение указателя стека в определенном потоке в режиме пользователя, и считать его "дном", тогда ячейки, адрес которых попадает в диапазон между этим значением и текущем значением указателя стека, точно находятся в стеке. Однако, этот способ не сможет идентифицировать все ячейки, если запись трассы началась не с начала выполнения программы.

При вызове функции в стек сохраняются адрес возврата, сохраняемые регистры (если это требуется), выделяется место для локальных переменных. При многократных вызовах функции указатель стека в начале функции может быть разным, и как следствие, адреса локальных переменных будут разными. Постоянным будет лишь смещение относительно начала stack frame. Поэтому элементы стека, использующиеся в функции, идентифицируются по смещению.

В других инструментах, восстанавливающих высокоуровневые объекты данных, глобальные статические переменные определяются либо по шаблонам обращения к памяти (обращение по абсолютному адресу) [[13]], [[15]], [[16]], либо с помощью анализа секций данных исполняемого файла [[17]]. Локальные переменные, как правило, определяются по обращению с использованием указателя стека или его копии. Для выделения участков динамической памяти, ищутся вызовы функций malloc и free.

5. Заключение

В статье был описан метод повышению уровня представления трасс выполнения программ для построения моделей алгоритмов. Данный метод включает в себя работу нескольких модулей. Модули для выделения локальных переменных, параметров функций, а также загрузки прототипов функций из заголовочных файлов были реализованы для среды динамического анализа бинарного кода TrEx.

В дальнейшем планируется реализовать остальные модули для системы TrEx, позволяющие повысить уровень представления трассы с машинных инструкций до C-подобного языка, такие как восстановление базовых и структурных типов данных, восстановление прототипов функций с привязкой

их к переменным, обнаруженным к трассе, а также восстановление высокоуровневых управляющих конструкций и арифметических выражений.

Полученное высокоуровневое представление позволит строить модель алгоритма программы, которую можно использовать для обнаружения закладок и недокументированных возможностей, поиска обратного алгоритма, а также восстановления алгоритма при отсутствии исходных текстов программы.

Список литературы

[1] C. Cifuentes and K. J. Gough. Decompilation of binary programs. Softw. Pract. Exper., 25(7):811–829, 1995. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.4380250706, дата обращения 31.10.2012

[2] В.А. Падарян, А.И. Гетьман, М.А. Соловьев. Программная среда для динамического анализа бинарного кода. Труды Института системного программирования РАН, том 17, 2009 г. Стр. 51-72

[3] А.Ю.Тихонов, А.И. Аветисян. Комбинированный (статический и динамический) анализ бинарного кода. Труды Института системного программирования РАН, том 22, 2012 г. Стр. 131-152.

[4] Nathan E. Rosenblum, Barton P. Miller, Xiaojin Zhu. Extracting compiler provenance from program binaries. Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. Toronto, Ontario, Canada, 2010, pp. 21-28

[5] А.Г. Назаров. Автоматическая идентификация интерфейса блока трассы в задаче восстановления модели алгоритма. **Вестник НовГУ выпуск №62. 2012 стр. 41 - 50**

[6] J. Caballero, N.M. Johnson, S.McCamant, and D. Song. Binary code extraction and interface identification for security applications. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, February 2010, 1-18 pp

[7] clang: a C language family frontend for LLVM. http://clang.llvm.org/, дата обращения 31.10.2012

[8] Intel® 64 and IA-32 Architectures Software Developer's Manual. http://download.intel.com/products/processor/manual/325462.pdf, дата обращения 31.10.2012

[9] ARM® v7-M Architecture Reference Manual. https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR580-DC-11001-r0p0-02rel0/DDI0403D_arm_architecture_v7m_reference_manual_errata_markup_1_0.pdf, дата обращения 31.10.2012

[10] MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS64® Architecture. http://www.mips.com/auth/MD00083-2B-MIPS64INT-AFP-03.02.pdf, дата обращения 31.10.2012

[11] Климушенкова М.А. Методы восстановления объектов данных из бинарного кода. **Вестник НовГУ выпуск №62. 2012 стр. 51 - 57**

[12] Alan Mycroft. Type-based decompilation. Proceedings of European Symposium on Programming, Mar. 1999, pp. 208 – 223

[13] E. Dolgova and A. Chernov. Automatic reconstruction of data types in the decompilation problem. Programming and Computer Software 35, Mar. 2009, pp. 105 – 119

[14] JongHyup Lee, Thanassis Avgerinos and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. Proceedings of the Distributed System Security Symposium, NDSS 2011, pp. 1 – 18
[15] G. Balakrishnan and T. Reps. DIVINE: Discovering variables in executables. Proceedings of the International Conference on Verification Model Checking and Abstract Interpretation, Jan. 2007, pp. 1 - 28
[16] G. Balakrishnan. WYSINWYX: What You See Is Not What You eXecute. PhD thesis, Computer Science Department, University of Wisconsin at Madison, Aug. 2007, pp. 20 - 97
[17] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. Proceedings of the Network and Distributed System Security Symposium, 2010, pp. 1 - 18

Raising the level of abstraction of program execution trace

Nazarov A.G., Klimushenkova M.A., Dovgalyuk P.M., Makarov V.A.
{Anton.Nazarov, Maria.Klimushenkova, Pavel.Dovgaluk,
Vladimir_Makarov}@ispras.ru
ISP RAS, Moscow, Russia

Abstract. This paper presents a method for raising the level of abstraction of a program execution trace by building of an algorithm model.

Dynamic analysis of executable files is often used to understand the logic of a program in the absence of source code. One of dynamic analysis methods is analysis of program execution traces containing register values and sequence of instructions executed by the processor. Traces are difficult for understanding because of the large amount of information.

First stage of building an algorithm model is identification of function calls in the trace. Input and output parameters are determined for each call. If the trace has got several calls for the same function, the information about them is combined, defining low-level parameters, return value, and dependencies between inputs and outputs.

Second stage is variable recovery. Low-level data elements are mapped on variables. Each processor has a fixed set of registers and limited memory and the number of higher-level variables in the program is not limited. Variable lifetime is evaluated for mapping variables to their locations. Lifetime is a range of trace step indexes from variable creation to its last usage. Return value and parameters of the function are recovered using its calling convention.

Third stage is examination of library calls that are used in the trace. Symbolic information can be extracted from binary libraries and added to the corresponding functions in the trace. Header files are available for some libraries. Full high level function prototypes can be found from them and mapped to the trace.

This allows us to get high level types of parameters that are propagated along the trace through global variables and function calls.

Function models are combined into a high level model algorithm that can be used to restore or analyse it.

Keywords: dynamic analysis, variables recovery, interfaces recovery, decompilation

References

[1]. Cifuentes C., Gough K. J. Decompilation of binary programs. Softw. Pract. Exper., 25(7):811–829, 1995. ISSN 0038-0644. doi:10.1002/spe.4380250706

[2]. Padaryan V.A., Get'man A.I., Solov'ev M.A. Programmnaya sreda dlya dinamicheskogo analiza binarnogo koda [Software environment for dynamic analysis of binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], 2009, vol. 17, pp. 51-72 (in Russian).

[3]. Tikhonov A.YU., Avetisyan A.I. Kombinirovannyj (staticheskij i dinamicheskij) analiz binarnogo koda [Combined (static and dynamic) analysis of binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 131-152 (in Russian).

[4]. Nathan E. Rosenblum, Barton P. Miller, Xiaojin Zhu. Extracting compiler provenance from program binaries. Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on

Program analysis for software tools and engineering. Toronto, Ontario, Canada, 2010, pp. 21-28
[5]. Nazarov A.G. Avtomaticheskaya identifikatsiya interfejsa bloka trassy v zadache vosstanovleniya modeli algoritma [Automatic interface identification of trace block in recovery algorithm model task]. Vestnik NovGU [Vestnik NovGU], 2012, vol. 62, pp. 41 - 50 (in Russian).
[6]. J. Caballero, N.M. Johnson,S.McCamant, and D. Song. Binary code extraction and interface identification for security applications. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, February 2010, 1-18 pp
[7]. clang: a C language family frontend for LLVM. clang.llvm.org
[8]. Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual. download.intel.com/products/processor/manual/325462.pdf
[9]. ARM® v7-M Architecture Reference Manual. silver.arm.com/download/ARM_and_AMBA_Architecture/AR580-DC-11001-r0p0-02rel0/DDI0403D_arm_architecture_v7m_reference_manual_errata_markup_1_0.pdf
[10]. MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS64® Architecture. www.mips.com/auth/MD00083-2B-MIPS64INT-AFP-03.02.pdf
[11]. Klimushenkova M.A. Metody vosstanovleniya ob"ektov dannykh iz binarnogo koda [Methods for data objects recovery from a binary code]. Vestnik NovGU [Vestnik NovGU], 2012, vol. 62, pp. 51 - 57 (in Russian).
[12]. Alan Mycroft. Type-based decompilation. Proceedings of European Symposium on Programming, Mar. 1999, pp. 208 – 223
[13]. E. Dolgova and A. Chernov. Automatic reconstruction of data types in the decompilation problem. Programming and Computer Software 35, Mar. 2009, pp. 105 – 119
[14]. JongHyup Lee, Thanassis Avgerinos and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs.Proceedings of the Distributed System Security Symposium, NDSS 2011, pp. 1 – 18
[15]. G. Balakrishnan and T. Reps. DIVINE: Discovering variables in executables. Proceedings of the International Conference on Verification Model Checking and Abstract Interpretation, Jan. 2007, pp. 1 - 28
[16]. G. Balakrishnan. WYSINWYX: What You See Is Not What You eXecute. PhD thesis, Computer Science Department, University of Wisconsin at Madison,Aug. 2007, pp. 20 – 97
[17]. Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. Proceedings of the Network and Distributed System Security Symposium, 2010, pp. 1 – 18

Подход для проведения рефакторинга «Выделение функции» в инструменте Klocwork Insight

Н.Л. Луговской
lugovskoy@ispras.ru

Аннотация. В статье рассматривается подход для проведения рефакторинга исходного кода на языках Си/Си++, реализованный в инструменте Klocwork Insight. Приводится подробное описание подхода на примере рефакторинга «Выделение функции». Разбираются способы обработки различных языковых конструкций при проведении рефакторинга, показывается, как структурные изменения в синтаксическом дереве отображаются в изменения исходного кода программы. На основе описанного подхода для проведения рефакторинга предлагается выделить методы для реализации произвольных изменений в программе, выходящих за рамки широко используемых рефакторингов. В конце статьи проводится сравнение с существующими инструментами для проведения рефакторинга.

Ключевые слова. рефакторинг; выделение функции; трансформация кода; статический анализ

1. Введение

Задача раннего выявления ошибки в дизайне программы является важной темой для исследования, так как стоимость подобных ошибок впоследствии может оказаться достаточно высокой. Ряд решений для раннего обнаружения ошибок предлагают автоматические средства статического и динамического анализа кода [1]. Однако, они мало помогают в исправлении ошибок модели программы. Одним из самых распространенных методов улучшения модели программы является рефакторинг программного кода.

Рефакторинг — это процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы [2]. Как правило, он применяется для изменения дизайна программы с целью упрощения исходного кода, улучшения его понимания разработчиком и подготовки к добавлению новой функциональности. Более того, рефакторинг может применяться на ранних стадиях процесса построения программы. Важно отметить, что использование рефакторинга увеличивает возможности программиста по «расширению» программы, что, в свою

очередь, облегчает использование таких приемов проектирования, как паттерны проектирования.

Klocwork Insight - это инструмент статического анализа кода. Он обладает гибкой системой встраивания в сборку программного продукта, использует собственные синтаксический и семантический анализаторы и поддерживает множество расширений языков Си и Си++. Использование собственных анализаторов исходного кода дает большую свободу при разработке инструмента для проведения рефакторинга.

При описании нашего подхода и для сравнения с другими решениями мы будем использовать один из самых популярных и распространенных рефакторингов, а именно, «Выделение функции».

2. Рефакторинг «Выделение функции»

В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение программы) преобразований. Поскольку каждое из преобразований, как правило, незначительно влияет на изменение исходного кода программы, то программисту легче проследить за их корректностью, и в то же время вся последовательность преобразований может привести к существенной перестройке программы и улучшению её дизайна.

В известной книге «Рефакторинг. Улучшение существующего кода» [2] описываются такие классы рефакторингов, как «Составление методов», «Перемещение функций между объектами», «Организация данных», «Упрощение условных выражений», «Упрощение вызовов методов», «Решение задач обобщения». Стоит заметить, что количество различных видов рефакторинга постоянно растет, самый полный список рефакторингов можно найти на web-сайте [3]. В данной работе мы будем рассматривать один из самых популярных рефакторингов – «Выделение функции», который относится к классу рефакторингов «Составление методов».

Проводя рефакторинг «Выделение функции», программист создает новую функцию из выделенного фрагмента кода, который заменяется вызовом этой функции. Такая трансформация позволяет быстро и аккуратно реорганизовать исходный код для лучшей поддержки и читаемости [4]. Более короткое описание рефакторинга «Выделение функции» звучит так: «*У вас есть фрагмент кода, который может быть сгруппирован. Выделите этот фрагмент в функцию, чье имя объяснит ее смысл*» [5]. Применение рефакторинга «Выделение функции», очевидно, имеет положительный эффект для исходного кода программы. Например, возможность повторного использования выделенной функции уменьшает повторения одного и того же фрагмента кода; улучшается документация, т.к. имя выделенной функции объясняет ее смысл; более того, т.к. новые функции относительно малы и, как

правило, реализуют какую-то законченную функциональность, то программисту легче ими оперировать.

3. Существующие решения

Для удобной и быстрой, а главное, корректной разработки программных систем автоматические средства трансформации кода играют немаловажную роль. Они позволяют производить сложные структурные изменения исходного кода с минимальным участием программиста. При использовании такого автоматического инструмента уменьшается риск ошибки, что положительно влияет на качество кода.

Наибольшее развитие получили средства автоматического рефакторинга для языков Java и C#. Для языков Си/Си++ подобные средства развиты в гораздо меньшей степени. Это обусловлено сложностью и низкоуровненностью языков Си/Си++, например, присутствием таких конструкций, как оператор goto, глобальные переменные, указатели, адресная арифметика, ненадежное приведение типов, списки аргументов переменной длины и, особенно, использованием макросов. Отдельно стоит упомянуть, что в языках Си/Си++ используются заголовочные файлы и раздельная компиляция.

На данный момент существует небольшое число инструментов для проведения рефакторинга Си/Си++ кода. Их можно поделить на две группы: инструменты, реализованные как подключаемые модули для программной среды Microsoft Visual Studio, и встроенные компоненты сред разработки приложений. К первой группе относятся следующие инструменты:

- CodeRush - это программный инструмент, разработанный компанией DevExpres, который включает в себя такие возможности, как анализ и поиск ошибок в коде, усовершенствованный метод выделения языковых конструкций, генерация кода, средства визуализации, модуль для проведения рефакторинга Refactor! Pro и т.д. Инструмент поддерживает довольно большое количество рефакторингов, например, «Переименование функции», «Выделение функции», «Встраивание функции» [6].
- Visual Assist X - это продукт компании Whole Tomato, который включает в себя такие возможности, как рефакторинг кода, автоматическое дополнение языковых конструкций, добавляет новые виды подсветки, обладает улучшенной навигацией по сравнению со встроенной в программную среду Visual Studio, связывающей языковые сущности. Инструмент поддерживает около 12-ти рефакторингов, среди которых есть, например, «Переименование», «Выделение функции», «Инкапсуляция поля» [7].

Ко второй группе относятся следующие программные среды:

- Eclipse CDT - это свободная интегрированная среда разработки программ на языках C/C++, базирующаяся на платформе Eclipse.

Поддерживает довольно небольшое количество рефакторингов, среди которых «Выделение функции», «Переименование», «Выделение переменной» [8]. Среда разработки предоставляет интерфейс для написания подключаемых модулей, в котором присутствуют методы для проведения трансформации исходного кода [9].

- XCode - программа для разработки приложений под OS X и iOS, разработанная компанией Apple. Включает в себя среду разработки, документацию и измененную версию набора компиляторов GNU GCC и Clang. Для языка Си поддерживается два рефакторинга: «Переименование» и «Выделение функции» [10].

Существующие инструменты для проведения автоматического рефакторинга не гарантируют, что результат рефакторинга будет синтаксически корректен и сохранится первоначальное поведение программы. Как правило, они лишь проверяют несколько предусловий, чтобы отсечь случаи, когда проведение автоматического рефакторинга невозможно или требует слишком сложного анализа [11]. При этом окончательная проверка корректности результата ложится на программиста, что значительно снижает пользу от использования автоматического средства. Стоит ли упоминать о том, что существует человеческий фактор, который в этой ситуации также может привести к ошибке в программе. Описанный принцип не дает возможности использовать существующие инструменты для проведения автоматического рефакторинга в качестве повседневного инструмента программиста для «улучшения» программного кода.

4. Принцип для проведения рефакторинга, реализованный в Klocwork Insight

Так как ни одно из существующих решений не может использоваться как надежный инструмент для проведения рефакторинга, то был предложен другой принцип для проведения рефакторинга Си/Си++ кода, основным требованием к которому является сохранение корректности кода и его поведения после применения рефакторинга. Если инструмент не может гарантировать корректность результата проводимого рефакторинга, то он либо совсем отказывается его проводить, либо сообщает некую подсказку пользователю о потенциальной ошибке.

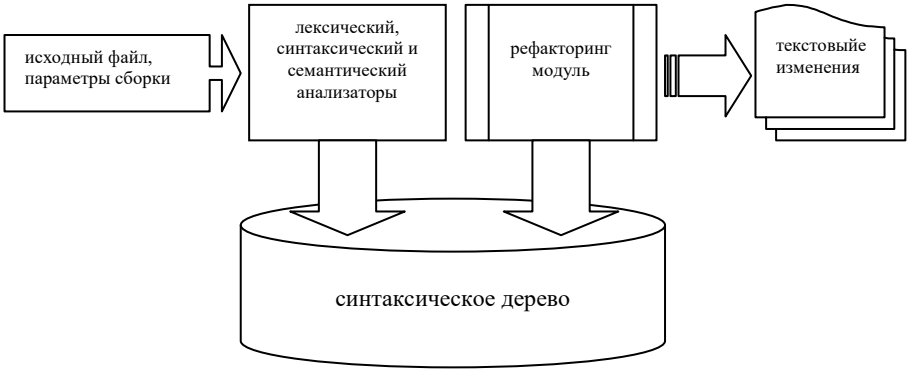
Придерживаясь такого принципа, нам удалось итеративно расширить рамки применимости нашего инструмента. В итоге, метод для проведения рефакторинга, реализованный в Klocwork Insight, поддерживает подавляющее количество языковых конструкций, сложные виды выделения, использует ряд эвристик для улучшения результата рефакторинга. В инструменте для проведения рефакторинга используется база знаний о языковых элементах и их текстовом представлении полученная напрямую от компилятора. Такая глубокая связка компилятора и инструмента для рефакторинга дает последнему возможность проводить рефакторинг предельно точно, вплоть до

сохранения пользовательской индентации и комментариев [12]. И, как будет видно из сравнительного анализа существующих инструментов для рефакторинга в главе 8, метод для проведения рефакторинга, реализованный в Klocwork Insight, во многом превосходит существующие аналоги.

4.1. Стадии процесса рефакторинга

Процесс рефакторинга исходного файла состоит из нескольких стадий. На первой стадии исходный файл с информацией, описывающей компиляцию этого файла (директории заголовочных файлов, определения макросов, специальные опции), подается на вход компилятору. В процессе компиляции последовательно запускаются лексический, синтаксический и семантический анализаторы. В результате работы анализаторов строится синтаксическое дерево исходного файла, которое передается в рефакторинг модуль. Далее, рефакторинг модуль анализирует внутреннее представление исходного файла и создает набор текстовых изменений для трансформации исходного кода. Таким образом, применяя правила из этого набора к исходному файлу, можно изменить исходный код соответственно заданному рефакторингу.

Схематически последовательность стадий процесса рефакторинга исходного файла можно изобразить схемой:



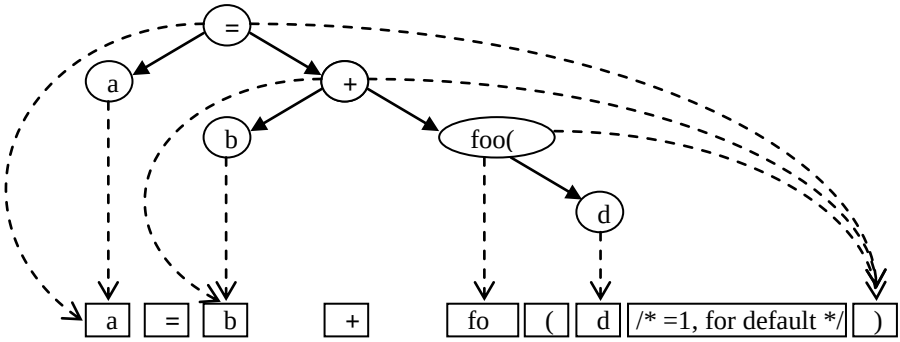
В инструменте Klocwork Insight присутствуют подключаемые модули для сред программирования Microsoft Visual Studio и Eclipse, что позволяет запускать все стадии рефакторинга автоматически. Единственное интерактивное общение с пользователем возможно только для уточнения параметров рефакторинга, например, имя новой переменной или функции.

4.2. Препроцессирование, лексический и семантический анализ

Итак, на вход компилятору подается исходный файл. На первой стадии анализа - в лексическом анализаторе, исходный файл разбивается на лексеммы-токены. Каждый токен - это одиночная атомарная частица языка, например, ключевое слово или идентификатор. Следующей стадией анализа

является препроцессирование. На этой стадии раскрываются макросы и выполняется условная компиляция. Далее, происходит синтаксический анализ последовательности токенов. На этой стадии определяется синтаксическая структура программы. И, как результат, из токенов строится синтаксическое дерево программы. Для построения синтаксического дерева используются правила, описанные в грамматике языка. Завершающей стадией компиляции является семантический анализ. На этой стадии семантический анализатор добавляет семантическую информацию к узлам синтаксического дерева. Семантический анализатор выполняет такие задачи, как проверка типов, связывание определения и использования переменных и функций, вычисление областей видимости. После стадии семантического анализа построенное синтаксическое дерево используется для анализа и проведения изменений, соответствующих выбранному рефакторингу.

Традиционно компиляторы не сохраняют последовательность токенов [13]. Она удаляется в момент построения синтаксического дерева. Однако, без этой последовательности невозможно однозначно восстановить исходный текст. Если пытаться строить исходный код программы по синтаксическому дереву, то неизбежна потеря «стиля» исходного кода программы. Более того, т.к. комментарии представляют из себя токены, но не имеют соответствующих узлов, то при генерации исходного кода программы по синтаксическому дереву комментарии будут утеряны. Вопрос о генерации исходного кода программы очень важен для рефакторинга, т.к. изменения на уровне синтаксического дерева необходимо преобразовывать в изменения на уровне исходного кода. Поэтому, чтобы сохранить оригинальное текстовое представление программы, синтаксический анализатор не удаляет токены, а связывает их с узлами синтаксического дерева. При таком подходе сохраняются и комментарии, и исходный стиль программного кода. Например, синтаксическое дерево и токен-последовательность для выражения «*a = b + foo(d /* =1, for default*/)*» будет выглядеть следующим образом:



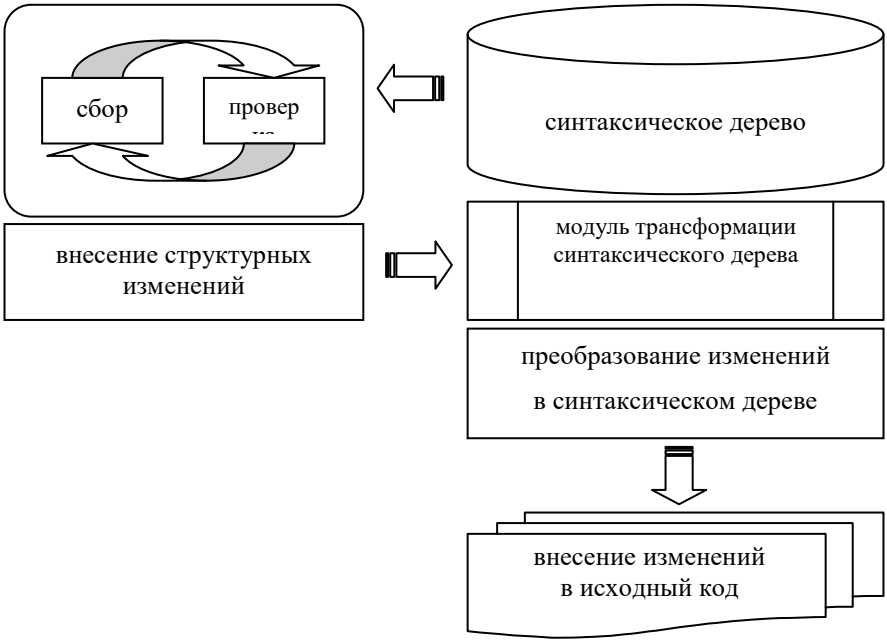
Можно добавить, что некоторые существующие анализаторы сохраняют также пробелы и переводы строк в виде токенов [14]. Однако, эта информация избыточна, т.к. ее можно получить из позиций обычных токенов.

4.3. Рефакторинг-модуль

Основным компонентом, где, собственно, и происходит анализ и построение изменений для выбранного рефакторинга, является Рефакторинг-модуль (далее РМ). Он анализирует синтаксическое дерево и, если дерево было построено корректно и без ошибок, создает набор текстовых изменений для трансформации исходного кода, необходимых для проведения рефакторинга. РМ также запрашивает у пользователя информацию о виде рефакторинга, его параметры. Например, для рефакторинга «Выделение функции» запрашивается имя новой функции. Работа РМ состоит из нескольких последовательных стадий:

- поиск данных в выделенном фрагменте и всей программе;
- проверка на корректность выделенного фрагмента кода, а также локальной, для выделенного фрагмента, области;
- внесение структурных изменений в синтаксическое дерево;
- преобразование изменений в синтаксическом дереве на уровень исходного кода.

Эту последовательность стадий удобно представить в виде схемы:



Первые три стадии работы РМ являются уникальными для каждого отдельного вида рефакторинга. Поиск и анализ данных, необходимых для проведения рефакторинга, происходит одновременно с проверкой на их корректность. Такими данными являются, например, используемые переменные, вызовы функций, декларации и т.д. Собрав необходимую «базу»

данных о программе и выделенном фрагменте, РМ проводит структурные изменения в синтаксическом дереве программы, которые соответствуют выбранному рефакторингу. На последней стадии сканируется синтаксическое дерево и для измененных частей создается текстовое представление для внесения изменений в исходный код. Эти текстовые представления могут быть приведены, например, к *нормальному формату* утилиты diff [15] для автоматического внесения изменений в файл.

5. Процесс рефакторинга «Выделение функции»

Процесс рефакторинга «Выделение функции» начинается со сбора и анализа данных о выделенном фрагменте. Главным образом, РМ собирает информацию обо всех переменных, находящихся внутри выделенного фрагмента. Например, необходимо определить следующие характеристики: тип переменной, входит ли в выделенный фрагмент ее декларация, является ли она аргументом функции, область видимости переменной, ее использование до и после выделенного фрагмента, различные виды использования (чтение, запись, перезапись, взятие адреса и т.д.).

Переменные, найденные в выделенном фрагменте кода, являются, по сути, основными данными для построения новой функции. Фактически, зная перечисленную информацию о выделенных переменных, можно уже определить сигнатуру новой функции. Далее достаточно поместить выделенный фрагмент кода внутрь тела новой функции и изменить в нем способ доступа к переменным. Такой подход является базовым при проведении рефакторинга «Выделение функции». Поддержку новых языковых конструкций, которые требуют дополнительного анализа, можно реализовывать постепенно, так как они не отменяют, а лишь добавляют соответствующие корректировки в базовый подход.

5.1. Анализ переменных в выделенном фрагменте кода

Чтобы собрать информацию о переменных внутри выделенного фрагмента кода, достаточно совершить обход синтаксического поддерева, соответствующего этому фрагменту. В рефакторинге «Выделение функции» заданный фрагмент кода всегда находится внутри функции, поэтому РМ обходит только синтаксическое поддерево, соответствующее этой функции.

Среди всех найденных переменных автоматически выделяются переменные, которые только читаются и только записываются. Их мы будет называть соответственно входными и выходными параметрами новой функции. Если выходная переменная всего одна, то она будет возвращаемым параметром. Остальные будут передаваться в функцию согласно виду использования. Например, входная целочисленная переменная должна передаваться в новую функцию по значению, а выходная по указателю или ссылке.

5.1.1 Использование переменной вне выделенного фрагмента

Хотя анализ выделенного фрагмента точно определяет, является переменная входной или выходной, этого бывает недостаточно, чтобы правильно определить новую функцию. Например:

```
int main() {
    int a = 0;
    if (bar())
        a = 1;
    return a;
}
```

При анализе выделенного фрагмента переменная «*a*» будет помечена как выходной параметр. Более того, так как это единственная выходная переменная, то она становится возвращаемым параметром. В результате может быть проведен ошибочный рефакторинг:

```
int extracted_function() {
    int a;
    if (bar())
        a = 1;
    return a;
}
int main() {
    int a = 0;
    a = extracted_function();
    return a;
}
```

Очевидно, что если «*bar() == 0*», то в переменную «*a*» попадут неинициализированные данные, что приведет к неправильной работе программы. Если же мы знаем об инициализации переменной перед выделенным фрагментом, то это поможет определить новую функцию правильно. В этом случае рефакторинг можно провести, например, так:

```
void extracted_function(int *a) {
    if (bar())
        *a = 1;
}
int main() {
    int a = 0;
    extracted_function(&a);
    return a;
}
```

Если внутри выделенного фрагмента содержится определение переменной, то необходимо знать об использовании переменной вне выделенного фрагмента. Без этой информации невозможно определить, стоит ли выносить определение переменной внутрь новой функции.

5.1.2 Обработка переменной-массива

Для выделенной переменной, являющейся массивом, необходимо, во-первых, корректно формировать ее тип, т.е. при передаче в качестве параметра двух и более размерного массива в ее типе указывать все размеры, за исключением главного. Во-вторых, необходимо отслеживать использование оператора «*sizeof*», так как это может привести к ошибке, например:

```
int main () {
    char buf[16];
    printf(«bufsize is %u», sizeof(buf));
    return 0;
}
```

В этом примере, новая функция может иметь вид:

```
void extracted_function(char buf[]) {
    printf(«bufsize is %u», sizeof(buf));
}
```

Но, по сути, это будет изменением поведения программы потому, что «*sizeof(char[])*» не равно «*sizeof(char[16])*». Поэтому новая функция должна иметь следующий вид:

```
void extracted_function(char buf[16]) {
    printf(«bufsize is %u», sizeof(buf));
}
```

5.1.3 Оператор взятия адреса

Если в выделенном фрагменте кода присутствует оператор взятия адреса от выделенной переменной, то она обязательно должна передаваться как аргумент в новую функцию, даже если она не используется за пределами выделенного фрагмента. Рассмотрим пример:

```
int main() {
    int *ptr;
    int port = 8080;
    ptr = &port;
    set_socket_port(*ptr);
    return 1;
}
```

Как видно из примера, переменная «*port*» не используется вне выделенного фрагмента, поэтому в результате рефакторинга «Выделение функции» она может быть вынесена в новую функцию следующим образом:

```
int *extracted_method() {
    int *ptr;
    int port = 8080;
    ptr = &port;
    return ptr;
}
int main() {
    int *ptr;
    ptr = extracted_method();
    set_socket_port(*ptr);
    return 1;
}
```

Что, очевидно, приведет к ошибке, т.к. указатель «*ptr*» будет указывать на данные, находящиеся в освобожденной части стека.

5.1.4 Зависимость от локальных директив и типов

В выделенном фрагменте кода могут присутствовать узлы синтаксического дерева, зависящие от локальных объявлений типов и using-директив. Локальными здесь названы директивы и объявления, определенные внутри функции. В этом случае рефакторинг «Выделение функции» может привести к ошибке, если выделенный фрагмента кода содержит зависимости от локальных объявлений и директив, которые не входят в этот фрагмент. Для того, чтобы провести корректный рефакторинг «Выделение функции» необходимо отслеживать такие зависимости и добавлять используемые объявления типов и using-директивы в начало новой функции. Рассмотрим пример:

```
string foo(vector<string> &v)
{
    typedef string MyString;
    for (int i = 0; i < 10; i++) {
        MyString s = get_string();
        v.push_back(s);
    }
    return v[2];
}
```

В этом примере рефакторинг «Выделение функции» нужно делать следующим образом:

```
void extracted_method(vector<string> &v)
{
    typedef string MyString;
    for (int i = 0; i < 10; i++) {
        MyString s = get_string();
        v.push_back(s);
    }
}
```

С другой стороны, если выделенный фрагмент кода содержит локальное объявление типа или using-директиву, то, перемещая их в новую функцию, можно нарушить зависимости, идущие к ним из-за пределов выделенного фрагмента. Поэтому, локальные объявления типов и using-директивы находящиеся внутри выделенного фрагмента кода не выносятся, а только копируются в новую функцию.

5.2. Построение функции

При построении новой функции первым делом определяется ее сигнатура, причем входные переменные передаются по значению, а выходные - через указатель или ссылку (для языка Си++). Сложные типы данных также передаются по указателю или ссылке. Определение новой функции помещается перед функцией, в которой производится рефакторинг. Таким образом, все внешние зависимости остаются ненарушенными (например, зависимость от определений типов и using-директив). Построение тела новой функции происходит в два этапа. На первом этапе синтаксическое поддерево, соответствующее выделенному фрагменту кода, копируется и помещается в тело новой функции. На втором этапе изменяются способы доступа к переменным в созданной функции. Если у функции есть возвращаемая переменная, то для нее создаются определение и return-узел для возвращения ее значения.

5.3. Декларации переменных

Если выделенный фрагмент кода содержит декларации переменных, то возникают ограничения, которые необходимо учитывать при проведении рефакторинга «Выделение функции». В простом случае, когда переменная не используется за пределами выделенного фрагмента, достаточно вынести ее декларацию в новую функцию. Если же она используется за пределами выделенного фрагмента, то необходимо проводить дополнительный анализ. Рассмотрим пример:

```
class A {
public:
    int v;
    A(int i) { v = i; }
```

```
private:
    A(const A &a) { v = a.v; }
};
int foo() {
    A a(10);
    return a.v;
}
```

Выделенный фрагмент кода может быть изменен в результате рефакторинга следующим образом:

```
A extracted_function() {
    A a(10);
    return a;
}
int foo() {
    A a = extracted_function();
    return a.v;
}
```

Если обратить внимание на определение класса «*A*», то можно заметить, что конструктор копирования объявлен в секции «*private*». С другой стороны, он будет вызван в функции «*foo()*» при инициализации переменной «a», что приведет к ошибке компиляции.

Для проведения корректного рефакторинга «Выделение функции» на фрагменте кода, в котором присутствуют декларации переменных, необходимо выделить декларации с инициализаторами и для каждой из них выбрать такой метод инициализации переменной, который будет использоваться в новой функции и не будет приводить к ошибке. В тех или иных случаях необходимо проверить конструктор по умолчанию, операторы присваивания, использование ссылок. Декларации переменных, у которых нет инициализатора, но которые присутствуют в выделенном фрагменте, не влияют на вид новой функции.

Для языка Си (стандарта C89) существует правило, по которому декларации переменных обязаны стоять вначале открытого блока кода. В этом случае, если выделенный фрагмент содержит только декларации переменных, причем за ним следуют другие декларации, то необходимо проверить используются ли выделенные переменные внутри инициализаторов деклараций, которые следуют за выделенным фрагментом. Например, для следующего выделенного фрагмента кода нельзя провести рефакторинг «Выделение функции», т.к. в языке Си нельзя вставить вызов функции перед декларацией переменной «*c*».

```
int main()
{
    int a = 1;
    int b = 2;
    int c = a++ + b;

    return a + b + c;
}
```

Кроме этого, необходимо учитывать количество выделенных деклараций, т.к. это влияет на вид новой функции. Например, если выделена только одна декларация переменной, то при проведении рефакторинга «Выделение функции» достаточно заменить инициализатор декларации на вызов новой функции. При выделении фрагмента кода, в котором присутствуют две и более декларации, логичнее создавать функцию для инициализации переменных, не привязанную к какой-либо из деклараций.

5.4. Узлы выхода

Для языков Си/Си++ узлами выхода являются операторы «*return*», «*continue*» и «*break*», причем можно заметить, что два последних оператора стоит учитывать только, если выделенный фрагмент находится внутри цикла. В зависимости от количества узлов «выхода», их вида и места расположения должны применяться различные стратегии для проведения рефакторинга «Выделение функции». Рассмотрим пример:

```
int main(int argc, char *argv[])
{
    int i;
    if (argc < 2)
        return -1;
    i = argc;
    …
    return 0;
}
```

Очевидно, что этот выделенный фрагмент нельзя заменить только вызовом новой функции. Для того чтобы сохранить поведение программы необходимо провести замену выделенного фрагмента так, чтобы сохранилось условие вызова и возвращаемое значение оператора «*return*». В приведенном примере можно заменить выделенный фрагмент кода условным оператором с вызовом новой функции в качестве условия вызова оператора «*return*»:

```
int main(int argc, char *argv[])
{
    int i;
    if (extracted_method(argc, &i))
        return -1;
    …
    return 0;
}
```

Выделенный фрагмент кода может быть сложнее, например, могут присутствовать несколько узлов выхода с разными возвращаемыми значениями. В таком случае можно завести новую переменную, через которую функция будет возвращать значение, например:

```
int foo(char *ptr)
{
    int ret;
    ret = bar(ptr);
    if (ret == -1)
        return 128;
    if (ret == -2)
        return 129;
    ...
}
```

В этом примере рефакторинг «Выделение функции» можно провести следующим образом:

```
int extracted_function(int *ret, char *p, int *o)
{
    *ret = bar(p);
    if (*ret == -1) {
        *o = 128;
        return 1;
    }
    if (*ret == -2) {
        *o = 129;
        return 1;
    }
    return 0;
}
int foo(char *ptr)
{
    int ret;
```

```
    int o;
    if (extracted_func(&ret, ptr, &o))
        return o;
    ...
}
```

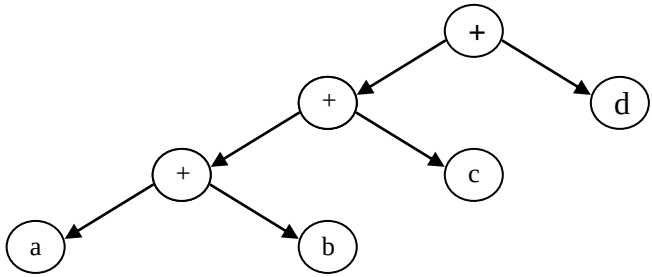
Если в выделенном фрагменте кода одновременно присутствуют различные узлы выхода, то предпочтительнее не проводить рефакторинг. Т.к. результат, полученный при применении рефакторинга, может оказаться более запутанным, нежели исходный фрагмент кода, что, очевидно, противоречит цели рефакторинга.

5.5. Проверка выделенного выражения

Если выделенный фрагмент кода является частью другого выражения, например, «*d - a + b*», то необходимо проверить, что проведение рефакторинга «Выделение функции» не нарушит приоритет операций, а, следовательно, не изменит поведение программы.

Рассмотрим пример: «*a = b && c == d + e*». Очевидно, что после проведения рефакторинга «Выделение функции», а именно вынесения выражения «*b && c*» в новую функцию, будет нарушен приоритет операций, и, как следствие, будет изменено поведение программы.

Частично эту проблему решает синтаксический анализатор, т.к. для выделенного фрагмента не будет существовать узла в синтаксическом дереве. Можно сказать, что существование соответствующего узла в синтаксическом дереве является достаточным условием для подтверждения корректности выделенного фрагмента. Однако, поиск необходимого условия осложняется тем, что синтаксическое дерево может не содержать соответствующего узла для корректно выделенного фрагмента. Например, рассмотрим выражение «*a + b + c + d*», для которого будет построено следующее синтаксическое дерево:



В этом дереве нет узла, соответствующего выражению «*b + c*», но для проведения рефакторинга «Выделение функции» это выражение корректно.

Более того, можно указать похожее синтаксическое дерево, где выделение выражения «*b + c*» уже не будет корректно: «*a * b + c + d*».

Поиск необходимого условия для подтверждения корректности выделенного фрагмента кода описывается в виде алгоритма:

1. найти наименьшую вершину синтаксического дерева, являющуюся родителем для всех узлов из выделенного фрагмента кода;
2. пройти путь снизу-вверх от самого левого и правого узлов выделенного фрагмента до найденной вершины;
3. если при прохождении пути приоритет операций не понижается, значит выделение корректно.

Стоит отметить, что для большего удобства при проведении рефакторинга «Выделение функции» существуют инструменты, которые помогают бороться с некорректным выделением, подсказывая пользователю начало и конец выделенного выражения [16]. Однако, если для выделенного выражения не существует соответствующего узла в синтаксическом дереве, как было показано выше, то эти инструменты ведут себя некорректно.

6. Модуль трансформации синтаксического дерева

Рассмотрим, как можно распечатать синтаксическое дерево в виде исходного кода:

- обходом по дереву (например, вглубь) и распечаткой заранее заготовленных текстовых шаблонов для каждого конкретного узла
- проходом по токен-последовательности верхнего узла дерева

Эти два способа сильно разнятся между собой. Первый из них базируется на структуре дерева, но не имеет информации об его текстовом представлении. Второй базируется на точном текстовом представлении дерева, но не имеет информации об его структуре. Рассмотрим пример:

```
#define START_NUM 0x5
for (i = START_NUM /* start num of … */; i < len;
i++) {
    a = str[i]; // str is …
    if (a) // fatal
        exit(0);
}
```

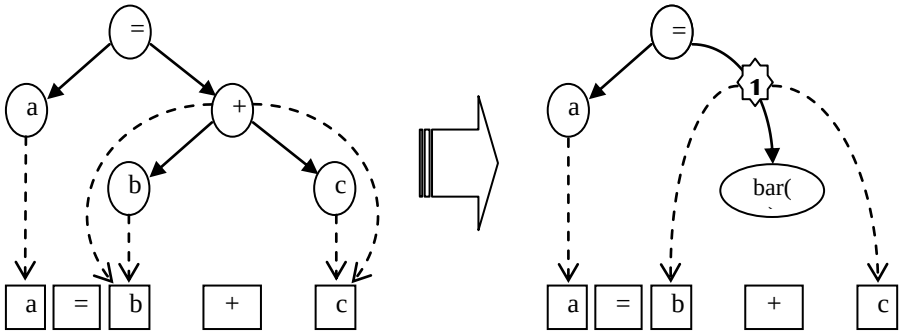
В результате распечатки синтаксического дерева первым способом произойдет потеря ряда важной информации, и результат будет примерно следующим: «*for(i=0x5;i<len;i++){a=str[i];if(a)exit(0);}*». Очевидно, что такой стиль исходного кода является неприемлемым. С другой стороны, если в результате рефакторинга меняется структура синтаксического дерева, то

распечатка синтаксического дерева проходом по токен-последовательности верхнего узла не будет отражать эти изменения.

Для решения задачи распечатки синтаксического дерева с максимально возможным сохранением пользовательской индентации был создан механизм, который отображает изменения в структуре синтаксического дерева на уровню его токен-последовательности. Этот механизм реализован в модуле трансформации синтаксического дерева. Он предоставляет простой интерфейс для работы с деревом (удалить/заменить/добавить узел) и позволяет сохранять по возможности оригинальный стиль исходного кода. Все изменения, которые вносятся в синтаксическое дерево, сохраняются и учитываются при распечатке узла обходом по токен-последовательностей.

6.1. Принцип работы с деревом

Синтаксическое дерево обладает единой связной токен-последовательностью, и каждый узел дерева имеет ссылки на свои начальный и конечный токены. Кроме того, каждый узел обладает точно определенными ребрами, которые описаны в спецификации синтаксического дерева. Можно заметить, что изменение узла в синтаксическом дереве - это изменение поддерева, находящегося на неком ребре, поэтому, чтобы сохранить информацию об изменении в синтаксическом дереве, ребро, ведущее к измененному узлу, помечается специальной меткой. В этой метке указывается начальный и конечный токены старого узла. Сам узел заменяется на новый. Этого достаточно, чтобы правильно определить позицию для внесения изменений в исходный код и найти токены, которые соответствуют измененным узлам синтаксического дерева. Рассмотрим пример, где в выражении «*a = b + c*» происходит замена «*b + c*» на вызов функции «*bar()*».



В этом примере ребро, идущее к узлу «*b + c*», помечается меткой «*1*». Для этой метки сохраняются токены «*b*» и «*c*», как начальный и конечный токены старого узла.

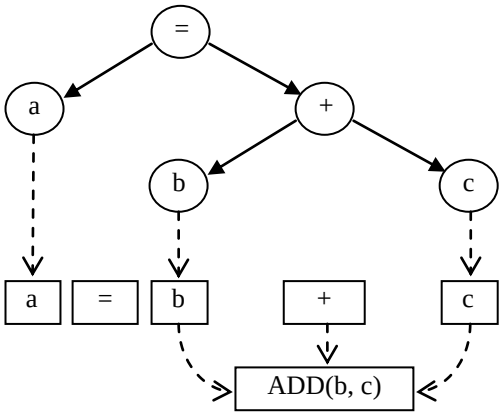
Можно отметить ряд положительных моментов в описанном подходе. Например, синтаксическое дерево изменяет свою структуру согласно трансформациям, которые проводит пользователь. Это дает возможность для

внесения изменений в уже измененные части дерева и дальнейшего структурного анализа. Сохраняется целостность токен-последовательности синтаксического дерева и, как следствие, сохраняется максимальное количество пользовательской индентации. Более того, узлы, которые были заменены другими узлами, не меняют свою токен-подпоследовательность.

Отдельно стоит упомянуть, что при распечатке синтаксического дерева в виде исходного кода важно сохранять макро-вызовы. Для того, чтобы информация о макросах присутствовала в синтаксическом дереве, существует дополнительное отображение из токенов, представляющих раскрытый макрос, в макро-вызов. Это дает возможность определить, какие токены были частью раскрытого макроса. Рассмотрим пример:

```
#define ADD(x, y) x + y
a = ADD(b, c);
```

Для этого примера будет построено следующее синтаксическое дерево:



В этом примере токены, находящиеся внутри раскрытого макроса, имеют ссылку на текстовое представление макро-вызова.

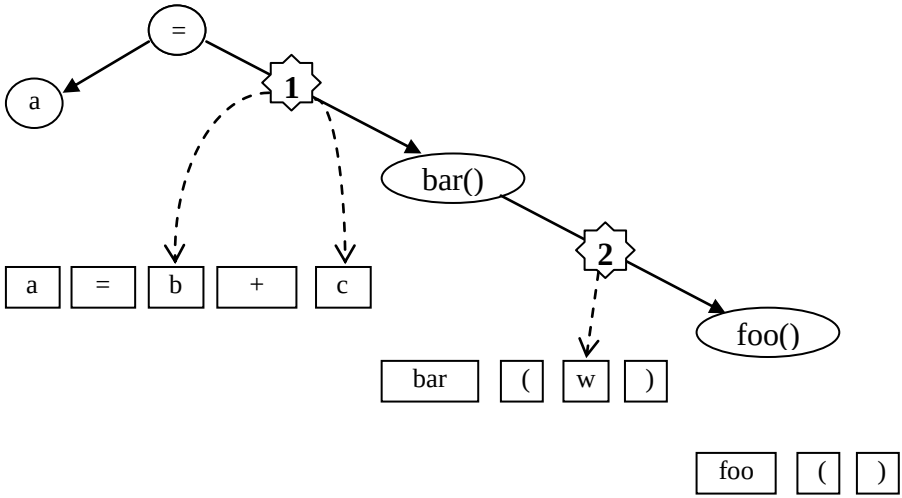
Стоит заметить, что в рассматриваемом рефакторинге «Выделение функции» выделенный фрагмент либо содержит макро-вызов целиком, либо не содержит вовсе. Более того, нет необходимости рассматривать случаи, когда макро-вызов задает неполную синтаксическую конструкцию, т.к. при выделении кода выполняется проверка на корректность выделенного фрагмента, которая, в частности, включает проверку на синтаксическую целостность.

6.2. Создание набора текстовых изменений для исходного кода

Результатом работы РМ являются изменения в исходном коде. Для этого на последней стадии РМ изменения в синтаксическом дереве отображаются в изменения исходного кода. Найти измененные узлы синтаксического дерева

является простой задачей, так как ведущие к ним ребра имеют специальную метку. Чтобы определить место, где в исходном коде начинается и заканчивается измененный узел, достаточно посмотреть на начальный и конечный токены, сохраненные для метки, т. к. каждый токен хранит свою позицию в тексте. Далее происходит распечатка измененного узла. Распечатывается узел проходом по токен-последовательности, причем токены, соответствующие измененным частям узла, пропускаются, а вместо них распечатывается новый узел (это узел, которым заменили старый узел в процессе трансформации синтаксического дерева). Процесс распечатки узла естественным образом реализуется с применением рекурсии, т.к. каждый новый узел распечатывается по тому же принципу, как и его прародитель. Более того, глубина изменений в узле может быть произвольной.

Рассмотрим пример, где в выражении «*a = b + c*» сначала «*b + c*» заменяется на вызов функции «*bar(w)*», а потом переменная «*w*» заменяется на вызов функции «*foo()*». В результате будет построено следующее синтаксическое дерево:



При замене выражения «*b + c*» на вызов функции «*bar(w)*» ребро, ведущее к «*b + c*» помечается меткой «*1*». Для метки сохраняются начальный и конечный токены выражения, а именно, токены «*b*» и «*c*». При замене переменной «*w*» на вызов функции «*foo()*» ребро, ведущее к «*w*», помечается меткой «*2*». Для нее сохраняются начальный и конечный токены узла, которые в данном случае совпадают, т.к. переменной соответствует только один токен - «*w*». Рассмотрим, как изменения на уровне синтаксического дерева отобразятся в изменения на уровне исходного кода. Обходом верхнего узла «в глубину» ищутся измененные ребра, причем обходчик не заходит внутрь таких ребер. Результатом поиска будет ребро, помеченное меткой «*1*». Используя сохраненные токены метки «*1*», можно определить позицию в

исходном коде, куда нужно вставить измененную часть дерева. После этого происходит распечатка измененной части дерева, а именно узла, к которому ведет ребро, помеченное меткой «*1*». Для текстовой распечатки используется токен-последовательность узла, причем для рассматриваемого узла такой последовательностью будет: «*bar*», «*(*«, «*w*», «*)*». Чтобы определить, какие токены в токен-последовательности соответствуют измененным частям узла, опять проводится обход в глубину с поиском измененных ребер. Таким ребром является ребро, помеченное меткой «*2*». Оно ссылается на токен «*w*», который соответствует измененному узлу. Т.к. все токены, соответствующие измененным узлам, найдены, то начинается распечатка токен-последовательности узла. Распечатываются токены «*bar*» и «*(*». Далее, т.к. токен «*w*» соответствует измененному узлу, распечатывается узел «*foo()*». После этого распечатывается последний токен «*)*». Таким образом, выражение «*a + b*» в исходном коде заменяется на выражение «*bar(foo())*».

7. Обработка ошибок компиляции

В процессе разработки программисту может понадобиться провести рефакторинг неполного, а, следовательно, синтаксически некорректного кода. Это, очевидно, дает ему определенную свободу, т.к. позволяет улучшать структуру программы еще в самом начале процесса ее написания. В этот момент программа может содержать незаконченные синтаксические конструкции. Однако, для успешного проведения рефакторинга необходимо, чтобы программа была не только синтаксически, но и семантически корректна. С другой стороны, если рефакторинг проводится над одной оконченной частью программы, то неверно в процессе рефакторинга рапортовать об ошибке, ссылаясь на незаконченность или некорректность другой части. Например, странно отказываться проводить рефакторинг «Выделение функции» внутри некой выбранной функции, если другая независимая функция программы еще не дописана. Чтобы определить, можно ли провести рефакторинг на некорректном синтаксическом дереве, необходимо проанализировать некую локальную область выделенного фрагмента кода. Т.е. проверить на синтаксическую и семантическую корректность только те узлы дерева, которые подвергнутся изменениям в процессе рефакторинга. Таким образом, подобный подход, с одной стороны, подскажет пользователю, что выделяемый для рефакторинга фрагмент кода еще не готов и должен быть дописан, с другой стороны, проведет рефакторинг без учета неоконченности программы.

8. Сравнительное тестирование существующих решений

Для объективного сравнения существующих решений, которые реализуют рефакторинг «Выделение функции», были опрошены 5 программистов.

Каждому из них было дано задание выбрать около 7 тестовых примеров различной сложности, содержащих разнообразные языковые конструкции. Одно из основных условий - это выбор примеров из реальных проектов, а не создание синтетических наборов для тестирования.

	количество пройденных тестов	*процент пройденных тестов*
Visual Assist X	4	11 %
Eclipse CDT	11	31 %
XCode	15	42 %
CodeRush	26	74 %
Klocwork Insight	31	87 %

Табл. 1.

Полученный в результате опроса набор примеров для тестирования существующих решений охватывает большое количество различных конструкций языка программирования. Например, в него входят тесты с проверками на конфликты имен, обработку типов, использование пространств имен, using-директив, сложные возвращаемые значения, обработку деклараций и узлов возврата. Каждое из существующих решений было протестировано с помощью примеров из тестового набора. Результат тестирования представлен в таблице 1.

Подробные результаты тестирования:

Visual Assist X

Правильно отрабатывает лишь на простейших примерах. Удивило отсутствие проверки выделяемого фрагмента (например, можно выделить часть комментария в новую функцию). Отсутствие правильной работы с типами, узлами возврата, декларациями переменных, анализа входных/выходных параметров постоянно приводит к неверным результатам рефакторинга.

Eclipse CDT

Инструмент правильно отрабатывает только на простейших примерах. Иногда сообщает о невозможности провести рефакторинг с указанием причины. Также присутствует проверка выделенного фрагмента, что уменьшает риск проведения рефакторинга на неправильно выделенном коде. Но поддержка ограниченного количества языковых конструкций и некорректная обработка типов и входных/выходных параметров приводит, либо к отказу проводить рефакторинг, либо к неправильным результатам.

XCode

Стоит отметить, что инструмент не поддерживает рефакторинг Си++ кода. Правильный рефакторинг возможен только на простейших примерах. Неправильно обрабатываются сложные типы данных, узлы выхода. Нет возможности выделить часть выражения, в результате в новую функцию выносится весь оператор. Не сохраняется пользовательская индентация.

CodeRush

Инструмент показал высокий уровень анализа кода. Из недостатков можно отметить недостаточный анализ связей между выделенным фрагментов и окружающими конструкциями (например, локальные объявления типов), использование конструкций языка Си++ в исходном коде на языке Си, обработку сложных возвращаемых конструкций (например, указатель на массив), ошибки при анализе макро вызовов.

Klocwork Insight

К недостаткам можно отнести отсутствие обработки частей кода, находящихся под условной компиляцией, не раскрытой в текущей конфигурации, избыточное вынесение переменных в параметры при использовании оператора «взять адрес» от этих переменных (при условии, что они используются только внутри выделенного фрагмента), отказ от рефакторинга, если необходимо изменить способ доступа к переменной, которая является аргументом макроса.

9. Перспективы развития

Рассматриваемый подход для проведения рефакторинга имеет несколько направлений для развития. Первое направление - это реализация новых видов рефакторинга. Для добавления нового рефакторинга достаточно описать правила преобразования узлов синтаксического дерева, а работу по конструированию изменений на уровне исходного кода выполнит модуль трансформации синтаксического дерева. Но стоит отметить, что процесс описания правил преобразования узлов синтаксического дерева достаточно трудоемок. Несмотря на наличие синтаксической и семантической информации о программе в виде синтаксического дерева, способ работы с

деревом нельзя назвать удобным. Например, трудоемкость написания функций, осуществляющих поиск по критерию в дереве, достаточно высока, кроме того, разнообразие типов, атрибутов и структур приводит к написанию сложных конструкций для реализации простых задач. Отсюда вытекает второе направление для развития - разработка «удобного» средства поиска и доступа к узлам и атрибутам синтаксического дерева. В качестве такого средства могут быть использованы методы работы с синтаксическим деревом, описанные в статье «Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST» [17]. С их помощью, используя модуль трансформации синтаксического дерева, можно автоматически проводить почти любую трансформацию кода, например, переносить программу на новую архитектуру, или проводить сложные изменения в программе, выходящие за рамки широко используемых рефакторингов.

10. Заключение

Рефакторинг исходного кода является одной из самых распространенных и успешных техник для улучшения дизайна программы. Но, не смотря на то, что каждый из рефакторингов описывается довольно кратким и понятным образом, реализация инструмента для автоматического проведения рефакторинга является довольно сложной задачей. С другой стороны, даже несмотря на недостатки, существующие инструменты для проведения автоматического рефакторинга активно используются среди программистов, что подчеркивает актуальность выбранной темы. Приведенный подход для проведения рефакторинга «Выделение функции» содержит принципы и методы, с помощью которых результат рефакторинга обладает не только высокой точностью, но и сохраняется стиль исходного кода. Стоит отметить, что описанный подход для проведения рефакторинга был применен при реализации нескольких рефакторингов. В частности, были реализованы рефакторинги «Встраивание функции», «Введение переменной», «Встраивание переменной», которые также вошли в состав инструмента Klocwork Insight.

Список литературы

[1] В.О. Савицкий, Д.В. Сидоров. *Инкрементальный анализ исходного кода на языках C/C++*. Труды Института системного программирования РАН, том 22, 2012 г.
[2] Мартин Фаулер. *Рефакторинг. Улучшение существующего кода.*
[3] Martin Fowler. *Refactoring Home Page.* http://www.refactoring.com/
[4] http://msdn.microsoft.com/en-us/library/0s21cwxk.aspx
[5] http://www.refactoring.com/catalog/extractMethod.html
[6] http://www.devexpress.com/Subscriptions/DXperience/DXv2/index.xml
[7] http://www.wholetomato.com/products/featureRefactoring.asp
[8] http://www.eclipse.org/cdt
[9] Michael Ruegg. *Eclipse CDT refactoring overview and internals.*
[10] https://developer.apple.com/xcode

[11] Max Schaefer, Oege de Moor. *Specifying and implementing refactorings.*
[12] Zhiying (Vicky) Wang. *A Survey of Refactoring Tool Researches.*
[13] Peter Sommerlad. *Retaining comments when refactoring code.*
[14] Jeffrey Overbey, Ralph Johnson. *Generating Rewritable Abstract Syntax Trees.*
[15] http://www.opennet.ru/docs/RUS/diff/diff-3.html
[16] Emerson Murphy-Hill, Andrew Black. *Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method.*
[17] С.В. Сыромятников. *Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST.* Труды Института системного программирования РАН, том 20, 2011 г.

The refactoring approach used in Klocwork Insight toolkit

N. L. Lugovskoy
lugovskoy@ispras.ru
ISP RAS, Moscow, Russia

Abstract. The paper describes refactoring technics used in Klocwork Insight toolkit for C/C++ programming languages. Being the most popular Extract Function refactoring is chosen to describe all stages of refactoring process. Different language elements and constructions are processed during extraction of a new function and most of them are included as examples. Additionally it's shown that extract function refactoring has to use data-flow analysis to resolve conflicts or different ways of data usage on different conditional branches. Furthermore refactoring has to change and move declarations of variables. So it has to resolve type conflicts and implement declaration order of used types in case type is declared inside function. There are also some points about syntax tree transformations. Such transformations can be automatically mapped to source code changes. Automatically modifying the program's source code, based on the changes in the syntax tree is commonly known as code transformation, and can be used not only for refactoring purposes. The technic of code transformation used in our approach can be used as a key idea for a new tool for doing non-trivial transformations of source code.

Keywords: refactoring; extract function; code transformation; static analysis

References

[1]. [1] V. O. Savitsky, D. V. Sidorov, Inkremental'nyj analiz iskhodnogo koda na yazykakh C/C++[Incremental source code analysis for C/C++ languages] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol 22, pp. 119-129 (in Russian)
[2]. [2] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison-Wesley, 2000
[3]. [3] Martin Fowler. Refactoring Home Page. http://www.refactoring.com/
[4]. [4] http://msdn.microsoft.com/en-us/library/0s21cwxk.aspx
[5]. [5] http://www.refactoring.com/catalog/extractMethod.html
[6]. [6] http://www.devexpress.com/Subscriptions/DXperience/DXv2/index.xml
[7]. [7] http://www.wholetomato.com/products/featureRefactoring.asp
[8]. [8] http://www.eclipse.org/cdt
[9]. [9] Michael Ruegg. Eclipse CDT refactoring overview and internals. Parallel Tools Platform (PTP) Workshop, Chicago, September, 2012
[10]. [10] https://developer.apple.com/xcode

«Ленивый» анализ исходного кода на языках C и C++

Савицкий В.О., Сидоров Д.В.
ssavitsky@ispras.ru, sidorov@ispras.ru

Аннотация. В статье описывается метод построения синтаксического анализатора, позволяющий существенно сократить требуемые для анализа ресурсы. Метод основан на том факте, что каждый исходный файл подключает множество заголовков, из которых используется лишь небольшое количество определений. Разбор определений из заголовков можно пропускать до момента непосредственного обращения к ним, таким образом, неиспользуемые определения анализироваться не будут. Отличительной особенностью метода является необходимость внесения лишь небольшого количества изменений в существующий парсер. Метод реализован в статическом анализаторе Klocwork Insight.

Ключевые слова: ленивый анализ, синтаксический анализ, C/C++.

1. Введение

По мере развития любого продукта, объем программного кода растет. Зачастую скорость прироста строк кода в проекте не линейна по времени. Такой рост, например, показывает ядро системы Linux [1]. Как следствие, растет время компиляции проектов. Кроме того, при работе с большими проектами программисты часто используют инструменты анализа и трансформации кода. Эти инструменты включают в себя рефакторинг кода, статический анализ, построение графа зависимостей проекта и интеллектуальное автодополнение. Такие задачи обычно выполняются по дереву абстрактного синтаксиса, которое получается в результате работы синтаксического анализатора (парсера) [2]. Как правило, время работы парсера сравнимо со временем последующего анализа, поэтому разработчики компиляторов прилагают немало усилий для минимизации требуемого количества ресурсов.

Проблема сокращения времени анализа наиболее актуальна для языков C и C++, так как они не обладают свойством инкапсуляции времени компиляции. Например, для языка C++ это означает, что изменение защищенного члена класса потребует повторной компиляции всего кода, в котором этот класс используется. Одним из вариантов сократить время компиляции кода является инкрементальный анализ [3] - кеширование используемых в файле заголовков

для ускорения повторного анализа. Однако, этот метод работает только при многократном анализе одного и того же файла, и требует работы в интегрированной среде разработки. Рассмотрим метод синтаксического анализа, который позволит ускорить первый разбор файла, а также может использоваться при полной сборке проекта.

2. Обзор существующих решений

Обычно структура проектов, написанных на языках C и C++, такова: в начале каждого исходного файла записаны директивы препроцессора для подключения заголовочных файлов, затем следуют определения, относящиеся непосредственно к этому файлу. Ниже показан пример такого файла.

```
#include <windows.h>
#include <iostream>
#include <user_types.h>
typedef unsigned int boxes;
```

В подключаемые файлы выносятся декларации и определения, которые могут использоваться во многих файлах. Каждый файл может использовать лишь небольшое количество определений из заголовочного файла, но вынужден подключать его целиком. Кроме того, одни и те же заголовки анализируются много раз. В результате время компиляции файла и всего проекта неоправданно увеличивается. Одним из возможных решений проблемы является исключение из компиляции неиспользуемых объектов. Такое решение частично реализовано в компиляторе MS Visual Studio [4].

В подключаемых заголовках содержится много шаблонов, многие их которых никогда не используются. Когда парсер обнаруживает заголовок какого-либо шаблона, он пропускает его тело, сохраняя контекст, необходимый для последующего разбора. Если затем парсер обнаружит инстанциацию (инициализацию типом) шаблона, то будет выполнен анализ его тела. Если же такой инстанциации не обнаружится, то тело шаблона проанализировано не будет. В результате такой код не вызовет ошибки компилятора:

```
template <typename T> class error_example {
  >>>> example error  <<<<
};
```

Такое поведение парсера можно назвать "ленивым" - анализ тела шаблона откладывается настолько, насколько возможно, или не выполняется совсем.

Другим примером «ленивого» поведения являются бесконечные списки в языке Haskell. Значение элемента такого списка вычисляется только при

непосредственном обращении к нему, если же элемент не требуется, то и значение его вычислено не будет.

«Ленивый» метод анализа можно распространить и на другие существенные части кода языков C и C++ – тела функций, структур и классов, не обязательно шаблонных.

3. «Ленивый» анализ.

Для оценки количества объектов, которые реально используются в файлах проекта, мы проанализировали несколько проектов с открытым исходным кодом. Множество используемых функций и классов определяется так. Просмотрим все объекты в исходном коде следующих типов. Если это класс, то просматриваем его члены и функции-члены. Если это функция, то просматриваем ее тело. Если встречается класс или функция, которых нет в множестве просмотренных, то добавляем их. Таким образом мы получим множество функций и классов, которые нужны для транслирования данного исходного файла. Остальные функции и классы можно пропустить при анализе. На рис. 1 представлено общее количество классов и функций на проектах в сравнении с количеством используемых.

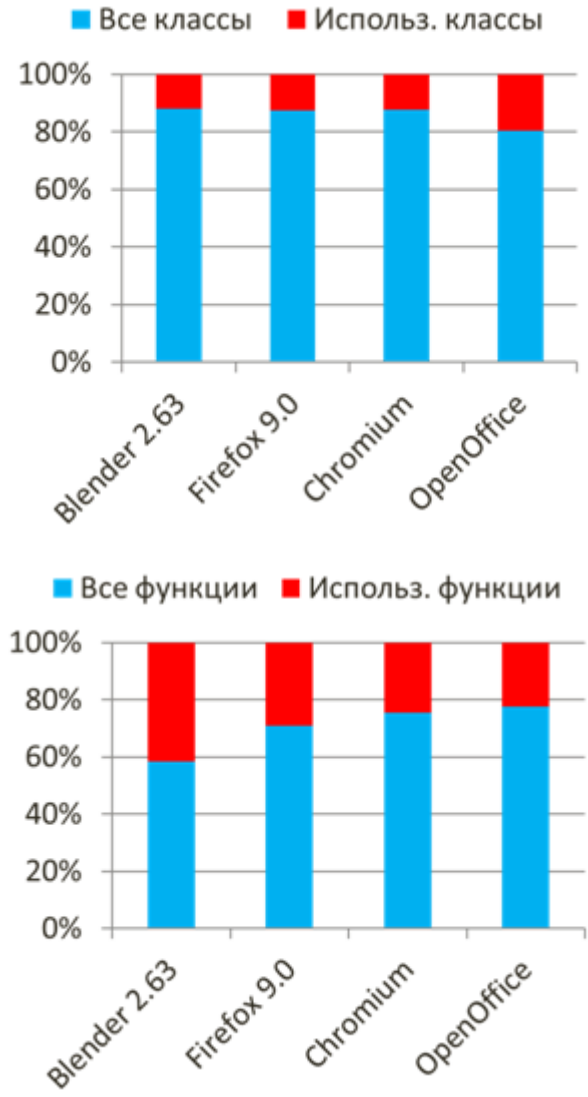


Рис. 1. Соотношение использованных и всех объектов

Получается, что при анализе этих проектов можно пропускать более половины строк, относящихся к классам, и почти 80% строк функций. Рассмотрим подробнее, как будет работать такой анализатор.

4. Описание алгоритма.

Схема работы компилятора выглядит так. Для языков C\C++ вначале исходный текст обрабатывается препроцессором, который заменяет макросы и раскрывает директивы подключения заголовочных файлов. Затем на стадии лексического анализа полученный текст разбивается на распознаваемые единицы текста - лексемы, которые передаются синтаксическому анализу (парсеру). Синтаксический анализ определяет, может ли такая последовательность лексем соответствовать грамматике анализируемого языка, и строит дерево разбора, отражающее синтаксическую структуру исходной программы. После синтаксического анализа или одновременно с ним выполняется семантический анализ. Семантический анализ осуществляет проверку типов, вычисляет значения выражений времени компиляции, разрешает вызовы функций и снабжает узлы дерева разбора атрибутами – семантическими элементами.

Как правило, парсер для языков C/C++ строится по алгоритму восходящего синтаксического разбора LALR(1). Такой парсер просматривает последовательность токенов слева направо, принимая решение о сдвиге или свертке на основании предпросмотра одного токена. При сдвиге, текущий токен или нетерминал переносится на стек; при свертке символы, соответствующие правилу свертки, со стека снимаются. Грамматики языков C и C++ в особенности не являются однозначными, и зачастую выбор продукции для свертки зависит от типа идентификатора. Поэтому при описании грамматики языка нередки конфликты вида сдвиг\свертка и свертка\свертка. Такие конфликты разрешаются по методу обобщенного LR(1) анализа - стек дублируется для каждого из возможных вариантов, и анализ проводится для каждого из стеков отдельно до тех пор, пока на одном из них не будет обнаружена ошибка синтаксиса, или оба стека не придут в одинаковое состояние.

Для добавления функциональности отложенного разбора тел функций и классов потребуется не большое количество изменений. Когда в процессе просмотра списка лексем парсер встречает символ открывающей фигурной скобки, он должен пропустить все лексемы вплоть до символа закрывающей фигурной скобки, с сохранением баланса скобок. Вместо узла дерева, соответствующего телу класса, парсер вставит специальный узел, содержащий необходимую информацию, чтобы обеспечить разбор тела, если он понадобится в дальнейшем. Эта информация будет включать в себя указатели на первый и последний символы тела класса в последовательности лексем, а также указатель на текущую область видимости. В дальнейшем, если будет необходимо создать объект класса или обратиться к объекту внутри области видимости класса, парсер установит сохраненную область видимости в качестве текущей и вызовет действие для анализа тела класса. Как правило, этой действие является процедурой того же парсера, поэтому потребуются изменения для обеспечения реентерабельности процедуры.

Следующий фрагмент демонстрирует отложенный анализ тела функции.

```
int foo(int n) {
  if (n > 0)
    return foo(n – 1);
  return 0;
}
…
int main() { // тело функции main не пропускается
  return foo(3);  // анализ тела функции foo
}
```

В этом примере анализ тела функции **foo** будет отложен до момента обнаружения ее вызова в теле функции **main**. В качестве текущей лексемы в парсере будет установлена открывающая фигурная скобка в первой строке, а в качестве текущей области видимости будет установлена область видимости функции **foo**. В ходе анализа необходимо отслеживать рекурсивный вызов текущей функции и обращение к области видимости текущего класса, чтобы не войти в бесконечный цикл при попытке запустить анализ по той же сохраненной информации. Для этого можно помечать узел, из которого берется контекст для анализа, как «используемый», и в дальнейшем не запускать анализ при обнаружении таких узлов.

При анализе тела класса можно в свою очередь откладывать разбор тел функций-членов и вложенных классов. То же верно и для тел обычных функций.

Естественно, что синтаксическое дерево, построенное при «ленивом» разборе включает не все узлы из дерева, которое будет получено при полном анализе. На момент завершения анализа в таком дереве будут содержаться специальные узлы с сохраненной информацией для тел классов и функций, к которым не было обращений. Это необходимо учитывать при обходе дерева, к примеру, с целью поиска дефектов. Однако, отсутствие дефектов для неиспользуемого кода вполне оправданно, ведь такие дефекты не будут влиять на качество программы. Кроме того они будут обнаружены после изменений, приводящих к использованию пропущенной части кода.

При сохранении контекста для отложенного разбора тела класса - указателей на лексемы и области видимости, необходимо запоминать информацию о моменте, в который была сохранена область видимости. Это нужно для того, чтобы избежать ошибок в ситуациях, когда в область видимости добавляются имена, которые в ней уже есть, но с другими значениями. Так, в следующем примере на момент объявления функции **foo** в текущей области видимости имя **x** обозначает переменную типа int. Если мы отложим разбор тела функции до момента вызова, то при восстановлении области видимости как текущей,

имя x в ней будет обозначать тип. Таким образом вместе со ссылкой на область видимости необходимо сохранять ее состояние. К примеру, можно ввести понятие **ревизии** наподобие систем контроля версий. При добавлении имени в область видимости, ее ревизия будет увеличиваться на единицу, а добавленное имя будет хранить номер ревизии. Затем значение ревизии можно будет использовать при поиске имени в области видимости - в данном примере тело функции было сохранено с областью видимости с ревизией 2, значит, все имена с номером ревизии больше 2 были добавлены позже, и не могут учитываться при поиске имен в процессе анализа тела функции.

```
template <int wheels> class Vehicle {};
-----------------------------------------------
int x;          // x: rev. 1
void foo() {    // foo: rev. 2
 x = 2;
}
typedef int x; // x: rev. 3
foo();
```

В этом примере анализ тела функции будет происходить только, когда парсер определит вызов функции **foo**. Для этого в качестве текущей будет установлена глобальная область видимости с номером ревизии 2 - то есть на момент декларации функции **foo**. В процессе определения, к какой из двух деклараций на верхнем уровне относится имя x в теле функции, будут использованы ревизии. Декларация синонима типа не попадает в список кандидатов: это имя было добавлено в область видимости с номером ревизии 3, то есть уже после объявления тела функции.

Так как шаблон **Vehicle** не используется, то его тело проанализировано не будет. Ленивый анализ имеет смысл проводить только для элементов в заголовках, ведь исходные файлы обычно используют все свои декларации, кроме того их размер не велик в сравнении с размерами заголовков.

5. Результаты.

В таблице 1 представлено суммарное время синтаксического и семантического анализа для всех файлов проектов с использованием ленивого режима и без.

проект	строк кода, тысяч	обычный режим, сек	«ленивый» режим, сек	увеличение скорости анализа
Blender 2.63	1,029	168	132	22%
Firefox 9.0	2,198	1,144	735	36%
Chromium	3,099	13,069	8,816	33%
OpenOffice 3	6,986	27,706	21,693	22%

Таб. 1.

Видно, что данные результаты не показывают такого сокращения времени, как ожидалось по оценкам количества строк кода. Это связано с тем, что зависимость времени анализа от количества строк кода не линейна. Так, например, при декларации переменной, создается всего один семантический элемент, который добавляется в текущую область видимости. При декларации массива, таких элементов будет создано несколько, а при описании шаблона кроме семантических элементов необходимо сохранять и структуру шаблона.

Однако, полученные результаты – ускорение от 22% до 36% - представляют существенное улучшение времени синтаксического анализа.

6. Заключение.

Описанный метод позволяет получить выигрыш более 20% времени синтаксического анализа проектов. Данный метод реализован в инструменте статического анализа Klocwork Insight. Еще одним способом оптимизации времени выполнения анализа может стать совмещение инкрементального и ленивого анализа при работе в средах разработки.

Список литературы.

[1] http://www.easterbrook.ca/steve/?p=694

[2] А. Аветисян, А. Белеванцев, А. Бородин, В. Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН, том 21, с.23-38, 2011

[3] Савицкий В.О., Сидоров Д.В., Инкрементальный анализ исходного кода на языках C/C++. Труды ИСП РАН, том 22, с. 119-129, 2012
[4] http://msdn.microsoft.com/en-us/library/x5w1yety%28v=vs.71%29.aspx
[5] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986

Lazy source code analysis for C/C++ languages.

V. O. Savitsky, D. V. Sidorov
ssavitsky@ispras.ru, sidorov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. The article describes a way to implement syntax analysis for C/C++ languages which allows to reduce significantly the required resources. The method is based on the fact that each source file includes lots of header files from which only a portion of definitions is used. The analysis of definitions in headers can be postponed to the moment of actual access to them, thus avoiding the analysis of definitions that are never used. Such definitions as templates, classes, functions could be skipped during code parsing until access of such definitions will be found in source code. For these purposes parser of C/C++ code modified to support postponed parsing of skipped parts along with revision system like caching of semantic information for parsed code. As the result of implementation of such parser performance growth of source code parsing up to 36% observed on several open source projects. The distinctive feature of this method is that only a small amount of changes is required to add the "lazy" behaviour to an existing parser. This method of source code parser performance improvement is implemented in the Klocwork Insight static analyzer.

Keywords. lazy analysis, syntax analysis, C/C++

References

[1]. http://www.easterbrook.ca/steve/?p=694
[2]. A. Avetisyan, A. Belevantsev, A. Borodin, V. Nesov Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode programm[Using static analysis for finding security vulnerabilities and critical errors in source code] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol 21, pp. 23-38 (in Russian)
[3]. V. O. Savitsky, D. V. Sidorov, Inkremental'nyj analiz iskhodnogo koda na yazykakh C/C++[Incremental source code analysis for C/C++ languages] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol 22, pp. 119-129 (in Russian)
[4]. http://msdn.microsoft.com/en-us/library/x5w1yety%28v=vs.71%29.aspx
[5]. Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986

Большие данные: современные подходы к хранению и обработке

П.А. Клеменков, С.Д. Кузнецов
parser@cs.msu.su, kuzloc@ispras.ru

Аннотация. Большие данные поставили перед традиционными системами хранения и обработки новые сложные задачи. В данной статье анализируются возможные способы их решения, ограничения, которые не позволяют сделать это эффективно, а также приводится обзор трех современных подходов к работе с большими данными: NoSQL, MapReduce и обработка потоков событий в реальном времени.

Ключевые слова: большие данные; реляционная модель; nosql; mapreduce; hadoop; s4; storm

1. Введение

Мы живем в информационный век. Нелегко измерить общий объем электронных данных, но по оценкам IDC размер «цифровой вселенной» в 2006 г. составлял 0.18 зеттабайт, а к 2011 г. должен был достигнуть 1.8 зеттабайт, продемонстрировав десятикратный рост за 5 лет! Вот только несколько примеров источников таких объемов [1]:

1. Нью-Йоркская фондовая биржа генерирует около терабайта данных в день.
2. Объем хранилища социальной сети Facebook каждый день увеличивается на 500 терабайт.
3. Проект Internet Archive уже хранит 2 петабайта данных и прирастает 20 терабайтами в месяц.
4. Эксперименты на Большом адронном коллайдере могут генерировать около петабайта данных в секунду!

Стремительно растущий объем информации ставит перед нами новые сложные задачи по организации ее хранения и обработки. В статье мы постараемся проанализировать то, как изменился традиционный подход к

организации данных, а также проанализируем новые инструменты работы с большими данными.

2. Что такое «большие данные»?

Итак, данных очень много. Но что такое «много» ? Где тот порог, преодолев который, данные становятся «большими»? Часто используется характеристика, данная исследовательской компанией Gartner: ««Большие данные» характеризуются объемом, разнообразием и скоростью, с которой структурированные и неструктурированные данные поступают по сетям передачи в процессоры и хранилища, наряду с процессами преобразования этих данных в ценную для бизнеса информацию» [2].

Как видно из этого определения, большие данные имеют четыре основные характеристики: объем, разнообразие, скорость и ценность. Рассмотрим их подробнее:

1. **Объем.** Нарастающее количество данных, создаваемых как людьми, так и машинами, предъявляет к IT инфраструктуре новые требования в отношении хранения, обработки и предоставления доступа.
2. **Разнообразие.** Данные содержат разнообразную информацию, представленную разными структурами. Со всем этим, от логов доступа к веб-серверу до операций по кредитным картам, от результатов научных экспериментов до фотографий и видео, необходимо уметь работать.
3. **Скорость.** Важно осознавать, что под скоростью понимается не только скорость, с которой данные поступают в хранилище, но и скорость с которой важная информация из этих данных извлекается.
4. **Ценность.** Большие объемы данных — это ценный ресурс. Но еще ценнее он становится, если позволяет отвечать на насущные вопросы или вопросы, которые могут возникнуть в будущем.

3. Подходы к работе с большими данными

Так сложилось, что инструменты, существовавшие до недавнего времени, оказались не способны справиться с большими объемами. О проблемах, пришедших с эпохой больших данных, способах их преодоления и новых инструментах поговорим в следующих разделах.

3.1. Реляционные СУБД

До определенного момента, практически единственным ответом на вопрос «как хранить и обрабатывать данные?» являлась какая-нибудь реляционная СУБД. Но с увеличением объемов появились проблемы, с которыми классическая реляционная архитектура не справлялась, поэтому инженерам пришлось придумывать новые решения. Попробуем представить те шаги, которые можно предпринять, если СУБД прекращает справляться с объемом выполняемых операций:

1. Естественным первым шагом является попробовать наименее затратные способы. Самый простой, при наличии финансов, способ — это ничего не делать, а просто купить **более мощное оборудование (вертикальное масштабирование)**. Однако бесконечно мощного сервера не существует, а значит вертикальный рост конечен.
2. Более затратный способ — это **оптимизировать запросы**, проанализировав планы их исполнения, и **создать дополнительные индексы**. Такой метод может принести временное облегчение, но дополнительные индексы порождают дополнительные операции, а с ростом объемов обрабатываемых данных эти дополнительные операции приводят к деградации.
3. Следующим шагом может быть внедрение **кэша на чтение**. При правильной организации такого решения, мы можем избавить СУБД от существенной части операций чтения, но жертвуем строгой консистентностью данных. К тому же, этот подход приводит к усложнению клиентского ПО.
4. Выстраивание операций вставки/обновления в **очередь** — неплохое решение, но размер очереди ограничен. К тому же, для обеспечения строгой консистентности, нам придется организовать персистентность самой очереди, а это непростая задача.
5. Наконец, когда все прочие способы перестают работать, наступает момент пересмотреть способ организации самих данных. В первую очередь — произвести **денормализацию** схемы, чтобы уменьшить число нелокальных обращений.
6. Ну а когда и это не работает, то остается только **масштабировать горизонтально**, т.е. разносить вычисления на разные узлы. Здесь приходится окончательно попрощаться с нормализацией и внешними ключами, к тому же нужно ответить на вопросы «по каким признакам распределять новые кортежи по узлам?» и «как произвести миграцию существующей схемы?».

Подводя итоги, можно заключить, что попытки приспособить реляционную СУБД к работе с большими данными приводят к следующему:

1. Отказу от строгой консистентности.
2. Уходу от нормализации и внедрение избыточности.
3. Потере выразительности языка SQL и необходимости моделировать часть его функций программно.
4. Существенному усложнению клиентского программного обеспечению.
5. Сложности поддержания работоспособности и отказоустойчивости получившегося решения.

Необходимо, правда, отметить, что производители реляционных СУБД осознают все эти проблемы и уже начали предлагать масштабируемые кластерные решения. Однако стоимость внедрения и сопровождения подобных решений зачастую не окупается.

3.2. NoSQL

Взглянув на выводы из предыдущего раздела, в голове сразу рождается довольно очевидная мысль — а почему бы не спроектировать архитектуру, способную адаптироваться к возрастающим объемам данных и эффективно их обрабатывать? Подобные мысли привели к появлению движения NoSQL.

Сразу хочется обратить внимание, что NoSQL не подразумевает бездумного отказа от всех принципов реляционной модели. Более того, термин «NoSQL» впервые был использован в 1998 году для описания реляционной базы данных, не использовавшей SQL [3]. Просто теоретики и практики данного подхода справедливо утверждают, что при выборе инструментария необходимо отталкиваться от задачи, а реляционные СУБД подходят не всегда, особенно в эпоху больших данных. Популярность NoSQL стал набирать в 2009 г, в связи с появлением большого количества веб-стартапов, для которых важнейшей задачей является поддержание постоянной высокой пропускной способности хранилища при неограниченном увеличении объема данных. Рассмотрим основные особенности NoSQL подхода [4,5]:

1. **Исключение излишнего усложнения.** Реляционные базы данных выполняют огромное количество различных функций и обеспечивают строгую консистентность данных. Однако для многих приложений подобный набор функций, а также удовлетворение требованиям ACID являются излишними.
2. **Высокая пропускная способность.** Многие NoSQL решения обеспечивают гораздо более высокую пропускную способность

данных нежели традиционные СУБД. Например, колоночное хранилище Hypertable, реализующее подход Google Bigtable, позволяет поисковому движку Zvent сохранять около миллиарда записей в день. В качестве другого примера можно привести саму Bigtable, способную обработать 20 петабайт информации в день [6].

3. **Неограниченное горизонтальное масштабирование.** В противовес реляционным СУБД, NoSQL решения проектируются для неограниченного горизонтального масштабирования. При этом добавление и удаление узлов в кластере никак не сказывается на работоспособности системы. Дополнительным преимуществом подобной архитектуры является то, что NoSQL кластер может быть развернут на обычном аппаратном обеспечении, существенно снижая стоимость всей системы.

4. **Консистентность в жертву производительности.** При описании подхода NoSQL нельзя не упомянуть теорему CAP. Следуя этой теореме, многие NoSQL базы данных реализуют доступность данных (availability) и устойчивость к разделению (partition tolerance), жертвуя консистентностью в угоду высокой производительности. И действительно, для многих классов приложений строгая консистентность данных — это то, от чего вполне можно отказаться.

3.2.1 Классификация NoSQL хранилищ

На сегодняшний день создано большое количество NoSQL решений. Все они основываются на четырех принципах из предыдущего раздела, но могут довольно сильно отличаться друг от друга. Многие теоретики и практики создавали свои собственные классификации, но наиболее простой и общеупотребительной можно считать систему, основанную на используемой модели данных, предложенную Риком Кейтелем (Rick Cattel) [7]:

1. **Хранилища ключ-значение.** Отличительной особенностью является простая модель данных — ассоциативный массив или словарь, позволяющий работать с данными по ключу. Основная задача подобных хранилищ — максимальная производительность, поэтому никакая информации о структуре значений не сохраняется.

2. **Документные хранилища.** Модель данных подобных хранилищ позволяет объединять множество пар ключ-значение в абстракцию, называемую «документ». Документы могут иметь вложенную структуру и объединяться в коллекции. Однако это скорее удобный способ логического объединения, т.к. никакой жесткой схемы у документов нет и множества пар ключ-значение, даже в рамках одной коллекции, могут быть абсолютно произвольными. Работа с документами производится по ключу, однако существуют решения,

позволяющие осуществлять запросы по значениям атрибутов.

3. **Колоночные хранилища.** Этот тип кажется наиболее схожим с традиционными реляционными СУБД. Модель данных хранилищ подобного типа подразумевает хранение значений как неинтерпретируемых байтовых массивов, адресуемых кортежами <ключ строки, ключ столбца, метка времени> [6]. Основой модели данных является колонка, число колонок для одной таблицы может быть неограниченным. Колонки по ключам объединяются в семейства, обладающие определенным набором свойств.

4. **Хранилища на графах**. Подобные хранилища применяются для работы с данными, которые естественным образом представляются графами (например, социальная сеть). Модель данных состоит из вершин, ребер и свойств. Работа с данными осуществляется путем обхода графа по ребрам с заданными свойствами.

Тип	**Примеры**
Хранилища ключ-значение	Redis Scalaris Riak Tokyo Tyrant
Документные хранилища	SimpleDB CouchDB MongoDB
Колоночные хранилища	BigTable Hbase Cassandra
Хранилища на графах	Neo4j

Табл. 1. Классификация NoSQL хранилищ по модели данных.

3.3. MapReduce

Пионером в области больших данных можно считать компанию Google, которая в 2003 г. описала распределенную файловую системы GFS [8], а в 2004 г. представила миру вычислительную модель MapReduce [9]. Именно эти публикации помогли разработчикам свободного поискового движка Apache Nutch создать проект Hadoop [10], который сегодня фактически стал синонимом термина «большие данные».

3.3.1 Hadoop

Перед тем как рассмотреть Hadoop внимательней, следует ответить на вопрос: «а зачем нужен еще один продукт, если многие NoSQL базы данных предоставляют интерфейсы для MapReduce вычислений?» [11, 12, 13]. Ответ можно получить, рассмотрев производительность современных жестких дисков. Средняя производительность жесткого диска сегодня ~100 МБ/с, что означает возможность прочитать 1 ТБ информации примерно за 2.5 часа. Улучшить такие удручающие показатели можно параллельным чтением с нескольких дисков. Например, тот же самый 1 ТБ можно прочесть со 100 дисков за 2 минуты. Но ведь почти все NoSQL решения поддерживают горизонтальное масштабирование, а следовательно и параллельные дисковые операции? И тут ключевым фактором становится время позиционирования головки. Для того, чтобы операции обновления и чтения были эффективными, NoSQL базам (CouchDB, MongoDB) приходится использовать структуры с произвольным доступом, например B-деревья [14]. А значит, если отказаться от произвольного обновления данных и обрабатывать весь набор последовательно, можно добиться серьезного прироста производительности. Именно этот принцип и положен в основу архитектуры Hadoop.

За хранение и организацию данных в Hadoop кластере отвечает распределенная файловая система HDFS [15]. При проектировании которой использовались следующие принципы:

1. Аппаратные сбои неизбежны. Поэтому HDFS реализует надежные алгоритмы репликации данных, а для метаданных файловой системы поддерживается журнал, позволяющий восстановить требуемое состояние.

2. Поточная обработка и большие объемы. HDFS устроена таким образом, чтобы обеспечить максимальную производительность поточного доступа к данным. К тому же структуры файловой системы оптимизированы для работы с большими файлами.

3. Локальность данных. Намного эффективней выполнять вычисления рядом с данными. HDFS предоставляет приложениям программный интерфейс, который позволяет выполнять вычисления ближе к необходимым данным, сокращая пересылки между узлами кластера.

Вычисления в Hadoop представляются в виде последовательности map и reduce задач. В начале вычислений входное множество данных разбивается на несколько подмножеств. Каждое подмножество обрабатывается на отдельном узле кластера. Map задача на каждом узле получает на вход множество пар ключ-значение и возвращает другое множество пар. Далее все пары

группируются по ключу, сортируются и подаются на вход reduce задаче, которая формирует финальный результат или вход для другой map задачи [1].

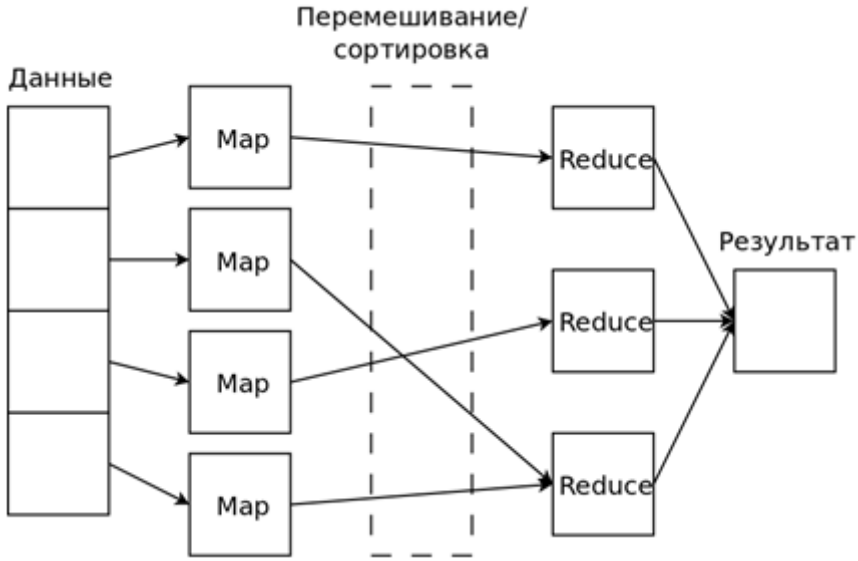


Рис. 1. Схема MapReduce вычислений.

Одним из интересных примеров эффективности Hadoop является тест скорости сортировки [16], целью которого является сортировка записей по 100 байт — 10 байт на ключ и 90 байт на значение. В ноябре 2008 г. рекордом в сортировке 1 ТБ владел кластер компании Google — 68 с. А в мае 2009 г. вышел отчет Yahoo! в котором утверждалось, что их Hadoop кластер отсортировал тестовый набор в 1 ТБ за 62 с! [17]

Вокруг Hadoop сформировалась целая экосистема проектов:

1. Hive — распределенное хранилище данных. Hive управляет данными в HDFS и предоставляет язык запросов HiveQL, основанный на SQL. Запросы HiveQL автоматически транслируются в MapReduce задачи [18].
2. Pig — среда исполнения и высокоуровневый язык, для описания вычислений в Hadoop. Программы Pig также транслируются в MapReduce задачи [19].
3. Hbase — распределенная колоночная база данных. Hbase использует HDFS, как хранилище и поддерживает как пакетные вычисления, так

и произвольный доступ [20].

4. ZooKeeper — высокодоступный координационный сервис, используемый для построения распределенных приложений [21].

3.3.2 Disco

Управление процессами в распределенной системе — сложная задача. А сложнее всего справиться с частичными отказами, когда ошибки в отдельных процессах не должны влиять на вычисления в общем. MapReduce избавляет разработчика от необходимости думать об ошибках, требуется только лишь обеспечить код двух функций: map и reduce. Об остальном позаботится реализация, автоматически определив неудачно завершившиеся задачи и перезапустив их. Такая возможность возникает, потому что задачи независимы, ибо не имеют разделяемых ресурсов [22]. Подобная особенность делает модель акторов [23] идеальной для реализации MapReduce фреймворка.

Этим и воспользовались разработчики исследовательского центра Nokia, создав проект Disco [24]. Особенностью Dicso является то, что ядро системы разработано на функциональном языке Erlang [25], снискавшем славу инструмента, отлично подходящего для программирования распределенных вычислений на основе модели акторов. Компания Ericsson, использующая язык для программирования коммутационных узлов своей телефонной сети, даже заявила о достижении показателя отказоустойчивости оборудования в 99.9999999% [26].

Hadoop, Disco и подобные им проекты отлично выполняют задачу распределенной пакетной обработки больших объемов данных. Фокус на пакетной обработке, в частности, приводит к тому, что вычисления происходят с большой задержкой. Подобные задержки могут быть неприемлемы для целого класса задач, где ответы на вопросы нужно получать незамедлительно.

3.4. Обработка потоков событий в реальном времени

Несколько лет, с момента первого публичного релиза Hadoop, пакетная обработка являлась, пожалуй, единственным способом анализа больших данных. Однако развитие таких приложений как поисковые системы реального времени, высокочастотная торговля и социальные сети диктовало необходимость мгновенно реагировать на новую информацию. Индустрии все больше не хватало «Hadoop реального времени». Эту нехватку восполнил фреймворк S4 (Simple Scalable Streaming System) от компании Yahoo!.

3.4.1 Yahoo! S4

Разработчики S4 писали, что изначально рассматривали возможность адаптировать Hadoop для вычислений на неограниченных потоках событий в

реальном времени, но архитектура Hadoop оптимизирована для пакетной обработки статических данных, что делает создание универсальной системы слишком трудоемким и сложным процессом [27].

Проектируемая система должна была удовлетворять следующим требованиям:

1. Предоставлять простой программный интерфейс поточной обработки данных.

2. Обеспечивать высокую доступность кластера.

3. Минимизировать задержки, используя только оперативную память узлов кластера.

4. Архитектура должна быть децентрализованной и симметричной. Все узлы идентичны, нет единой точки отказа.

Для упрощения первоначальной реализации были введены следующие допущения:

1. Восстановление от ошибок может приводить к потере состояния процессов на данном узле.

2. Динамическое добавление и удаление узлов в кластер невозможно.

Важно отметить, что первое допущение так и оставалось в реализациях S4, вплоть до версии вышедшей в августе 2012 г. Добавление механизма контрольных точек позволило частично решить проблему утери текущего состояния.

Вычисления в S4 естественным образом представляются графом, вершинами которого являются вычислительные элементы (PE), а ребрами — потоки событий. Событие представляет из себя кортеж именованных значений. Именование является важным для реализации группирования потоков. Так как архитектура S4 подразумевает хранение состояния в памяти PE, то часто бывает важно направлять кортежи, удовлетворяющие определенным критериями, на заданные узлы. Достигается это объединением потоков по именам значений в кортежах. Вычислительный элемент, получая на входе события из одного или нескольких потоков, может или создать новый выходной поток или опубликовать результат. Еще одним примитивом S4, который, однако, не является элементом графа, является адаптер. Адаптеры производят преобразование внешних входных/выходных потоков в необходимый формат.

О производительности S4 можно судить по результатам,опубликованным авторами в [27]. В качестве эксперимента приводится приложение производящее расчет CTR различных блоков на странице поисковой выдачи. Целью приложения является определение рекламных блоков с низким CTR и

исключение их дальнейшего показа. В работе утверждается, что производительность кластера из 16 четырехядерных серверов с 2 ГБ памяти начала падать при достижении пропускной способности в 7268 событий/с или 9.7 Мбит/с.

В другой работе [28] было проведено синтетическое нагрузочное тестирование кластера S4. Из эксперимента сделаны следующие выводы:

1. Добавление узлов в кластер не всегда приводит к повышению производительности.

2. Распределение событий по узлам неравномерно, что может привести к общему падению производительности.

3. Механизм обеспечения отказоустойчивости также может приводить к падению производительности. При выходе из строя одного из узлов S4 пытается перераспределить нагрузку на другие узлы, что может привести к их деградации.

4. Для обмена сообщениями используется протокол UDP без программного подтверждения доставки, что может приводить к потере данных.

3.4.2 Storm

Публичный релиз проекта Storm [29] от компании Twitter был сделан примерно год спустя первого публичного релиза S4. Не удивительно, что концептуально фреймворки очень похожи. Однако Storm, как заявляют авторы, устраняют некоторые недостатки продукта Yahoo!.

Основными свойствами Storm являются:

1. Широкий набор вариантов использования. Storm может быть использован для непрерывных вычислений над потоками событий, непрерывного обновления баз данных, распараллеливания сложных вычислений (распределенный RPC) и др.

2. Масштабирование. Storm поддерживает прозрачное горизонтальное масштабирование.

3. Гарантия сохранности данных. Storm, в отличие от S4, гарантирует обработку каждого сообщения.

4. Отказоустойчивость. Если во время вычислений происходит отказ оборудования или возникает ошибка, Storm перераспределяет задания на другие узлы.

5. Независимость от языка программирования. Вычисления можно программировать на любом языке.

Вычисления в Storm представляются графами, называемыми топологиями (рис. 2). Вершины графа определяют вычисления, а ребра создают маршруты передачи данных. Фреймворк определяет два основных вида вершин: труба (spout) и молния (bolt). Трубы, как и следует из названия, являются источниками потоков данных в топологии. По сути они определяют способы получения внешних данных и преобразования их в потоки кортежей. Молнии подключаются к одному или нескольким потокам, производят вычисления и, возможно, создают новые потоки.

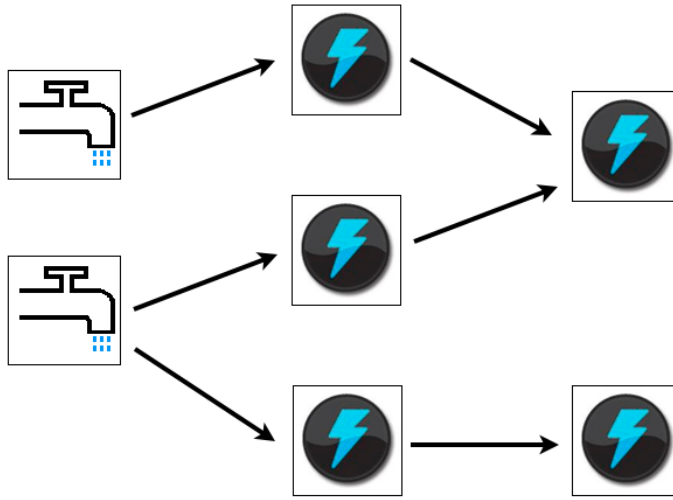


Рис. 2. Топология Storm.

Интересным представляется то, как Storm гарантирует обработку всех событий. Учитывая графовую структуру топологии, каждый кортеж может порождать целое дерево производных кортежей. Storm отслеживает состояние этого дерева и считает первоначальный кортеж полностью обработанным, когда обработан каждый элемент дерева. Если кортеж полностью не обработан по истечению заданного временного интервала, то вычисления повторяются. Этот механизм, правда, может приводить к повторной обработке одних и тех же данных и получению неверных результатов. Решается эта проблема использованием транзакционных топологий, гарантирующих единственность вычислений над каждым событием.

В виду молодости проекта Storm достоверных публикаций о производительности найти не удалось. Однако авторы утверждают, что одно из первых приложений, разработанное с использованием фреймворка

обрабатывало 1 млн сообщений/с на кластере из 10 машин, одновременно делая несколько сотен запросов/с к базе данных.

4. Заключение

В этой работе были рассмотрены проблемы, которые поставили перед реляционными СУБД большие данные. Проанализировав возможные пути решения этих проблем, мы указали на те концептуальные ограничения, которые не позволяют классической реляционной архитектуре справляться со стремительно возрастающим объемом информации. Далее были рассмотрены три подхода к работе с большими данными: NoSQL, MapReduce и обработка потоков событий в реальном времени. Мы обратили внимание на те архитектурные особенности, которые позволяют каждому из них эффективно решать поставленную задачу. Важно отметить, что ни один из представленных подходов не предлагает решения всех возможных задач, которые возникли в контексте больших данных. Каждый из них эффективно решает свой класс задач, позволяя пользователю выбрать наиболее подходящий для него инструмент.

Список литературы

[1] Tom White. Hadoop: The Definitive Guide, 3rd Edition. O'Reilly Media, 2012, 688 p.
[2] Mark A. Beyer, Douglas Laney. The Importance of «Big Data»: A Definition. http://www.gartner.com/DisplayDocument?id=2057415, 21 June 2012.
[3] Carlo Strozzi. NoSQL: A Relational Database Management System.http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page
[4] Jaroslav Pokorny. NoSQL databases: a step to database scalability in web environment. Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, p. 278-283, ACM New York, NY, USA, 2011.
[5] Christof Strauch. NoSQL Databases. http://www.christof-strauch.de/nosqldbs.pdf
[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: a distributed storage system for structured data. Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, vol. 7, p. 15-15, USENIX Association Berkeley, CA, USA, 2006.
[7] Rick Cattel. Scalable SQL and NoSQL data stores. ACM SIGMOD Record, 39(4), p. 12-27, ACM New York, NY, USA, December 2010.
[8] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google File System. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.
[9] Jeffrey Dean, Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, vol. 6, p. 10-10, USENIX Association Berkeley, CA, USA, 2004.
[10] Apache Hadoop. http://hadoop.apache.org/
[11] Apache CouchDB. http://couchdb.apache.org/
[12] MongoDB. http://www.mongodb.org/
[13] Riak. http://basho.com/products/riak-overview/
[14] J. Chris Anderson, Jan Lehnardt, Noah Slater. CouchDB: The Definitive Guide. O'Reilly Media, 2010, 272 p.

[15] Konstantin Shvachko, Hairong Kuang, Sanjai Radia, Robert Chansler. The Hadoop Distributed File System. MSST '10 Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010, pp. 1-10.
[16] Sort Benchmark Home Page. http://sortbenchmark.org/
[17] Ajay Anand. Hadoop Sorts a Petabyte in 16.25 Hours and a Terabyte in 62 Seconds. http://developer.yahoo.com/blogs/hadoop/posts/2009/05/hadoop_sorts_a_petabyte_in_162/, 2009.
[18] Apache Hive. http://hive.apache.org/
[19] Apache Pig. http://pig.apache.org/
[20] Apache Hbase. http://hbase.apache.org/
[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.
[22] Сергей Кузнецов. К свободе от проблемы Больших Данных. «Открытые системы», №02, 2012.
[23] G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. Cambridge, MA, USA: MIT Press, 1986.
[24] The Disco Project. http://discoproject.org/
[25] Erlang Programming Language. http://www.erlang.org/
[26] Joe Armstrong. Concurrency Oriented Programming in Erlang. http://ll2.ai.mit.edu/talks/armstrong.pdf, November 2002.
[27] Leonardo Neumeyer, Bruce Robbins, Anish Nair, Anand Kesari. S4: Distributed Stream Computing Platform. Data Mining Workshops (ICDMW), 2010 IEEE International Conference, 2010.
[28] Jagmohan Chauhan, Shaiful Chowdhury and Dwight Makaroff, Performance Evaluation of Yahoo! S4: A First Look, IEEE Seventh International Conference on P2P, Parallel, GRID, Cloud and Internet computing, 2012.
[29] Storm: Distributed and Fault-tolerant realtime computation. http://storm-project.net/

Big data: modern approaches to storage and analysis

Pavel Klemenkov
MSU, Moscow, Russia
parser@cs.msu.su
Sergey Kuznetsov
ISP RAS, Moscow, Russia
kuzloc@ispras.ru

Abstract: Big data challenged traditional storage and analysis systems in several new ways. In this paper we try to figure out how to overcome this challenges, why it's not possible to make it efficiently and describe three modern approaches to big data handling: NoSQL, MapReduce and real-time stream processing. The first section of the paper is the introduction. The second section discuss main issues of Big Data: volume, diversity, velocity, and value. The third section describes different approaches to solving the problem of Big Data. Traditionally one might use a relational DBMS. The paper propose some steps that allow to continue RDBMS using when it's capacity becomes not enough. Another way is to use a NoSQL approach. The basic ideas of the NoSQL approach are: simplification, high throughput, and unlimited scaling out. Different kinds of NoSQL stores allow to use such systems in different applications of Big Data. MapReduce and it's free implementation Hadoop may be used to provide scaling out Big Data analytics. Finally, several data management products support real time stream processing under Big Data. The paper briefly overviews these products. The final section of the paper is the conclusion.

Keywords: big data; relational model; nosql; mapreduce; hadoop; s4; storm

References

[1]. Tom White. Hadoop: The Definitive Guide, 3rd Edition. O'Reilly Media, 2012, 688 p.
[2]. Mark A. Beyer, Douglas Laney. The Importance of «Big Data»: A Definition. http://www.gartner.com/DisplayDocument?id=2057415, 21 June 2012.
[3]. Carlo Strozzi. NoSQL: A Relational Database Management System.http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page
[4]. Jaroslav Pokorny. NoSQL databases: a step to database scalability in web environment. Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, p. 278-283, ACM New York, NY, USA, 2011.
[5]. Christof Strauch. NoSQL Databases. http://www.christof-strauch.de/nosqldbs.pdf
[6]. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: a distributed storage system for structured data. Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, vol. 7, p. 15-15, USENIX Association Berkeley, CA, USA, 2006.
[7]. Rick Cattel. Scalable SQL and NoSQL data stores. ACM SIGMOD Record, 39(4), p. 12-27, ACM New York, NY, USA, December 2010.
[8]. Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google File System. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.

[9]. Jeffrey Dean, Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, vol. 6, p. 10-10, USENIX Association Berkeley, CA, USA, 2004.
[10]. Apache Hadoop. http://hadoop.apache.org/
[11]. Apache CouchDB. http://couchdb.apache.org/
[12]. MongoDB. http://www.mongodb.org/
[13]. Riak. http://basho.com/products/riak-overview/
[14]. J. Chris Anderson, Jan Lehnardt, Noah Slater. CouchDB: The Definitive Guide. O'Reilly Media, 2010, 272 p.
[15]. Konstantin Shvachko, Hairong Kuang, Sanjai Radia, Robert Chansler. The Hadoop Distributed File System. MSST '10 Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010, pp. 1-10.
[16]. Sort Benchmark Home Page. http://sortbenchmark.org/
[17]. Ajay Anand. Hadoop Sorts a Petabyte in 16.25 Hours and a Terabyte in 62 Seconds. http://developer.yahoo.com/blogs/hadoop/posts/2009/05/hadoop_sorts_a_petabyte_in_162/, 2009.
[18]. Apache Hive. http://hive.apache.org/
[19]. Apache Pig. http://pig.apache.org/
[20]. Apache Hbase. http://hbase.apache.org/
[21]. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.
[22]. Sergej Kuznetsov. K svobode ot problemy Bol'shih Dannyh [Toward the freedom from the Big Data problem]. «Otkrytye sistemy», №02, 2012 (in Russian).
[23]. G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. Cambridge, MA, USA: MIT Press, 1986.
[24]. The Disco Project. http://discoproject.org/
[25]. Erlang Programming Language. http://www.erlang.org/
[26]. Joe Armstrong. Concurrency Oriented Programming in Erlang. http://ll2.ai.mit.edu/talks/armstrong.pdf, November 2002.
[27]. Leonardo Neumeyer, Bruce Robbins, Anish Nair, Anand Kesari. S4: Distributed Stream Computing Platform. Data Mining Workshops (ICDMW), 2010 IEEE International Conference, 2010.
[28]. Jagmohan Chauhan, Shaiful Chowdhury and Dwight Makaroff, Performance Evaluation of Yahoo! S4: A First Look, IEEE Seventh International Conference on P2P, Parallel, GRID, Cloud and Internet computing, 2012.
[29]. Storm: Distributed and Fault-tolerant realtime computation. http://storm-project.net/

Web-приложения и данные: проблемы абстракции и масштабируемости

Андрей Посконин, <aposk@yandex.ru>

Аннотация. Сегодня всё чаще звучит утверждение, что универсальных хранилищ данных не существует.

Несмотря на то, что SQL-ориентированные системы управления базами данных (СУБД) широко применялись и применяются сегодня, масштабирование приложений при использовании этих систем сильно затруднено. В связи с этим появилось много различных хранилищ данных, стремящихся соответствовать требованиям современных высоконагруженных Web-приложений. В настоящее время наблюдается тенденция к использованию нескольких технологий и систем в рамках одного приложения для решения различных задач. В данной статье рассматриваются основные подходы к реализации доступа к данным и проблемы абстракции от используемых хранилищ, а также предлагаются некоторые принципы организации слоя работы с данными при использовании нескольких хранилищ данных разного типа.

Ключевые слова. Web-приложения, масштабируемость, SQL, NoSQL, абстракция от деталей хранения данных.

1. Введение

В последнее время Web-приложения приобрели огромную популярность. С развитием Интернета всё острее встаёт проблема разработки приложений, которые могут легко масштабироваться, то есть адаптироваться к постоянно возрастающим нагрузкам. Важнейшую роль в возможностях такой адаптации играют используемые технологии хранения данных, так как на уровне приложения масштабирование является относительно легкой задачей и сводится к добавлению новых серверов, обрабатывающих запросы.

Наиболее часто используемой для Web-приложений архитектурой является Модель – Вид – Контроллер (Model – View – Controller, MVC). MVC – это архитектурный шаблон, который разбивает приложение на три части: Модель (объектная модель, бизнес-логика и доступ к данным), Вид (отображение данных, генерация страниц) и Контроллер (маршрутизация запросов) [1]. Вид и Контроллер обычно поддерживаются различными фреймворками (программными каркасами) и библиотеками (например, Zend Framework [2]), тогда как реализация Модели целиком ложится на разработчика. В данной статье будут рассматриваться вопросы, связанные с Моделью, которая обычно

состоит из объектной модели предметной области (ключевые абстракции и связи между ними), слоя доступа к данным, структур для проверки прав доступа, форм, правил проверки корректности данных и других компонентов, реализующих бизнес-логику приложения. Также Модель часто включает в себя слой сервисов [1]. Этот слой состоит из классов, которые манипулируют объектами предметной области и координируют выполнение сложных задач.

Объектная модель приложения практически всегда должна долговременно сохраняться, например, с использованием SQL-ориентированной СУБД. Так как объектная модель приложения и модель данных хранилища имеют различные структуры и оперируют разными понятиями, то необходим промежуточный слой доступа к данным, осуществляющий отображение между объектами языка программирования и базой данных и предоставляющий интерфейс вышестоящим уровням. В данной статье рассматриваются различные аспекты реализации доступа к данным: выбор подходящего хранилища, модели и уровня согласованности данных, взаимосвязь абстракции и производительности, а также совместное использование различных СУБД для решения задач масштабирования.

2. Web-приложения и SQL

SQL-ориентированные СУБД господствуют на рынке уже более 30 лет. Эти системы до сих пор широко используются для хранения данных в различных Web-приложениях, для них было разработано много методов, помогающих частично преодолеть проблемы масштабирования при увеличивающихся нагрузках. К сожалению, особенности архитектуры этих систем не позволяют им хорошо масштабироваться горизонтально, однако мощный язык запросов, богатые возможности, поддержка ACID-транзакций и большое число разработанных инструментов делают их популярным инструментом для решения многих задач.

2.1. SQL и ООП

Как уже было сказано, объектная модель приложения и модель данных SQL оперируют разными понятиями, поэтому напрямую их связать не получается, и зачастую приходится прибегать к объектно-реляционному отображению (Object-Relational Mapping, ORM). Возможны, например, следующие варианты:

- Отказ от полноценной объектной модели, работа с БД напрямую.
- Использование слоя абстракции от конкретной реализации языка SQL.
- Использование простых оберток для строк и столбцов таблиц (образцы Row Data Gateway и Table Data Gateway, реализованные, например, в [2]).
- Ручная реализация отображения объектов (требует написания большого количества кода, содержащего SQL-запросы для работы с данными).

- Использование ORM-библиотек (например, [3], [4]).

Здесь уровень абстракции каждого следующего решения выше, чем предыдущего. Это значит, что объектная модель приложения может становиться (почти) независимой от деталей хранения, что упрощает разработку и сопровождение кода. Кроме того, уровень абстракции напрямую связан с производительностью: чем выше уровень абстракции, тем больше промежуточных слоев и меньше возможностей для низкоуровневых оптимизаций. Чем большая нагрузка планируется на приложение, тем ниже должен быть уровень абстракции, чтобы обеспечить требуемую производительность. Например, ORM-библиотеки предоставляют богатую функциональность и высокий уровень абстракции, но использовать такую библиотеку в высоконагруженном приложении не представляется возможным из-за слишком медленной работы и невозможности использовать низкоуровневые оптимизации, зависящие от конкретной СУБД [5].

2.2. Проблемы реализации ORM

Рассмотрим ключевые вопросы реализации объектно-реляционного отображения. По распределению обязанностей можно выделить два основных образца реализации ORM:

- Active Record – обязанности по доступу к БД возлагаются на сами классы-сущности: в этом случае они будут иметь, помимо своих методов, такие методы, как Save(), Update() или Delete().
- Data Mapper – обязанности по доступу к БД возлагаются на отдельные объекты, что позволяет отделить объектную модель приложения от логики взаимодействия с хранилищем. Другим полезным образцом является Repository – скрытие всех деталей работы с хранилищем от объектной модели.

При работе с базой данных крайне важно минимизировать объем выбираемых данных, тем более не выбирать лишних данных, поэтому в ORM часто применяется так называемая «ленивая» загрузка (Lazy Loading), суть которой сводится к тому, что объекты и коллекции загружаются из базы данных только тогда, когда к ним производится обращение (при помощи классов-заместителей). Несмотря на то, что такая стратегия кажется разумной, она несет в себе много опасностей, таких как непреднамеренная загрузка всей коллекции, «путешествия» по графу объектов, вызывающие множество отдельных запросов и т.д. В этом плане даже при использовании библиотек для ORM необходимо помнить о специфике доступа к БД.

На слой поддержки ORM также возлагаются обязанности по недопущению дубликатов объектов с одинаковым первичным ключом, для чего обычно применяется образец Identity Map. Оптимизировать выполнение операций изменения данных помогает образец Unit of Work путем накопления операций изменения и выполнения затем всех операций сразу в одной транзакции.

Полезным также может являться механизм событий, срабатывающих при записи или чтении из БД, что помогает выполнять сложные проверки корректности данных и прав доступа на уровне приложения (подобный механизм событий реализован, например, в [4]). Более подробно об образцах, применяемых при реализации объектно-реляционного отображения, можно узнать, например, из [6].

Что касается самого отображения объектов на таблицы, то в ORM-библиотеках оно осуществляется с помощью метаданных (в виде XML, аннотаций и т.п.), которые определяют атрибуты и их типы, а также связи между объектами (таблицами). На основе метаданных ORM-библиотека может генерировать схему целевой базы данных. Существенным недостатком метаданных является усложнение и замедление работы приложения, однако кэширование позволяет бороться с этой проблемой.

Абстракция от конкретного хранилища данных отчасти достигается с помощью «объектной» модификации языка SQL и конструкторов запросов, позволяющих обойтись без смешения кода приложения и запросов. По «объектному» запросу строится запрос к целевой базе данных, который также может кэшироваться, чтобы избежать трансляции при каждом вызове.

Таковы основные механизмы, позволяющие до какой-то степени отделить объектную модель приложения от деталей работы с СУБД. К сожалению, применение «тяжелых» библиотек ORM очень негативно сказывается на производительности приложений и затрудняет оптимизацию.

2.3. Оптимизация работы приложений с SQL СУБД

Наиболее важным вопросом при проектировании Web-приложения является выбор уровня абстракции при доступе к БД. Высоконагруженные приложения обычно используют написанные вручную классы для сохранения и загрузки объектов из БД, позволяющие применять низкоуровневые оптимизации для каждой конкретной СУБД [5]. Далее приводятся некоторые общие способы, позволяющие повысить производительность работы с БД:

- Оптимизация SQL-запросов
- Анализ использования индексов
- Избежание «долгих» транзакций и работы в режиме auto-commit
- Выбор уровня изолированности транзакций
- Партицирование таблиц (разбиение большой таблицы на логические части по строкам или по столбцам)
- Денормализация таблиц
- Кэширование объектов и результатов «долгих» запросов

2.4. Масштабирование на уровне SQL СУБД

Если оптимизация работы с БД и покупка более мощного оборудования (вертикальное масштабирование) не могут решить проблем с возрастающей нагрузкой, то существуют некоторые возможности горизонтального масштабирования, хотя и довольно ограниченные [7], [8]. Рассмотрим сначала репликацию данных (поддержку нескольких серверов с одними и теми же данными). Выделяют два вида репликации: синхронную (данные всегда совпадают) и асинхронную (данные на копиях могут в течение какого-либо промежутка времени различаться). Асинхронная репликация работает быстрее синхронной, однако при её использовании возникает проблема появления устаревших данных. Рассмотрим две схемы репликации, позволяющие масштабировать нагрузку:

- Master – Slaver: один сервер является ведущим, все запросы на запись идут к нему, а он пересылает измененные данные на второстепенный сервер, с которого может осуществляться только чтение. Если используется асинхронная репликация и устаревшие данные недопустимы, то можно осуществлять чтения с ведущего сервера. Схема Master-Slave позволяет масштабировать чтения.
- Master – Master: оба сервера доступны как для чтения, так и для записи. Эта схема сложнее в реализации, но позволяет масштабировать операции записи.

Кроме масштабирования, репликация применяется и для повышения надежности, а также для других целей (например, можно держать один сервер в запасе и выполнять на нем «долгие» аналитические запросы).

Кроме репликации существует еще одна схема масштабирования, называемая шардингом (разделением данных). Горизонтальный шардинг подразумевает перемещение части строк большой таблицы на новый сервер, что позволит распараллелить обращения к ним. Вертикальный шардинг подразумевает перемещение каких-либо таблиц (столбцов таблиц) на отдельный сервер с той же целью. Обычно шардинг и репликация применяются в комплексе, чтобы обеспечить не только распараллеливание обработки запросов, но и надежность.

К сожалению, горизонтальное масштабирование традиционных SQL-ориентированных систем довольно затруднительно, непрозрачно для приложений, затрудняет поддержку целостности данных и выполнение запросов, затрагивающих несколько разделенных частей таблицы.

Чтобы преодолеть трудности масштабирования, были предложены и разработаны распределенные SQL-ориентированные СУБД, которые масштабируются легче и поддерживают основные возможности SQL. Такие системы, однако, не лишены недостатков: проектирование схемы становится

сложнее, при написании запросов надо учитывать расположение данных, а ряд возможностей SQL недоступен.

2.5. Преимущества и проблемы SQL

К несомненным преимуществам модели данных и языка SQL можно отнести богатые возможности, поддержку интуитивно понятной семантики ACID-транзакций, четко определенную схему данных, возможность задавать ограничения целостности на уровне БД, большое количество инструментов, качество и зрелость программных продуктов.

Однако, как было показано ранее, SQL-ориентированные СУБД испытывают серьезные проблемы, связанные с масштабированием. Но это не единственная проблема: также SQL не очень хорошо сочетается с ООП, требуя достаточно сложного объектно-реляционного отображения. Рассмотрим, например, реализацию динамического набора атрибутов. При использовании SQL СУБД есть следующие варианты:

- Таблица с большим числом столбцов.
- Базовая таблица и таблицы с атрибутами для каждого подтипа.
- Entity-Attribute-Value, то есть поддержка трех таблиц – сущностей, атрибутов и значений, а потом их дорогостоящее и сложное соединение.

Ни один из этих вариантов не подойдет при большом количестве разнообразных объектов и атрибутов [9]. Возможно, для решения этой задачи лучше подойдет СУБД с моделью данных, отличной от SQL.

3. Web-приложения и NoSQL

Для преодоления ограничений SQL-ориентированных систем появилось множество новых распределенных систем, объединенных в класс так называемых NoSQL-систем (Not Only SQL). Эти системы отказываются от модели данных SQL, а также от поддержки строгой согласованности данных и ACID-транзакций. Взамен они предоставляют высокую надежность, производительность (благодаря хранению данных в основной памяти) и хорошую горизонтальную масштабируемость (благодаря репликации и шардингу). Модели данных, на которых строятся NoSQL-системы, могут быть как простыми (хранилища типа ключ-значение), так и более сложными (документные хранилища, хранилища расширяемых записей). Кроме того, существуют NoSQL-хранилища на основе облачных решений, что позволяет оптимизировать также и финансовые затраты [10].

3.1. Согласованность данных

В соответствии с утверждением, известным как теорема CAP [11], [12], распределенная система не может гарантировать одновременно:

- Согласованность данных (Consistency) - все копии объектов в любой момент времени находятся в согласованном состоянии. Таким образом, здесь понятие согласованности отличается от согласованности из свойств ACID-транзакций.
- Доступность (Availability) - система всегда отвечает на запросы.
- Устойчивость к разделению сети (Partition Tolerance) - система, даже будучи разделенной на части, продолжает обслуживать пользователей.

Каждая распределенная система реализует некий компромисс между этими свойствами. Другой, возможно, более важной, причиной отказа от строгой согласованности данных и распределенных транзакций является стремление уменьшить задержки и увеличить производительность [13]. Таким образом, в NoSQL-системах обычно поддерживаются более слабые модели согласованности – согласованность в конечном счете (Eventual Consistency) и её вариации [14]. Многие Web-приложения, однако, не требуют строгой согласованности и могут работать со слегка устаревшими данными. На практике значительная часть NoSQL-систем возвращает последние версии данных, допуская устаревшие данные только в случае сбоев или высокой нагрузки [15]. Некоторые из них поддерживают дополнительные гарантии согласованности, атомарные операции и другие возможности, упрощающие реализацию приложения.

3.2. NoSQL и ООП

Модели данных NoSQL-систем являются более простыми и гибкими, чем SQL. Например, динамический набор атрибутов может быть легко реализован при использовании документного хранилища, например, MongoDB [9]. Документ – это произвольный набор атрибутов, который может включать вложенные документы и коллекции документов, при этом поддерживается индексация и поиск по атрибутам. Такая модель данных лучше сочетается с ООП, чем SQL, а объектно-документное отображение (Object-Document Mapping, ODM) осуществляется существенно проще, чем ORM. Для ODM также уже существуют библиотеки, например Doctrine ODM [4], MongoMapper [16].

Более простой моделью данных является модель «ключ-значение». Хранилища этого типа поддерживают только доступ по уникальному ключу, но могут с успехом применяться для хранения сериализованных объектов, если не требуется поиск по дополнительным атрибутам. Такие хранилища просты в использовании, обеспечивают очень высокую производительность и могут быть использованы и как отдельное хранилище, и для кэширования каких-либо данных. Подробный обзор и сравнение различных хранилищ данных можно найти в [17].

3.3. Преимущества и проблемы NoSQL

Основными преимуществами систем класса NoSQL является их высокая производительность, хорошая горизонтальная масштабируемость и надежность. К сожалению, отсутствие строгой согласованности данных и поддержки ACID-транзакций ограничивают область применения этих систем.

В качестве примера рассмотрим реализованный в [18] прототип Web-форума, использующий NoSQL решения:

- MongoDB [19] (в конфигурации Replica-Set) в качестве основного хранилища данных (данные о пользователях, разделы, темы, сообщения). Для осуществления объектно-документного отображения использовалась библиотека Doctrine ODM.
- Membase [20] – распределенное хранилище типа «ключ-значение» для хранения данных пользовательских сессий и кэширования.

В качестве языка программирования использовался PHP5, приложение построено с применением Zend Framework [2]. Благодаря использованию NoSQL-решений может быть достигнута хорошая масштабируемость. Данное приложение не требует строгой согласованности данных и ACID-транзакций, поэтому может быть успешно реализовано и без них. Однако есть приложения, для которых наличие транзакционной семантики и контроль целостности данных являются критически важными (например, приложения в банковской сфере или в сфере электронной коммерции). Для таких приложений и задач лучше подходят SQL-ориентированные СУБД.

Проблема отсутствия универсальной СУБД приводит к новой тенденции – использованию нескольких различных систем в одном приложении для решения подходящих задач (“polyglot persistence”) [21], [22].

4. Совместное использование SQL и NoSQL-систем

Существует несколько вариантов совместного применения SQL и NoSQL-систем:

- Использование NoSQL-хранилища в качестве кэша при использовании SQL-ориентированной СУБД. В этом случае операции чтения, не требующие строгой согласованности данных, могут читать данные из кэша, чтобы снять нагрузку с основной СУБД. В зависимости от используемого NoSQL-решения, такой кэш может поддерживать даже достаточно сложные запросы и при этом обеспечивать высокую производительность.
- Использование NoSQL-хранилища через SQL-интерфейс. Этот подход позволяет переносить приложения, написанные под SQL-ориентированные СУБД, на NoSQL-системы с минимальными изменениями. Такой подход, однако, имеет некоторые ограничения, но зато не требует от разработчиков

изучения новых языков запросов и API. Например, GenieDB поддерживает и NoSQL-, и SQL-интерфейсы для доступа к данным [23].

- Репликация данных между SQL и NoSQL системами. Этот подход можно реализовать, например, с помощью Tungsten Replicator [24]. Таким образом, чтение и запись могут осуществляться из разных хранилищ (на уровне приложения возможно применение образца CQRS для разделения чтения и записи [25]).
- «Polyglot Persistence», то есть использование нескольких СУБД для решения различных задач внутри одного приложения [21], [22]. Этот подход будет подробнее рассматриваться далее.

4.1. «Polyglot Persistence»

Как было показано ранее, универсальной СУБД не существует, каждая конкретная система хорошо решает лишь определенный круг задач. Например, в [9] рассматривается совместное применение MySQL и MongoDB для реализации платформы электронной коммерции. Тем не менее, использование в одном Web-приложении нескольких разных хранилищ данных (даже при использовании библиотек объектного отображения) связано с определенными сложностями:

- Несколько различных языков запросов и API
- Сложно поддерживать согласованность данных между хранилищами
- Смена хранилища требует переписывания части приложения
- Проблема отображения объектов на разные хранилища

Существуют возможности частичного преодоления этих проблем (например, использование LINQ [26]), однако полностью решить их не удается. В данной ситуации может помочь применение слоя абстракции от деталей хранения, чтобы сделать объектную модель приложения максимально независимой от них. Кроме того, большинству проектов в начальной фазе развития трудно определиться с используемыми СУБД, и впоследствии приходится переписывать большую часть приложения. Поддержание определенного уровня абстракции может помочь более легкому переходу на другие системы хранения данных. В настоящее время появляются библиотеки отображения, поддерживающие (в ограниченной степени) отображение на разные источники данных (например, DataMapper [27]).

4.2. Абстракция от деталей хранения

Преодолеть проблемы отображения объектов на различные хранилища данных и сделать объектную модель приложения максимально независимой от деталей хранения можно с помощью применения дополнительного слоя абстракции. Основными его функциями могут быть следующие:

- Отображение объектов приложения на различные модели данных (SQL/NoSQL).
- Обеспечение независимости логики приложения от деталей реализации хранения данных.
- Поддержка абстрактного языка запросов, позволяющего выбирать объекты (примером такого языка может служить LINQ).
- Поддержка репликации и шардинга между различными хранилищами.
- Поддержка кэширования данных, по возможности, прозрачно для приложения.
- Обеспечение гибкости и расширяемости.

Несмотря на все преимущества подобного слоя абстракции, при реализации возникает много проблем.

4.3. Проблемы реализации и возможные решения

Основной проблемой здесь является обеспечение высокого уровня абстракции при сохранении приемлемой производительности и гибкости. Рассмотрим подробнее некоторые проблемы и возможные решения:

- Обеспечение независимости объектной модели приложения от деталей хранения может быть достигнуто с помощью применения образца Data Mapper [1].
- Поддержка абстрактного языка запросов может быть реализована с помощью конструкторов запросов. В этом случае отпадает необходимость в лексическом, синтаксическом разборе кода запроса, упрощается трансляция в язык запросов конкретного хранилища.
- Для описания правил отображения объектов требуются метаданные. С помощью метаданных (например, в виде XML-документов) может быть описана вся объектная модель приложения и правила её хранения (связи, репликация, хранилища для чтения и записи и т.д.) На основе метаданных также может осуществляться генерация кода. Для обеспечения производительности метаданные должны кэшироваться в основной памяти.
- Моделирование связей между объектами, находящимися в разных хранилищах, является еще одной сложной проблемой. Эти связи могут задаваться на уровне метаданных и реализовываться с помощью механизма «ленивой» загрузки. Возможен и вариант отказа от моделирования связей, что, однако, портит объектную модель и требует явных запросов. Компромиссом между этими подходами является автоматическая поддержка связей для объектов, находящихся в одном хранилище.

- Для обеспечения гибкости и расширяемости возможно поддержание нескольких уровней абстракции и интерфейсов. Кроме того, необходима возможность переопределять классы-мэпперы, чтобы обеспечить возможность оптимизации и расширяемости. При этом вышестоящие уровни не потребуют изменений.
- Для оптимизации обращения к хранилищам данных может быть применен образец Unit of Work. Кроме того, этот подход может использоваться для обработки ошибок и позволяет избежать компенсирующих операций при использовании хранилищ, не поддерживающих транзакции.
- Для усиления ограничений целостности на уровне приложения может быть реализован механизм событий, позволяющий проверять целостность данных перед записью.
- Для повышения производительности можно применять кэширование на разных уровнях. Так как NoSQL-системы в большинстве своем не поддерживают строгой согласованности данных, то кэширование не привнесёт в этом случае больших проблем.

5. Заключение

В этой статье был дан обзор основных проблем, возникающих при разработке Web-приложений: работа с данными, вопросы масштабирования и оптимизации, выбор хранилища данных и уровня абстракции для решения определенных задач. В целом можно сделать следующий вывод: выбор конкретной СУБД и методов работы с данными целиком определяется требованиями к приложению по части производительности, масштабируемости, надежности и скорости разработки. Для приложений, требующих транзакционной семантики и строгой согласованности данных по-прежнему лучше всего подходят SQL-ориентированные СУБД, в то время как для высокопроизводительных Web-приложений с высокими требованиями к горизонтальной масштабируемости лучше использовать распределённые SQL-ориентированные или NoSQL-системы.

Универсального решения проблемы работы с данными не существует, однако с помощью применения слоя абстракции можно облегчить разработку приложений при использовании нескольких хранилищ данных, а также сделать приложение более переносимым.

Список литературы

[1] M. Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, 2002.
[2] «Zend Framework», [В Интернете]. URL: http://framework.zend.com/.
[3] «Hibernate - Relational Persistence for Java and .NET», [В Интернете]. URL: http://hibernate.org/.
[4] «Doctrine Project», [В Интернете]. URL: http://www.doctrine-project.org/.

[5] «The case against ORM Frameworks in High Scalability Architectures», 2008. [В Интернете]. URL: http://highscalability.com/blog/2008/2/2/the-case-against-orm-frameworks-in-high-scalability-architec.html.
[6] J. Miller, «Design Patterns for Data Persistence», 2009. [В Интернете]. URL: http://msdn.microsoft.com/en-us/magazine/dd569757.aspx.
[7] A. Wiggins, «SQL Databases Don't Scale», 2009. [В Интернете]. URL: http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/.
[8] D. Obasanjo, «Building scalable databases: Denormalization, the NoSQL movement and Digg», 2009. [В Интернете]. URL: http://www.25hoursaday.com/weblog/2009/09/10/BuildingScalableDatabasesDenormalizationTheNoSQLMovementAndDigg.aspx.
[9] S. Francia, J. Hileman, «Augmenting RDBMS with MongoDB for eCommerce», 2011. [В Интернете]. URL: http://www.nosqldatabases.com/main/2011/4/11/augmenting-rdbms-with-mongodb-for-ecommerce.html.
[10] D. Florescu, D. Kossmann, «Rethinking Cost and Performance of Database Systems», *ACM SIGMOD Record*, т. 38, № 1, 2009.
[11] E. Brewer, «Towards Robust Distributed Systems», в *ACM Symposium on the Principles of Distributed Computing*, Portland, Oregon, 2000.
[12] S. Gilbert, N. Linch, «Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services», 2002.
[13] D. Abadi, «Problems with CAP, and Yahoo's little known NoSQL system», 2010. [В Интернете]. URL: http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html.
[14] W. Wogels, «Eventually Consistent,» *ACM Queue*, т. 6, № 6, 2008.
[15] H. Wada, A. Fekete, L. Zhaoy, K. Lee, A. Liu, «Data Consistency Properties and the Tradeoffs in Commercial Cloud Storages: the Consumers' Perspective» в *Conference on Innovative Data Systems Research*, 2011.
[16] «MongoMapper», [В Интернете]. URL: http://mongomapper.com/.
[17] R. Cattell, «Scalable SQL and NoSQL Data Stores», 2011. [В Интернете]. URL: http://www.cattell.net/datastores/Datastores.pdf.
[18] А. В. Посконин, «Новые направления в развитии СУБД: компромисс как основа архитектуры» в *Тезисы лучших дипломных работ факультета ВМК МГУ 2011 года*, Москва, 2011.
[19] «MongoDB», [В Интернете]. URL: http://www.mongodb.org/.
[20] «Membase Server», [В Интернете]. URL: http://www.couchbase.com/membase.
[21] M. Stonebraker, U. Çetintemel, «"One Size Fits All": An Idea Whose Time Has Come and Gone» в *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, Washington, 2005.
[22] M. Fowler, «Polyglot Persistence», 2011. [В Интернете]. URL: http://martinfowler.com/bliki/PolyglotPersistence.html.
[23] A. Snell-Pym, «One Database, Many Interfaces: A Look at SQL/NoSQL Integration Using GenieDB», *Cloudbook Journal*, т. 2, № 1, 2011.
[24] «Tungsten Replicator», [В Интернете]. URL: http://continuent.com/solutions/tungsten-replicator.
[25] M. Fowler, «Command Query Responsibility Segregation», 2011. [В Интернете]. URL: http://martinfowler.com/bliki/CQRS.html.
[26] «NoRM - MongoDB Driver for .NET», [В Интернете]. URL: https://github.com/atheken/NoRM.

[27] «DataMapper - Ruby Object Relational Mapper», [В Интернете]. URL: http://datamapper.org.

Web applications and data: achieving abstraction and scalability

Andrey Poskonin
MSU, CMC Department, Moscow, Russia
<aposk@yandex.ru>

Abstract: Nowadays it has become clear that no data store is all-purpose. SQL-based systems have been prevalent for several decades though this approach leads to serious issues in modern high scalability architectures. Thus, a number of different data stores have been designed to fulfill the needs of large-scale Web applications. Moreover, a single application often tends to use multiple data stores for different purposes and data. This paper covers common approaches to data persistence implementation and data store abstraction and also reveals basic considerations for design of a multi-store persistence layer for scalable Web applications. The paper consists of five sections. The first section (Introduction) discusses a problem of data management in the context of modern approaches to Web-applications design and development. The second section describes the most popular solutions of this problem based on traditional SQL-based DBMSs. Some limitations of these solutions are considered. The third section of the paper presents an overview of adventures and limitations of approached based on NoSQL DBMSs. A proposed novel approach of combine use SQL-based and NoSQL systems to develop and run Web-applications is a theme of the forth section. The final section concludes the paper.

Keywords: Web applications, scalability, SQL, NoSQL, data store abstraction.

References

[1]. M. Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, 2002.
[2]. «Zend Framework», [В Интернете]. URL: http://framework.zend.com/.
[3]. «Hibernate - Relational Persistence for Java and .NET», http://hibernate.org/.
[4]. «Doctrine Project», http://www.doctrine-project.org/.
[5]. «The case against ORM Frameworks in High Scalability Architectures», 2008. http://highscalability.com/blog/2008/2/2/the-case-against-orm-frameworks-in-high-scalability-architec.html.
[6]. J. Miller, «Design Patterns for Data Persistence», 2009. http://msdn.microsoft.com/en-us/magazine/dd569757.aspx.
[7]. A. Wiggins, «SQL Databases Don't Scale», 2009. http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/.
[8]. D. Obasanjo, «Building scalable databases: Denormalization, the NoSQL movement and Digg», 2009. http://www.25hoursaday.com/weblog/2009/09/10/BuildingScalableDatabasesDenormalizationTheNoSQLMovementAndDigg.aspx.
[9]. S. Francia, J. Hileman, «Augmenting RDBMS with MongoDB for eCommerce», 2011. http://www.nosqldatabases.com/main/2011/4/11/augmenting-rdbms-with-mongodb-for-ecommerce.html.
[10]. D. Florescu, D. Kossmann, «Rethinking Cost and Performance of Database Systems», ACM SIGMOD Record, т. 38, № 1, 2009.

[11]. E. Brewer, «Towards Robust Distributed Systems», в ACM Symposium on the Principles of Distributed Computing, Portland, Oregon, 2000.
[12]. S. Gilbert, N. Linch, «Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services», 2002.
[13]. D. Abadi, «Problems with CAP, and Yahoo's little known NoSQL system», 2010. URL: http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html.
[14]. W. Wogels, «Eventually Consistent,» ACM Queue, т. 6, № 6, 2008.
[15]. H. Wada, A. Fekete, L. Zhaoy, K. Lee, A. Liu, «Data Consistency Properties and the Tradeoffs in Commercial Cloud Storages: the Consumers' Perspective» в Conference on Innovative Data Systems Research, 2011.
[16]. «MongoMapper», http://mongomapper.com/.
[17]. R. Cattell, «Scalable SQL and NoSQL Data Stores», 2011. http://www.cattell.net/datastores/Datastores.pdf.
[18]. A.V. Poskonin, «Novye napravlenija v razvitii SUBD: kompromiss kak osnova arhitektury» [New directions of DBMS technology evolution: a tradeoff as a basis of architecture] v Tezisy luchshih diplomnyh rabot fakul'teta VMK MGU 2011 goda, Moskva, 2011 (in Russian).
[19]. «MongoDB», http://www.mongodb.org/.
[20]. «Membase Server», http://www.couchbase.com/membase.
[21]. M. Stonebraker, U. Çetintemel, «"One Size Fits All": An Idea Whose Time Has Come and Gone» в ICDE '05: Proceedings of the 21st International Conference on Data Engineering, Washington, 2005.
[22]. M. Fowler, «Polyglot Persistence», 2011. http://martinfowler.com/bliki/PolyglotPersistence.html.
[23]. A. Snell-Pym, «One Database, Many Interfaces: A Look at SQL/NoSQL Integration Using GenieDB», Cloudbook Journal, т. 2, № 1, 2011.
[24]. «Tungsten Replicator», http://continuent.com/solutions/tungsten-replicator.
[25]. M. Fowler, «Command Query Responsibility Segregation», 2011. http://martinfowler.com/bliki/CQRS.html.
[26]. «NoRM - MongoDB Driver for .NET», https://github.com/atheken/NoRM.
[27]. «DataMapper - Ruby Object Relational Mapper», http://datamapper.org.

Алгоритмы управления буферным пулом СУБД при работе с флэш-накопителями

Кузнецов С.Д., Прохоров А.А.
<kuzloc@ispras.ru>, <EmailToProkhorov@gmail.com>

Аннотация. Одним из важнейших механизмов повышения скорости работы современных СУБД является кэширование часто читаемых или записываемых данных в оперативной памяти. Классические алгоритмы замещения страниц БД в кэше стремятся минимизировать промахи буферного пула СУБД. Данный метод оптимизации негласно опирается на тот факт, что скорость записи и чтения данных одинакова. Постепенное совершенствование и удешевление технологии производства флэш-памяти привели к созданию твердотельных накопителей данных(SSD), которые в настоящее время все чаще используются как в персональных компьютерах, так и в системах хранения данных. Флэш-накопители имеют серьезные преимущества по сравнению с традиционными жесткими дисками, главные из которых – более высокая скорость чтения и записи, а также значительно меньшее время доступа к данным. Однако, самые распространенные виды флэш-памяти читают данные с большей скоростью, чем записывают. Из-за данной особенности использование классических алгоритмов замещения страниц при кэшировании дисковых данных неэффективно. В данной статье производится обзор современных алгоритмов управления буферным пулом СУБД, которые предназначены для работы с накопителями на флэш-памяти.

Ключевые слова. Буферный пул СУБД, кэширование дисковых данных, флэш-память, твердотельные накопители данных.

1. Введение

С появлением накопителей информации на флэш-памяти время доступа к данным снизилось с 10 мс., характерных для жестких дисков, до 0,1 мс. Однако, скорость работы современных СУБД по-прежнему лимитируется скоростью работы устройств хранения данных. В связи с этим для повышения скорости работы системы в целом выгодно кэшировать определенный набор данных БД в оперативной памяти.

При разработке алгоритмов управления буферным пулом решается задача минимизации математического ожидания среднего времени доступа к страницам данных. Для носителей информации на магнитных дисках скорость чтения и записи данных одинакова, поэтому выгодно хранить в буферном

пуле наиболее часто используемые страницы(алгоритм Белади) [1]. Такой алгоритм практически не реализуем, поэтому используются его эффективные приближения LRU, LFU, CLOCK и др. [2]

Разработчики флэш-накопителей были вынуждены не только добиваться превосходства над магнитными дисками по техническим характеристикам, но и стремиться к минимизации цены конечного устройства. Именно увеличение плотности хранения данных и экономия на вспомогательных управляющих элементах привели к ряду особенностей, характерных для наиболее распространенных видов флэш-памяти. Чтение и запись в этих устройствах производится блоками в несколько килобайт, удаление блоками размером в сотни килобайт, а запись может осуществляться только после удаления. Данные ограничения привели к асимметрии операций чтения и записи данных.

Для построения буферного пула, оптимизированного для работы с флэш-памятью, необходимо не только поддерживать высокий процент попаданий в дисковый кэш, но и минимизировать процент дорогостоящих промахов по записи. Для решения этих задач в последние годы было разработано несколько алгоритмов, обзор которых изложен в данной статье.

2. Накопители данных на флэш-памяти

Устройства хранения данных на основе флэш-памяти появились в конце 80-ых годов прошлого столетия [3]. Из-за дороговизны и несовершенства технологии производства твердотельные накопители долгое время не могли конкурировать с устройствами на магнитных дисках. Стоимость флэш-памяти в пересчете на 1 Гбайт по-прежнему на порядок выше, чем у дисковых устройств, однако, благодаря высокой скорости обмена и доступа к информации твердотельные накопители получают все большее распространение в современных системах хранения и обработки данных. Сам твердотельный накопитель представляет собой набор модулей флэш-памяти под управлением контроллера устройства.

Устройство флэш-памяти

Флэш-память – разновидность полупроводниковой технологии энергонезависимой электрически перепрограммируемой памяти. Ячейкой хранения данных является транзистор с плавающим затвором [4]. Подобный транзистор может хранить в себе заряд разной величины, поэтому в одной ячейке можно уместить несколько бит информации [5]. Из-за постепенной деградации полупроводника ячейка флэш-памяти обладает ограниченным ресурсом. Для современных накопителей количество циклов перезаписи составляет в среднем 5000.

По структуре компоновки элементов памяти в чипе различают NOR(Not OR) и NAND(Not AND) память. Конструкция NOR позволяет управлять состоянием каждой из ячеек в отдельности, в то время как NAND позволяет управлять только блоками данных. Типичный размер блоков чтения и записи 4 Кбайт, а удаления – 128 Кбайт [6]. Так как NAND память значительно дешевле, именно она используется в современных флэш-носителях.

Особенности работы твердотельных накопителей данных

Так как ячейки флэш-памяти подвержены износу, контроллеры твердотельных накопителей выполняют ряд вспомогательных операций, которые непосредственно не связаны с вводом-выводом, а служат лишь для продления времени работы устройств.

Основной механизм, который используется для этих целей, перемешивание содержимого диска [7]. Если в магнитных дисках логические блоки данных имеют постоянное во времени отображение на физические блоки, то для SSD подобное отображение может изменяться во времени. Так логические блоки часто записываемых файлов с каждой новой записью на накопитель записываются на новые физические блоки. В случае если новый блок также содержит полезные данные, они в свою очередь переписываются на другую позицию согласно алгоритмам контроллера. Таким образом, когда-либо записанный в последовательные физические блоки большой файл, к которому обращения происходят редко, со временем может “расползтись” на разрозненные блоки.

Отдельно стоит отметить, что перед записью данных в ячейки памяти, данные из них нужно удалить, что вызывает трудности, связанные с различными размерами блоков записи и удаления для модулей NAND памяти.

Рассмотрим последовательность действий, которые необходимо выполнить контроллеру при обращении на запись данных:

1. считывание блока, содержащего модифицируемые данные во внутренний буфер;
2. модифицирование необходимых байтов;
3. считывание полезных данных, оставшихся в данном блоке удаления;
4. стирание блока в микросхеме флэш-памяти;
5. вычисление нового местоположения блока согласно механизму перемешивания данных;
6. считывание и удаление данных из нового блока, если новый блок не пуст;
7. запись измененного блока, а также всей затронутой полезной информации на новые позиции.

Выполнение всех этих операций приводит к асимметрии чтения и записи данных.

Для того чтобы контроллеру не приходилось выполнять шаг 6, необходимо иметь информацию о том, какие блоки данных не содержат полезной информации. Для этого достаточно запоминать удаленные и помнить ни разу не задействованные области устройства. Сложность данной задачи связана с тем, что при работе с дисковыми устройствами в операции реального удаления данных не было никакой необходимости, и такая команда отсутствовала в интерфейсе обмена данными ОС с накопителями информации. В связи с её острой необходимостью для флэш-носителей, эту команду включили в современные интерфейсы обмена данными и она получила название TRIM.

Так как блоки удаления больше, чем блоки записи, контроллер зачастую вынужден записывать не только один изменившийся блок данных, но и затронутые удалением данные из смежных блоков. Данный эффект усиления записи приводит к ускоренному износу устройства.

Технические характеристики SSD

Рассмотрим сводную таблицу по техническим характеристикам, которая демонстрирует преимущества устройств на флэш-памяти, по сравнению с устройствами на магнитных дисках.

Характеристика	HDD	SSD(NAND) [8]
Время доступа на чтение, мс	20	0,1
Время доступа на запись, мс	20	0,2
Скорость последовательного чтения, Мбайт/сек	140	480
Скорость последовательной записи, Мбайт/сек	135	240
Произвольное чтение блока в 8 Кб, DQ 32, IOPS	150	9000
Произвольная запись блока в 8 Кб, DQ 32, IOPS	150	4000

Глубина очереди ввода/вывода важна и для твердотельных накопителей, так как обеспечивает большую гибкость при записи данных, а также позволяет осуществлять параллельный и асинхронный доступ к отдельным блокам памяти.

Из-за применения более сложных алгоритмов и использованию буферизации современные твердотельные накопители демонстрируют хорошую скорость записи данных, которая лишь вдвое уступает скорости чтения. Из-за применяемых оптимизаций скорость записи может зависеть как от текущей наполненности накопителя, так и от истории предыдущих обращений к устройству.

3. Буферный пул СУБД

Внешние устройства хранения данных работают на несколько порядков медленнее, чем оперативная память. Для снижения времени, которое тратится на обмен данными с внешними накопителями, СУБД поддерживает определенный набор часто используемых данных в буферном пуле. Буферный пул представляет собой область оперативной памяти, которая находится под полным контролем СУБД, т.е. она не может быть выгружена во внешнюю память операционной системой.

Страничная организация хранения данных в БД

Наибольшее распространение получила страничная организация памяти. Место во внешней памяти, предоставляемое для размещения информации в базе данных, логически разделяется на страницы фиксированного размера [9]. Каждая страница данных обычно содержит множество записей лежащих в одной таблице либо узлы существующих индексов. В большинстве систем обработку данных на уровне страниц осуществляет операционная система, а обработку записей внутри страницы ведет СУБД. Такая структура хранения данных обеспечивает эффективное хранение и доступ к записям с учетом всех ограничений, которые накладывают механизмы СУБД.

Большинство современных баз данных используют страницы размером от 2 до 16 килобайт. К примеру, в PostgreSQL страницы имеют размер 8 килобайт. На выбор размера страницы влияют множество факторов. Для увеличения эффективности работы с внешними устройствами требуется использовать страницы как можно большего размера. С другой стороны, для предотвращения косвенных блокировок записей, которые могут возникнуть при блокировке страницы на чтение или запись, необходимо стремится к уменьшению размера страницы.

Страничная организация памяти позволяет рассматривать задачу кэширования дисковых данных в виде кэширования определенного набора страниц БД.

Доступ к страницам данных через буфер

Доступ к страницам БД через буферный пул осуществляется в 2 этапа. Вначале идет проверка на наличие страницы в буферном пуле, а только после этого идет реальное обращение к странице на внешнем носителе. Страница, к которой выполняется обращение СУБД, должна быть обязательно помещена в оперативную память, так как работа со страницами во внешней памяти не допускается. В буферном пуле также могут храниться страницы, содержимое которых отличается от первоначального, такие страницы называются грязными. Они возникают в том случае, если страница сразу запрашивалась на запись, а также в том случае, если первоначально запрошенная на чтение страница подверглась изменениям в последствии.

Если размер буферного пула превысил выделенное под него место, для вставки новой страницы необходимо выбрать страницу-жертву, которая будет

удалена из пула. Если выбранная страница помечена грязной, перед удалением из буфера её необходимо записать на внешний носитель.

Базовые принципы замещения страниц

Для уменьшения времени работы с флэш-носителем можно выделить следующие принципы построения буферного пула СУБД [10].

1. Минимизация операций записи.
2. Использование страниц как можно большего размера для увеличения скорости чтения/записи с устройства.
3. Сокращение общего количества обращений к накопителю, за счет высокого процента попаданий в буфер.
4. Преобразование последовательности обращений на запись к флэш-устройству для уменьшения коэффициента усиления записи и ускорения выполнения операций.

Данные принципы невозможно рассматривать изолированно друг от друга, а также не принимая во внимание ограничения со стороны базовой функциональности СУБД.

Для уменьшения количества записей во внешнюю память можно вытеснять из буфера в первую очередь чистые страницы, но при этом будет возрастать процент промахов буфера.

Использование страниц большего размера уменьшает эффективность кэширования данных в целом. При обращении к записям, кэшируются не только они, но и все те записи, что находятся в той же странице. Так как побочные записи могут быть не нужны, оперативная память на них используется напрасно.

Страницы больших размеров также могут негативно сказаться на скорости обработки транзакций. Логическая синхронизация доступа к данным, которая используется при сериализации транзакций, не обеспечивает физической синхронизации доступа к страницам буферного пула. Одновременное изменение страницы данных различными транзакциями может привести к потере информации. В связи с этим, для обеспечения корректного доступа к страницам данных используется механизм блокировок. Перед каждым обращением к записи, содержащейся в странице, сама страница блокируется на чтение или запись. При блокировке страницы на запись её содержимое не может быть ни прочитано, ни изменено другими транзакциями. Увеличение размеров страниц приведет к увеличению количества взаимных блокировок транзакций.

Для ускорения выполнения обращений к носителю на запись, необходимо массово выталкивать страницы [11], расположенные в одном записываемом или удаляемом блоке данных флэш-памяти. Так как точно определить физическое расположение данных невозможно, используется предположение о том, что логически подряд расположенные страницы, имеют близкие

физические адреса. Техника совместного выталкивания близко расположенных страниц получила название кластерной записи, а непересекающиеся наборы таких страниц – кластерами. Стоит отметить, что использование данной техники напрямую способствует увеличению физической локальности логически близких данных, ввиду особенностей работы флэш-памяти.

Современные механизмы сериализации транзакций позволяют выталкивать на внешний носитель грязные страницы ещё не завершенных транзакций, что делает процесс кэширования грязных страниц независимым от основных механизмов СУБД.

Для увеличения процента попаданий в буферный пул возможно использование техники предварительного чтения страниц данных. Использование данной методики также может привести к ускорению работы с флэш-носителями. Однако, из-за того что скорость произвольного доступа к данным у флэш-устройств на несколько порядков выше, чем у дисковых накопителей, а точность угадывания страниц, которые могут пригодиться впоследствии, не высока, от использования данной техники можно отказаться.

4. Алгоритмы управления буферным пулом, учитывающие особенности флэш-памяти

Рассмотрим три современных алгоритма управления буферным пулом СУБД, оптимизированных для работы в системах с флэш-памятью [12].

Алгоритм CFDC(Clean First – Dirty Clustered)

Алгоритм CFDC [13] разбивает буфер на 2 области: W – рабочая область, содержащая горячие страницы, которые запрашиваются либо часто, либо в последнее время, I – приоритетная область, из которой выбираются страницы для выталкивания на внешне устройства.

Параметр λ регулирует долю страниц буферного пула, которые используются приоритетной областью P. Таким образом, если в буфере b страниц, то P содержит λ*b страниц, а оставшиеся (1- λ)*b находятся под управлением W. Отметим, что на область W не накладывается особых требований по управлению страницами, в ней могут находится как чистые, так и грязные страницы, а главное её назначение – предотвращать попадание наболее часто используемых страниц в приоритетную область.

Перемещение страниц из одной области буфера в другую:

1. При попадании в область W, необходимо отметить факт обращения к странице.
2. При попадании в область P, страница переносится в область горячих страниц.
3. В случае добавления новой страницы в область W, из неё в область P выталкивается страница-жертва(min(W)).
4. Если произошел промах буфера, то происходит выталкивание одной из страниц на внешний носитель. При этом страница-жертва всегда выбирается из области P, а свежая страница помещается в область W.
5. Выталкивание страницы-жертвы из буфера.

Более формально процесс обращения к страницам можно представить следующим алгоритмом.

Algorithm 1 CFDC

```
function REQUEST(Page p)
    * Переменные окружения:
    * Буфер Buffer, рабочая область W, приоритетная область P,
    * |W| = (1 − λ) · b ∧ |P| = λ · b.

    if p ∈ Buffer then
        if p ∈ W then
            Отметить попадание p в W;
        else
            Вытолкнуть min(W) в P, переместить p в W;
        end if
    else
```

```
        Страница q ← SelectVictimPr();
        if q is Null then
            q ← выбрать страницу-жертву из W;
        end if
        if q is Dirty then
            Записать q на внешний носитель;
        end if
        очистить q, переписать содержимое p из внещней памяти в q;
        p ← q;
        if p ∈ P then
            Вытолкнуть min(W) в P, переместить p в W;
        end if
    end if
    Return p;
end function
```

Рассмотрим механизм определения страницы жертвы, который используется в данном алгоритме. Приоритетная область P содержит 3 структуры: LRU список L чистых страниц, очередь с приоритетом Q, содержащая кластеры с грязными страницами, хэш-таблица для доступа к страницам кластеров.

Для обеспечения оптимальной работы запросов к внешнему устройству на запись необходимо придерживаться принципа локальности данных при выталкивании страниц. В качестве первого признака, указывающего на локальность данных, принимается близость логических адресов. Весь диапазон номеров страниц, разбивается на непересекающиеся интервалы равной длины, каждому из отрезков соответствует определенный кластер. При этом кластер может содержать только страницы из соответствующего диапазона. Для кластеров размером 4 страницы, получаем следующее разбиение.

Кластер	1				2				…	K+1				…
Стра-ницы	1	2	3	4	5	6	7	8	…	4*k	4*k+1	4*k+2	4*k+3	…

Вторым признаком локальности данных считается порядок попадания страниц в кластер. К примеру, если во второй кластер страницы попали в порядке {5,6,7,8}, он будет приоритетнее для выталкивания, чем первый кластер с порядком {3,1,4,2}.

Так как кластеры могут содержать произвольное число страниц(от 0 до 4), то с точки зрения эффективности записи на внешние носители, большие кластеры выталкивать эффективней. Также необходимо избегать долгосрочного хранения в буфере грязных страниц.

На основании этих факторов для кластера c, содержащего n страниц, используется следующая оценочная функция

$$P(c) = \frac{\sum_{i=1}^{n-1} |p_i - p_{i-1}|}{n^2 \times (globaltime - timestamp(c))},$$

где $p_0, \dots, p_{n-1}$ $p_0, \dots, p_{n-1}$ – порядок попадания страниц в кластер, globaltime – счетчик времени, отсчитывающий события выставки грязных страниц в буфер, timestamp(c) – значение globaltime в момент создания кластера. Сумма в числителе обозначается как IPD – inter-page distance.

Выталкивание страниц осуществляется по следующему алгоритму.

Algorithm 2 CFDC select victim

```
function SelectVictimPr
    * Переменные окружения:
    * Приоритетная область P, содержащая LRU список чистых
    * страниц L, а также очередь с кластерами грязных страниц
    * с их приоритетами Q.

    if L ≠ ∅ then
        v ← LRU страница из L;
    end if
    if v = null then
        cluster c ← кластер с наименьшим приоритетом из Q;
        if c ≠ null then
            v ← LRU страница из c;
            if v ≠ null then
                c.ipd ← 0;
            end if
        end if
    end if
    Return p;
end function
```

Так как операции с чистыми страницами выполняются значительно быстрей, страница-жертва в первую очередь выбирается их списка L. Если L – пуст, то выбирается кластер с наименьшим приоритетом. Следует отметить, что не весь выбранный кластер выталкивается на диск, а лишь одна LRU страница из него. Так как при этом у кластера сбрасывается IPD, все последующие выталкиваемые страницы будут выбраны из этого же кластера.

Разбиение буфера на 2 части также позволяет сэкономить процессорное время, которое тратится на выбор страниц жертвы, так как в этих расчетах не участвуют страницы из рабочей области.

Как видно, данный алгоритм стремится в первую очередь выталкивать из буфера чистые страницы, сохраняя только те из них, которые содержатся в области горячих страниц W. Данный подход эффективен, если запись

осуществляется значительно медленнее чтения, однако, оценить потери от снижения общего процента попаданий в буфер при таком подходе невозможно. Также недостатком данного алгоритма является негибкая политика по разделению буфера на 2 части.

Исследования эффективности данного алгоритма, проведенные его авторами, показывают 25% уменьшение времени исполнения TPC-C тестов, относительно алгоритма LRU, при снижении количества обращений на запись к носителю на 20%. Наилучшие показатели данный алгоритм демонстрирует при значении параметра $\lambda = 75\%$. Также эффективность данного алгоритма возрастает при использовании флэш-устройств, у которых более высокая разница скоростей чтения и записи.

Алгоритм CASA(Cost-Aware Self-Adaptive)

В отличие от алгоритма CFDC, CASA [13] предполагает динамический расчёт времени выполнения операция чтения и записи. Данный алгоритм разбивает буфер на 2 LRU списка. В одном из них хранятся чистые страницы, в другом грязные. Общая схема работы:

Параметр $\tau \in \mathbb{R}$ $\tau \in R$ определяет количество страниц, которое отводится для хранения чистых страниц. Пусть c_R и c_W нормализованные стоимости операция чтения и записи, то есть $(c_R + c_w = 1) \wedge (c_R \div c_w = \frac{read\ time}{write\ time})$

$(c_R + c_w = 1) \wedge (c_R \div c_w = \frac{readtime}{writetime})$Для определения τ используется следующий эвристический подход. При каждом попадании в буфер алгоритм пытается увеличить размер той области, в которой была найдена искомая страница. Параметр τ при этом увеличивается пропорционально нормализованной стоимости запроса, а также относительному размеру расширяемой области.

Существует только 2 случая, при которых проводится настройка параметра τ.

1. Попадание запроса на чтение в область C (CASA. строка 11).
2. Попадание запроса на запись в область D (CASA. строка 15).

В первом случае τ увеличивается на величину $c_R \times (|D| \div |C|)$, во втором – уменьшается на $c_W \times (|C| \div |D|)$. Ниже представлен алгоритм доступа к страницам буфера.

Algorithm 1 CASA

function REQUEST(Page p, Operation op)
    * **Переменные окружения:**
    * Буфер $Buffer$, емкостью b страниц; список свободных страниц E;
    * два LRU списка C - для чистых, D - для грязных страниц;
    * τ - размер списка C, где $0 \leq \tau \leq b$; нормализованые стоимости
    * чтения и записи c_R и c_W: $c_R + c_W = 1 \wedge c_R \div c_W = \frac{read\ time}{write\ time}$.
    * Изначально $|C| = 0; |D| = 0; |E| = b; \tau = 0$.

    if $p \in Buffer$ **then**
        if $p \in C \wedge op = R$ **then**
            $\tau \leftarrow min(\tau + c_R \times (|D| \div |C|), b)$;
            Переместить p на MRU позицию C;
        else if $p \in C \wedge op = W$ **then**
            Переместить p на MRU позицию D;
        else if $p \in D \wedge op = W$ **then**
            $\tau \leftarrow max(\tau - c_W \times (|C| \div |D|), 0)$;
            Переместить p на MRU позицию D;
        else
            * $p \in D \wedge op = R$
            Переместить p на MRU позицию D;
        end if
    else
        страница-жертва $v \leftarrow null$;
        if $|E| > 0$ **then**
            $v \leftarrow$ any from E;
        else if $|C| > \tau$ **then**
            $v \leftarrow$ LRU страница C;
        else
            $v \leftarrow$ LRU страница D;
            Записать страницу v на внешний носитель;
        end if
        Переписать содержимое p из внещней памяти в v;
        $p \leftarrow v$;
        if $op = R$ **then**
            Переместить p на MRU позицию C;
        else
            Переместить p на MRU позицию D;
        end if
    end if
    Return p;
end function

Данный алгоритм был усовершенствован внедрением кластерной записи грязных страниц. Полученный алгоритм получил название SAWC(Self Adaptive with Write Clustering). Сброс всего кластера, который содержит страницу-жертву, на внешнее устройство может привести к выталкиванию часто используемых страниц. Для избежания подобных ситуаций используется информация о частоте доступа к страницам данных. Для записи на накопитель из кластера выбираются только те страницы, которые использовались не чаще страницы-жертвы.

Кластерная запись описывается следующим алгоритмом.

Algorithm 2 CASA clustered

```
1: procedure ClusteredWrite(Page p)
2:     * Переменные окружения:
3:     * Хэш-таблица с кластерами грязных страниц T.
4:
5:     Записать страницу p на внешний носитель;
6:     кластер c ← кластер из T, содержавший p;
7:     if c exists then
8:         for all q ∈ c ∧ q ≠ p do
9:             if Freq(q) ≠ Freq(p) then
10:                Записать страницу q на внешний носитель;
11:                Удалить q из c;
12:            end if
13:        end for
14:        if |c| = 0 then
15:            Удалить c из T;
16:        end if
17:    end if
18: end procedure
```

Динамический расчет областей чистых и грязных страниц позволяет адаптироваться к текущей нагрузке на БД, увеличивая процент попаданий в буфер. Асимметрия чтения и записи учитывается при определении размеров областей. Так как алгоритм опирается на эвристический механизм разделения буфера, трудно теоретически оценить его эффективность. Также недостатком данного алгоритма является предположение о постоянной величине времени чтения и записи данных на флэш-носитель, которые, как известно, могут меняться в зависимости от текущей нагрузки на носитель и его заполненности.

Согласно исследованиям, которые проведены авторами данного алгоритма, при запуске TPC-C и TPC-H тестов произвольности он демонстрирует

снижение времени выполнения на 20%, относительно алгоритма LRU, и лишь немного уступает CFDC.

Алгоритм FD-Buffer

Данный алгоритм рассматривает задачу оптимизации работы буферного пула в терминах среднего времени доступа к странице. Пусть P_{total} вероятность промаха буфера, E_{dirty} доля грязных страниц среди всех выталкиваемых, а C_{read} и C_{write} время выполнения операций чтения и записи страниц. Тогда математическое ожидание среднего время доступа к странице можно выразить в следующем виде.

$$Cost_{io} = P_{total} \cdot C_{read} + P_{total} \cdot E_{dirty} \cdot C_{write}; (1)$$

Введем переменную $R = \frac{C_{write}}{C_{read}}$,

$\mathrm{R} = \frac{\mathrm{C_{write}}}{\mathrm{C_{read}}}$,которая отвечает за асимметрию операций чтения и записи, и преобразуем формулу 1 к нормализованному виду.

$$Cost'_{io} = P_{total}\big(1 + E_{dirty} \cdot R\big); (2)$$

Для подсчёта вероятности промаха используется метод, который предложил Мэтсон Р.Л. в 1970 году [14] для буферов, обладающих свойством вложенности. Буфер обладает свойством вложенности, если при увеличении размера буфера с M до M+K, новый буфер с 1-ой по M-ую позицию будет полностью аналогичен первоначальному.

Для оценки вероятности промаха буфера вводятся специальные счетчики попаданий $\mathrm{Hit}[1, \dots, \infty]$. При каждом обращении к буферу определяется глубина, при которой $Hit[1, \dots, \infty]$ запрашиваемая страница нашлась бы в пуле. Счетчик, соответствующий данной глубине, увеличивается на единицу. Очевидно, что максимальная глубина пула определяется общим количеством страниц данных БД, но поддерживать информацию обо всех страницах нерационально. Предположим, что мы ограничились поиском только до глубины равной N. Если требуемая глубина пула будет превышать это значение, необходимо увеличивать специальный счетчик $Hit[\infty]\mathrm{Hit}[\infty]$. Обладая такой информацией о попаданиях в пул, можно рассчитать вероятность промаха для буферного пула меньшего размера, чем N.

$$P(M) = 1 - \frac{\sum_{i=1}^{M} Hit[i]}{\sum_{i=1}^{N} Hit[i] + Hit[\infty]}; M \le N; (3)$$

Для поддержки механизма обновления счетчиков достаточно поддерживать виртуальный буфер размера N, а в памяти хранить только те страницы, которые попали на первые M позиций.

Отличие алгоритма FD-Buffer [15] от представленной модели в том, что он поддерживает 2 LRU стека страниц, в одном из них находятся чистые, а в другом грязные страницы. Память между стеками распределяется исходя из принципа минимизации среднего времени обращения к устройству(формула 2). Для оценки вероятности промаха для каждого стека поддерживаются счетчики попаданий, а сама вероятность оценивается с помощью формул, аналогичных формуле 3.

Обозначим стек с чистыми страницами *C*, а с грязными *D*, а их размеры M_c и M_d. Для двух стеков алгоритм доступа к страницам имеет следующий вид.

Algorithm 1 Reads and Writes on FD-Buffer

```
1:  function READ(Page p)
2:      Frame v = C.Lookup(p);
3:      if v ≠ NULL then
4:          C.Update(v);
5:      else
6:          v = D.Lookup(p);
7:          if v ≠ NULL then
8:              D.Update(v);
9:          else
10:             v = FindVictimForClean(C, D);          ▷ Algorithm 2.
11:             Загрузить страницу p из носителя в буфер v;
12:         end if
13:     end if
14:     Return frame v;
15: end function

16: function WRITE(Page p)
17:     Frame v = C.Lookup(p);
18:     if v ≠ NULL then
19:         C.Remove(v);
20:         D.Add(v);
21:     else
22:         v = D.Lookup(p);
23:         if v ≠ NULL then
24:             D.Update(v);
25:         else
26:             v = FindVictimForDirty(C, D);          ▷ Algorithm 2.
27:             Загрузить страницу p из носителя в буфер v;
28:         end if
29:     end if
30:     Return frame v;
31: end function
```

Так как размер каждой части буфера определяется динамически, страница жертва выбирается в первую очередь из того стека, размер которого превышает рассчитанный оптимум. В случае если рассчитанный размер не соответствует реальному, помимо выталкивания страниц происходит передача освободившегося места другому буферу.

Для поиска выталкиваемой страницы используются следующие алгоритмы.

Algorithm 2 Page evictions in FD-Buffer

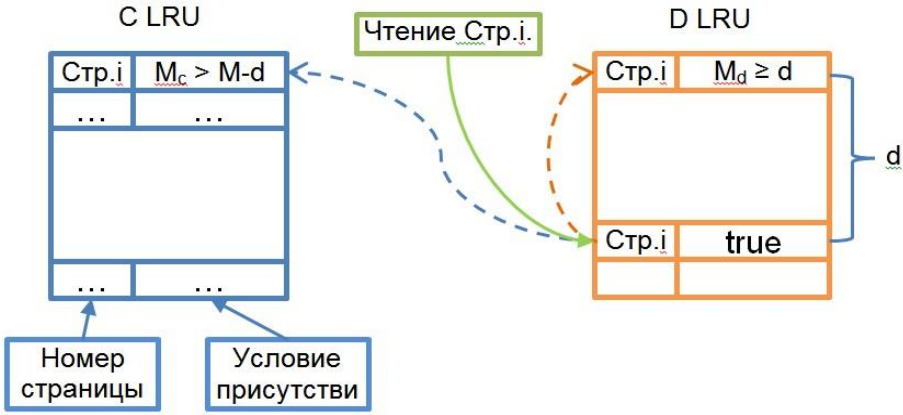
```
1:  function FindVictimForClean(C, D)
2:      if |D| > M_d then
3:          v = D.GetVictim();
4:          D.Remove(v);
5:          C.Add(v);
6:          Записать страницу из v на внешний носитель;
7:      else
8:          v = C.GetVictim();
9:      end if
10:     Return frame v;
11: end function

12: function FindVictimForDirty(C, D)
13:     if |C| > M_c then
14:         v = C.GetVictim();
15:         C.Remove(v);
16:         D.Add(v);
17:     else
18:         v = D.GetVictim();
19:         Записать страницу из v на внешний носитель;
20:     end if
21:     Return frame v;
22: end function
```

Использование 2 стеков сильно усложняет механизм подсчета количества попаданий в пул. Каждый запрос на чтение и запись может попасть как в стек чистых, так и грязных страниц. Обозначим вероятности попаданий в них как $\mathrm{P_c}$ и $\mathrm{P_d}$. Тогда для вероятности промаха в целом получим $P_{total} = P_C \times P_D$.$\mathrm{P_{total}} = \mathrm{P_C} \times \mathrm{P_D}$. Доля грязных страниц, среди всех вытолкнутых, это вероятность промаха по записи в стеке грязных страниц, т.е. $E_{dirty} = P_d^w \mathrm{E_{dirty}} = \mathrm{P_d^w}$. Преобразуем нормали-зованную стоимость обращения к буферу к следующему виду.

$$Cost'_{io} = P_c \times P_d(1 + P_d^w \cdot R); (4)$$

Как было отмечено ранее, для поддержки механизма подсчета промахов необходимо эмулировать стек большего размера, чем под него реально

выделено места. Так как используется 2 стека, то при перемещении страниц между ними возможна ситуация, когда страница окажется сразу в обоих стеках. Рассмотрим одну из них подробней. Расширим описатель буфера полем с условием присутствия странице в стеке, его смысл будет раскрыт далее.



При попытке доступа к странице на чтение, она может находиться в виртуальном стеке D на глубине d. Классическое поведение предполагает перемещение страницы на вершину стека D, однако, если эта страница реально в нем не находится, то она перемещается на вершину стека C. Так как принятое решение повлияет на будущее увеличение счетчиков попадания в буфер, для расчета оптимального размера стеков необходимо запомнить условие, которое определило местонахождение страницы. Для этого в виртуальных стеках запоминаются оба варианта перемещения страницы и запоминаются условия её попадания в них, при этом реально страница хранится только в одном из них.

Проверку на условие наличия страницы в буфере обозначим PCond. Для поддержки счетчиков попаданий в буфер операции чтения и записи модифицируются следующим образом.

Чтение:

Algorithm 3 Handling a read in Two-Stack

```
1: function ReadHandler(Pagep)
2:     if (p ∉ CLRU) ∧ (p ∉ DLRU) then
3:         Hit_c^r[∞] + +, Hit_d^r[∞] + +;
4:         Поместить p на вершину CLRU в буфер e_c;
5:         PCond_c[e_c] = true;
6:     end if
7:     if (p ∈ CLRU − буфер e_c) ∧ (p ∉ DLRU) then
8:         * Пусть d_c глубина вхождения e_c в стек CLRU;
9:         * Рассчитываем такую глубину стека i, чтобы при M_c ≥ i
10:        * выполнялось (M_c ≥ d_c) ∧ PCond_c[e_c];
11:        Hit_c^r[i] + +, Hit_d^r[∞] + +;
12:        Поместить содержимое e_c на вершину CLRU в буфер e'_c;
13:        PCond_c[e'_c] = true;
14:    end if
15:    if (p ∉ CLRU) ∧ (p ∈ DLRU − буфер e_d) then
16:        * Пусть d_d глубина вхождения e_d в стек DLRU;
17:        * Рассчитываем такую глубину стека j, чтобы при M_d ≥ j
18:        * выполнялось (M_d ≥ d_d) ∧ PCond_d[e_d];
19:        Hit_c^r[∞] + +, Hit_d^r[j] + +;
20:        Поместить содержимое e_d на вершину DLRU в буфер e'_d;
21:        PCond_d[e'_d] = (M_d ≥ d_d);
22:        Копировать содержимое e_d на вершину CLRU в буфер e_c;
23:        PCond_c[e_c] = (M_c > M − d_d);
24:    end if
25:    if (p ∈ CLRU − буфер e_c) ∧ (p ∈ DLRU − буфер e_d) then
26:        * Пусть d_c и d_d глубина вхождения e_c и e_d в CLRU and DLRU;
27:        * Рассчитываем такую глубину стека i, чтобы при M_c ≥ i
28:        * выполнялось (M_c ≥ d_c) ∧ PCond_c[e_c];
29:        * Рассчитываем такую глубину стека j, чтобы при M_d ≥ j
30:        * выполнялось (M_d ≥ d_d) ∧ PCond_d[e_d];
31:        Hit_c^r[i] + +, Hit_d^r[j] + +;
32:        Поместить содержимое e_c на вершину CLRU в буфер e'_c;
33:        PCond_d[e'_c] = (PCond_c[e_c] ∨ M_c > M − d_d);
34:        Поместить содержимое e_d на вершину DLRU в буфер e'_d;
35:        PCond_d[e'_d] = (PCond_d[e_d] ∨ M_d ≥ d_d);
36:    end if
37: end function
```

Запись:

Algorithm 3 Handling a write in Two-Stack

function WRITEHANDLER(Page p)
  if $p \in$ CLRU - буфер e_c **then**
    * Пусть d_c глубина вхождения e_c в стек CLRU;s
    * Рассчитываем такую глубину стека i, чтобы при $M_c \geq i$
    * выполнялось $(M_c \geq d_c) \wedge PCond_c[e_c]$;
    $Hit_c^w[i] + +$;
    Удалить буфер e_c из CLRU;
  end if
  if $p \notin$ CLRU **then**
    $Hit_c^w[\infty] + +$;
  end if
  if p in DLRU at entry e_d **then**
    * Пусть d_d глубина вхождения e_d в стек DLRU;
    * Рассчитываем такую глубину стека j, чтобы при $M_d \geq j$
    * выполнялось $(M_d \geq d_d) \wedge PCond_d[e_d]$;
    $Hit_d^w[i] + +$;
    Поместить содержимое e_d на вершину DLRU в буфер e'_d;
    $PCond_d[e'_d] = true$;
  end if
  if $p \notin$ DLRU **then**
    $Hit_d^w[\infty] + +$;
    Поместить p на вершину DLRU в буфер e''_d;
    $PCond_d[e''_d] = true$;
  end if
end function

Формулы определения вероятности промахов стеков для подсчета оптимальных размеров областей буфера можно выразить следующим образом.

$$P_c(M_c) = 1 - \frac{\sum_{i=1}^{M_c}(Hit_c^r[i] + Hit_c^w[i])}{\sum_{i=1}^{n}(Hit_c^r[i] + Hit_c^w[i]) + Hit_c^r[\infty] + Hit_c^w[\infty]}; (5)$$

$$P_d(M_d) = 1 - \frac{\sum_{i=1}^{M_d}(Hit_d^r[i] + Hit_d^w[i])}{\sum_{i=1}^{n}(Hit_d^r[i] + Hit_d^w[i]) + Hit_d^r[\infty] + Hit_d^w[\infty]}; (6)$$

$$P_d^w(M_d) = 1 - \frac{\sum_{i=1}^{M_d} Hit_d^w[i]}{\sum_{i=1}^{n} Hit_d^w[i] + Hit_d^w[\infty]}; (7)$$

Так как определение M_C и M_D в режиме реального времени требует много вычислительных ресурсов, пересчет оптимальных размеров стеков выполняется периодически, один раз за несколько обращений к буферу.

Приведенный алгоритм позволяет достаточно точно решать задачу оптимизации среднего времени доступа к странице, поддерживая баланс между различием в стоимости операций чтения и записи страниц данных и высоким процентом попадания в пул. Наличие двух виртуальных буферов, счетчиков попаданий, а также сложных формул расчета вероятности промахов буфера, приводит к большим вычислительным затратам, что может оказаться значительным ограничением в его использовании. Важным дополнением к алгоритму FD-Buffer является внедрение методики кластерной записи.

Для оценки эффективности приведенного алгоритма авторы предлагают рассчитывать среднее время доступа к страницам БД. Так при выполнении теста TPC-C среднее время доступа к страницам снижалось на 30%, по сравнению с алгоритмом LRU. Согласно экспериментальным данным, алгоритм FD-Buffer лишь немного опережает CFDC по производительности и только при условии внедрения кластерной записи.

5. Заключение

Накопители данных на флэш-памяти имеют ряд серьезных преимуществ перед накопителями на жестких дисках, поэтому системы работающие с флэш-памятью демонстрируют большую производительность. Технологическое удешевление стоимости подобных устройств и увеличение плотности хранения информации приводит к некоторым особенностям работы, например к асимметрии скоростей чтения и записи данных. Данные особенности приводят к снижению эффективности буферизации страниц данных – важнейшего механизма повышения скорости работы СУБД. В данной работе рассмотрены три современных алгоритма управления буферным пулом, которые оптимизированы для работы с флэш-памятью. Экспериментальные исследования алгоритмов CFDC и FD-Buffer на тестах производительности TPC-C демонстрируют преимущество до 30% перед классическим алгоритмом LRU. Основной вклад в повышение эффективности вносит использование кластерной записи страниц, при которой грязные страницы выталкиваются из буфера с учетом принципа локальности. К недостаткам представленных алгоритмов можно отнести более высокие вычислительные затраты и потребление памяти. В целом, представленные алгоритмы позволяют использовать преимущества флэш-памяти без потерь производительности связанных с особенностями её работы.

Список литературы

[1] Belady. L., «A study of replacement algorithms for a virtual-storage computer,» *IBM Systems Journal*, p. 78–101, 1966.

[2] Shaul Dar, Michael J. Franklin, Bjorn.T. Jonsson, «Semantic Data Caching and Replacement,» в *VLDB Conference*, 1996.

[3] Masuoka Fujio, Iizuka Hisakazu,, «Semiconductor memory device and method for manufacturing the same,» 1985.

[4] INTEGRATED CIRCUIT ENGINEERING CORP., «Flash Memory Technology,» [В Интернете]. Available: http://smithsonianchips.si.edu/ice/cd/MEMORY97/SEC10.PDF.
[5] Vättö Kristian, «Understanding TLC NAND,» 2012. [В Интернете]. Available: http://www.anandtech.com/show/5067/understanding-tlc-nand/2.
[6] Jesung K., Min J., Sam H., Sang L., Yookun C.;, «A Space-Efficient Flash Translation Layer for CompactFlash Systems,» *Proceedings of the IEEE 48 (2),* p. 366–375, 2002.
[7] Perdue Ken, «"Wear Leveling Application Note,» 2010.
[8] OCZ Technology Group, Inc., «OCZ Vertex 3 SSD Series – 3-rd Generation Harnessing the speed of the SATA III interface.,» 2011.
[9] С.Д. Кузнецов, Основы баз данных, 2007.
[10] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross,, «SSD Bufferpool Extensions for Database Systems.,» 2010.
[11] D. Seo, D. Shin. , « Recently-evicted-first buffer replacement policy,» *IEEE Transactions on Consumer,* p. 1228–1235, 2008.
[12] «Selected Papers on Flash-Based Database,» *Lab of Web and Mobile data Management,* 2011.
[13] Y. Ou, T. Härder, and P. Jin., «CFDC: a flash-aware replacement policy for database buffer management.,» в *DaMoN,* 2009.
[14] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger., «Evaluation techniques for storage hierarchies.,» *IBM System Journal 9(2),* 1970.
[15] Sai Tung On,Yinan Li,Bingsheng He, Ming Wu, Qiong Luo, Jianliang Xu., «FD-Buffer: A Buffer Manager for Databases on Flash Disks.,» 2010.

Flash-based algorithms of database buffer management

Kuznetsov S.D.
ISP RAS, Moscow, Russia
kuzloc@ispras.ru
Prokhorov A.A.
MIPT, Moscow, Russia
EmailToProkhorov@gmail.com

Abstract. One of the most important ways of increasing the speed of the modern databases is to cache frequently used data in RAM. Classical replacement policies are intended to minimize the number of buffer pool faults. This optimization method implicitly relies on the fact that the speeds of reading and writing to the hard disc are equal. Gradual technology improvement and cost reduction of flash memory have led to the creation of solid-state data storages (SSD) that are now increasingly used in personal computers and storage systems. Flash drives have advantages over traditional hard drives, high read and write speeds and significantly small time of random data access are the most important of them. However, the most popular flash-memory types read data at a higher speed than write it. Due to this feature the use of classical replacement algorithms of disk data caching is ineffective. This paper reviews recently developed algorithms of database buffer pool management designed to work with flash memory drives: CFDC (Clean First – Dirty Clustered), CASA (Cost-Aware Self-Adaptive), SAWC (Self Adaptive with Write Clustering), and FD-Buffer. Some of these algorithms demonstrate significant advantages over the classical algorithm LRU.

Keywords: Database buffer pool, disk data caching, flash memory, SSD

References

[1]. Belady. L., «A study of replacement algorithms for a virtual-storage computer,» IBM Systems Journal, p. 78–101, 1966.
[2]. Shaul Dar, Michael J. Franklin, Bjorn.T. Jonsson, «Semantic Data Caching and Replacement,» в VLDB Conference, 1996.
[3]. Masuoka Fujio, Iizuka Hisakazu,, «Semiconductor memory device and method for manufacturing the same,» 1985.
[4]. INTEGRATED CIRCUIT ENGINEERING CORP., «Flash Memory Technology,» [В Интернете]. Available: http://smithsonianchips.si.edu/ice/cd/MEMORY97/SEC10.PDF.
[5]. Vättö Kristian, «Understanding TLC NAND,» 2012. Available: http://www.anandtech.com/show/5067/understanding-tlc-nand/2.
[6]. Jesung K., Min J., Sam H., Sang L., Yookun C.;, «A Space-Efficient Flash Translation Layer for CompactFlash Systems,» Proceedings of the IEEE 48 (2), p. 366–375, 2002.
[7]. Perdue Ken, «"Wear Leveling Application Note,» 2010.
[8]. OCZ Technology Group, Inc., «OCZ Vertex 3 SSD Series – 3-rd Generation Harnessing the speed of the SATA III interface.,» 2011.
[9]. S.D. Kuznetsov, Osnovy baz dannyh [Foundations of databases], 2007 (in Russian).
[10]. Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross,, «SSD Bufferpool Extensions for Database Systems.,» 2010.

[11]. D. Seo, D. Shin. , « Recently-evicted-first buffer replacement policy,» IEEE Transactions on Consumer, p. 1228–1235, 2008.
[12]. «Selected Papers on Flash-Based Database,» Lab of Web and Mobile data Management, 2011.
[13]. Y. Ou, T. Härder, and P. Jin., «CFDC: a flash-aware replacement policy for database buffer management.,» в DaMoN, 2009.
[14]. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger., «Evaluation techniques for storage hierarchies.,» IBM System Journal 9(2), 1970.
[15]. Sai Tung On,Yinan Li,Bingsheng He, Ming Wu, Qiong Luo, Jianliang Xu., «FD-Buffer: A Buffer Manager for Databases on Flash Disks.,» 2010.

Обзор развития методов лексической оптимизации запросов

Мендкович Н. А., Кузнецов С. Д.,
mend@rambler.ru, kuzloc@ispras.ru

Аннотация. Статья посвящена лексической оптимизации запросов и описывает работы, опубликованные в течение последних четырех десятилетий. Особое внимание уделяется таким подходам к оптимизации как модификация, украшение и сокращение запросов. Обсуждаются алгоритмы оптимизации для реляционных и нереляционных СУБД.

Ключевые слова: оптимизация запросов; упрощение запросов; лексическая оптимизация запросов; магические множества

1. Введение

За 40 лет, прошедших с момента выхода первой работы, в которой была сформулирована проблема оптимизации запросов к базам данных [1], вышло множество публикаций, посвященных этой тематике. Полный их обзор, актуальный на настоящий момент, отсутствует, и некоторое представление об их численности может дать анализ библиографий вышедших на настоящий момент обзорных работ.

Так, исследование М. Ярке (M. Jarke) и Ю. Коха (J. Koch) включает в себя перечень 253 работ [2], М. Маннино (M. V. Mannino), П. Чу (P. Chu) и Т. Сейджера (T. Sager) – 64 работы) [3], Я. Ионнидиса (Y. E. Ionnidis) – 50 работ [4], С. Чаудхари (S. Chaudhari) – 61 работа [5], С. Д. Кузнецова – 128 работ [6]. Обзоры методов оптимизации запросов также содержатся в тематических главах работ Г. Грейфа (G. Graefe) [7, разделы 6, 11, 14], К. Дейта (C. Date) [8, стр. 639-690], М. Т. Оцсу (M. Ozsu) и П. Валдуреза (P. Valduriez) [9, pp. 228-273], Р. Рамакришнана (R. Ramakrishnan) и Дж. Герке (J. Gehrcke) [10, pp. 412-413]. Кроме этого, следует учесть краткую библиографию работ, посвященных оптимизации запросов, в которой содержится классификация этих работ по темам [11]. Множества рассмотренных исследований в перечисленных выше публикациях частично пересекаются, однако приведенные данные дают общее представление о публикациях, посвященных проблемам оптимизации запросов.

Наравне с академическими исследованиями, посвященными оптимизации запросов, параллельно выходят публикации компаний, посвященные решению

этой задачи в рамках выпускаемых ими СУБД (например, MySQL [12], [13, глава 19], Oracle [14-15, 92-93], DB2 [16], PostgreSQL [17]). Подробный обзор оптимизаторов СУБД различных производителей был представлен в [9], но позже подобные исследования не предпринимались, во многом по причине неготовности компаний подробно описать собственные разработки в области оптимизации запросов. Исключениями могли бы служить СУБД с открытым кодом, в частности, PostgreSQL, но публикации с описанием использованных в них алгоритмов нам неизвестны.

Как показывает представленная выше статистика публикаций в области оптимизации запросов, их полной обзор является нетривиальной задачей, и с высокой вероятностью в итоге такой обзор может быть неполным. Поэтому более разумным представляется сосредоточиться на одном из классов оптимизационных алгоритмов. Ранее публиковались подобные обзоры, посвященные применению гистограмм, в том числе, для целей оптимизации запросов [18], применению материализованных представлений в процессе обработки запросов [19], решению выбора порядка соединений [20].

Данная работа посвящена методам так называемой лексической оптимизации запросов, но чтобы определить, что собой представляют эти методы, необходимо представить краткий очерк основных этапов оптимизации и признаков, позволяющих отнести конкретный метод к одному из этих этапов.

Вслед за Я. Ионнидисом мы определим оптимизацию запросов как «изучение алгебраических выражений, эквивалентных заданному запросу, и выбор из них одного, оцениваемого как наиболее дешевое» [4, p. 1041].

На основе классификации, принятой в [2], в рамках задачи оптимизации запроса выделим четыре основных этапа, или класса оптимизирующих операций:

i преобразование запроса в стандартную внутреннюю форму;
ii модификация запроса, т.е. приведение его к форме, при которой выполнение наиболее эффективно;
iii анализ запроса, выбор потенциальных низкоуровневых процедур, используемых при его выполнении;
iv генерация и сравнительный анализ различных планов доступа – возможных способов выполнения запроса.

На практике четвертый класс операций воспринимается как основа оптимизационных алгоритмов; первые три класса используются для подготовки генерации планов доступа, в то время как на четвертом этапе оптимизатор использует эти планы, оценивает и выбирает лучший из них для выполнения запроса. Именно четвертый класс операций сегодня привлекает наибольшее внимание исследователей, и его описанию уделяется наибольшее внимание в корпоративных обзорах оптимизаторов выпускаемых СУБД. Так, например, оптимизатор СУБД компании Oracle, судя по доступным нам описаниям [21], ориентирован в первую очередь на сравнительный

«оценочный» анализ существующих путей доступа, и во многих обзорах, посвященных оптимизатору данной СУБД, методы лексической оптимизации не упоминаются вовсе [например, 22, с. 417-445]. В оптимизаторах Starburst и DB2 основное внимание уделяется семантической оптимизации запросов и совершенствованию системы выбора планов выполнения запросов [16].

Однако, несмотря на подчиненную роль, которую играют модификация и первичные преобразования запросов, рассмотрение наиболее эффективных подходов к решению данной задачи представляется вполне актуальным, так как оптимизатор по-прежнему нуждается в генерации достаточного числа субоптимальных, в рамках того или иного критерия, планов доступа.

При анализе подобных подходов к оптимизации запросов можно выделить два вида модификаций:

- *лексическая оптимизация*[1] запроса, при которой единственным источником информации является сам текст запроса как лексическая конструкция, и иные сведения о базе данных и ее структуре в анализе не используются. Процесс лексической оптимизации включает в себя анализ ограничения запроса, сравнение и анализ содержащихся в нем условий с целью выявления избыточности;
- *семантическая оптимизация*, включающая в себя использование ограничений целостности БД и сами хранимые данные с целью генерации оптимальных планов доступа к данным (обзор методов семантической оптимизации представлен в [2, глава 3], а ряд алгоритмов рассматривается в некоторых более поздних работах [23-24]).

Методы лексической оптимизации представляются особо интересными в связи с тем, что, во-первых, их можно легче адаптировать для нужд оптимизации запросов в базах данных с нереляционной структурой, во-вторых, в ряде существующих разработок предусматривается вынесение некоторых оптимизирующих операций из сервера базы данных в удаленное клиентское приложение [25]. Кроме того, поскольку наибольший интерес к исследованиям в данной области проявлялся в 1990-е и даже 1980-е гг., многие современные исследователи могут быть недостаточно знакомы с более ранними работами, алгоритмы и методы, представленные в которых, могут быть вновь востребованы в настоящее время.

[1] В зарубежной литературе для обозначения подобных преобразований часто используется термин «перезапись» (rewriting) [4, p. 1044]. Однако нам представляется, что этот термин в недостаточной степени выражает суть данного вида оптимизации запросов. Кроме того, в ряде работ он используется по отношению к преобразованиям запросов с использованием материализованных представлений запросов [19], т.е. выходит за рамки используемого определения.

С точки зрения логики, лексическая оптимизация может устранять три различных вида неоптимальности запроса тремя различными способами.

- Сокращение запроса – удаление избыточных условий, например, дублирующих одно другое или тождественно истинных, обработка которых является излишней с точки зрения выполнения запроса, а также условий, делающих запрос невыполнимым, противоречащих друг другу или тождественно ложных. При обнаружении таковых условий все ограничение или его часть признается ложной и исключается из выполнения запроса.
- Усовершенствование (украшение) – усложнение структуры запроса, которое, однако, ведет к оптимизации его выполнения.
- Преобразование запроса – изменение запроса, не относящееся ни к сокращениям, ни к усовершенствованиям запроса, и играющее роль, служебную по отношению к иным формам оптимизации запроса, например, стандартизация формы представления запроса.

Далее в статье приводятся классификация алгоритмов лексической оптимизации, обзор работ в данной области, а также отмечаются возможные направления дальнейших исследований.

Мы стремились свести к минимуму возможные повторы сведений, изложенных в предыдущих обзорных работах, поэтому в ряде случаев отказываемся от подробного анализа тех или иных вопросов, отсылая читателей к более ранним публикациям. Допускается краткое повторение некоторых сведений в ряде случаев с целью показать развитие исследований по совершенствованию методов оптимизации запросов.

2. Преобразования запросов

Критерием классификации алгоритма как относящегося к числу алгоритмов простого преобразования является его функциональное предназначение. Если итогом работы некоторого алгоритма должно стать обретение запросом более оптимальной формы с точки зрения того или иного критерия (например, числа условий в ограничении), то алгоритм может быть отнесен к числу алгоритмов сокращения или усовершенствования. Если же алгоритм, как упоминалось выше, играет подчиненную роль, преобразуя запрос к иному виду, в частности, меняя порядок операций, то речь идет именно о простом преобразовании запроса.

Прежде всего, к технологиям преобразования нужно отнести алгоритмы стандартизации запросов, приведения их к универсальной стандартной форме в целях дальнейшей обработки. Такие преобразования требуются не только для нужд дальнейшего улучшения или упрощения запроса, но также для сопоставления обрабатываемых запросов с ограничениями целостности базы данных и сохраняемыми материализованными представлениями запросов.

Возможные подходы к представлению запросов к реляционным базам данных в виде графов, табло, реляционных выражений и т.п. рассмотрены в [2, раздел 3.1], каковой источник, несмотря на давность, вполне актуален.

У представлений запросов в нереляционных СУБД имеется своя специфика. Основы представления запросов на языке XQuery уже описаны в рекомендациях W3C [26], однако ряд особенностей внутреннего представления запросов заслуживает отдельного рассмотрения. В интересах оптимизации запросов, формулируемых на новых языках, требуется создание новых алгебр с учетом особенностей моделей данных, лексики, синтаксиса и семантики этих языков.

Можно выделить два подхода к решению этой задачи: реляционно-центричный и алгебраический. Первый подход основывается на преобразовании запросов на нереляционных языках, а при необходимости и самих данных к реляционному виду и их представлению в терминах реляционных СУБД [27-30], чтобы иметь возможность полностью использовать для оптимизации новых видов запросов арсенал накопленных ранее средств. Обзор подобных работ представлен в [31].

Во втором подходе предлагается не приближать новые языки и модели данных к старым алгоритмам, а создавать новые алгебры для представления запросов с учетом особенностей новых языков. Проблема представления запросов к нереляционным базам данных решается с помощью расширения реляционной алгебры для соответствующих языков запросов: OQL [32-33] и XQuery [34-37]. Проблема алгебраического представления запросов к базам XML-данных также рассматривается в [38-39].

Примером преобразования запроса может также служить перемещение логических условий вниз в дереве запроса. Целью ряда улучшающих преобразований является минимизация размера конструируемых, сохраняемых и считываемых промежуточных результатов. Важная эвристика перемещает селективные операции, такие как ограничение и проекция, ниже конструктивных операций, таких как соединение и декартово произведение, чтобы выполнять селективные операции как можно раньше [2, 40]. В пользу как можно более раннего исполнения селективных операций высказывались и другие авторы [41]. Данный подход считается сейчас если и не оптимизирующим запрос, то, по крайней мере, не повышающим стоимость его обработки, и он автоматически применяется ко всем запросам. Также имеются работы, посвященные перемещению по дереву запроса операций сортировки [42] и предикатов (в тех случаях, когда запросы не имеют иерархической структуры) [43].

В ряде работ анализируются способы упорядочения операций проекции и соединения с целью упрощения обработки запросов (например, [44-47]). Однако наряду с некоторыми алгоритмами, которые можно отнести к лексическим, в этих подходах применяются методы, которые основываются на использовании описаний данных, содержащихся в базе, и поэтому мы не

имеем достаточных оснований останавливаться на таких подходах к оптимизации в настоящем обзоре. Более подробно существующие варианты решения задачи определения порядка соединений анализируются в [2, раздел 4], [5, раздел 4.1] и [20].

Наравне с подходом к оптимизации, основанным на упорядочении операций, имеется ряд работ, ориентированных на сокращение их числа; эти работы подробно рассматриваются в разделе 4 данной статьи.

Еще одну группу алгоритмов преобразования составляют алгоритмы устранения вложенных подзапросов, что для стандартных SQL-запросов означает исключение из блоков SELECT-FROM-WHERE вложенных в них аналогичных блоков путем их удаления или слияния с телом основного запроса. Приведем пример запроса о поиске всех поставщиков детали P2 из [8, с. 683]:

```
SELECT S.Name
FROM S
WHERE S.S# IN (SEL query optimization; query simplification; lexical query optimization; magic sets
ECT SP.S#
FROM SP
WHERE SP.P# = 'P2');
```

Его семантически эквивалентной, но более лаконичной и быстрой в обработке формулировкой является следующий запрос:

```
SELECT S.Name
FROM S, SP
WHERE S.S# =SP.S#
AND SP.P# = 'P2';
```

Впервые решение проблемы вложенных запросов было предложено в работе [48]. Автор выявил 5 типов вложенных запросов и описал алгоритмы устранения этих вложений. Позже работа [48] подвергалась критике с указанием на то, что в ряде случаев предложенные алгоритмы работают некорректно [49-51]. В этих работах предлагались свои варианты алгоритмов, в которых исправлялись недочеты, допущенные в [48], и обеспечивалась большая эффективность. Альтернативное решение задачи было недавно предложено в [52].

Поскольку проблема вложенных подзапросов существует и в нереляционных СУБД, в нескольких работах представляются алгоритмы исключения вложений из запросов, сформулированных на OQL [32-33] и XQuery [34-35, 53-54]. Важно отметить, что изучение этой проблемы исторически совпало с разработкой новых нереляционных алгебраических представлений запросов для этих языков.

Более подробно работы по проблеме вложенных подзапросов рассмотрены в [8, с. 682-684], [5, раздел 4.2.2] и [35, раздел 2]. Примеры применения алгоритмов устранения вложенности в различных системах описаны в [55-56].

3. Лексическая оптимизация путем улучшения

Оптимизация запроса путем улучшения основывается на усложнении структуры запроса, включении в него новых табличных выражений, использование которых позволяет сократить расходы на обработку исходного запроса. Большинство алгоритмов, решающих эту задачу, относятся к технологии «магических множеств».

Алгоритм магических множеств относится к числу алгоритмов лексической оптимизации запроса. Он означает перезапись запроса, исключающую генерацию нерелевантных кортежей путем вычисления дополнительных таблиц, которые содержат связывания, используемые для ограничения таблицы, и действуют как фильтры.

Перезапись исходного запроса с использованием этих фильтров включает следующие основные этапы:

- анализ связей в рамках реализации запроса, их отражение путем аннотирования предикатов;
- создание на основе полученных описаний дополнительных (магических) таблиц;
- интеграцию магических таблиц в уже существующее ограничение и изменение описания существующих условий.

Изначально алгоритм магических множеств был создан для нужд дедуктивных баз данных, поддерживающих язык запросов «Datalog»[57], где запрос на выборку данных описывается с помощью системы правил (rules) и фактов (facts).

Алгоритм магических множеств предлагает создать дополнительные правила и факты, которые усложнят запись правил системы, но позволят исключить из обработки индивидов, заведомо нерелевантных запросу. Для этого в систему правил включается отношение magic(x), обозначающее релевантность объекта по отношению к обрабатываемому запросу.

Предложенная перезапись увеличивает число правил и удлиняет их запись, однако очевидно, что скорость обработки запроса при этом возрастает, так как добавленные магические отношения исключают из анализа нерелевантные объекты и ускоряют работу.

Для создания магического варианта запроса все предикаты правил снабжаются «украшениями», своеобразными аннотациями, показывающими, какие аргументы связываются с константами или ограничиваются правилами, а какие являются свободными. «Украшение» для некоторого n-арного предиката, где n – число элементов, – это строка из символов b (bonded) и f (free). В [58] была введена третья литера украшения c (condition), используемая только для независимых условий. Условие на атрибуте X называется независимым, если оно может быть выражено без ссылки на

какой-либо свободный атрибут. Например, условие X > 10 является независимым. Условие X > Y независимо, если атрибут Y является связанным, в противном случае это условие зависимо.

Из «украшенных» предикатов создается граф правил и целей, представляющий структуру правил и фактов и их взаимодействие между собой. На графе должны быть представлены связи между предикатами и правилами, с помощью пунктирных линий – маршрут передачи данных при выполнении запроса. На основе анализа этого графа создается стратегия SIPS (Sideways Information Passing Strategy), определяющая, как передавать информацию сторонним образом в теле табличного выражения при вычислении табличного выражения.

Позже авторы предложили усовершенствованную версию алгоритма «магических множеств» для реляционных баз данных. Формальное определение алгоритма оставалось прежним, исключая замену терминов: «предикат» на «таблицу», а «магическое правило» - на «магическое выражение».

На основе полученных данных производится перезапись запроса, включающая в себя следующие элементы [59]): создании запроса и пустого множества таблиц, создание для каждой таблицы из начального запроса новой магической версии и модифицированного табличного выражения, создание начальной таблицы из предикатов сравнения по равенству в наиболее внешнем блоке запроса. После этой перезаписи запрос сокращается путем объединения табличных выражений с одним и тем же заголовком в одно выражение, в теле которого содержится объединение соответствующих тел.

Представим работу алгоритма на следующем примере. Исходный запрос, выбирающий фамилии старших программистов, которые получают зарплату, большую, чем средняя зарплата их отдела, выглядит следующим образом:

```
SELECT Ename FROM emp e1
WHERE Job = "Sr Programmer" AND
Sal > (SELECT AVG(e2.Sal)
      FROM emp e2
            WHERE e2.Dno =e1.Dno)
```

Исходный запрос обладает набором недостатков, снижающих эффективность его обработки: средняя заплата отдела может вычисляться несколько раз, доступ к e1 и e2 должен производиться в строгом порядке, обработка e2 является покортежной, а не ориентированной для множества, что делает недоступными в данном случае преимущества реляционной модели, ориентированной на обработку множеств.

Преобразованный с помощью алгоритма магических множеств запрос выглядит следующим образом:

```
SELECT Ename FROM s_mag, mag_avgsal
WHERE Sal > Asal AND s_mag.Dno = mag_avgsal.Dno
mag_avgsal(Dno, Asal) AS
(SELECT Dno, AVG(Sal)
FROM mag, emp
WHERE mag.Dno = emp.Dno GROUPBY Dno)
mag(Dno) AS
(SELECT DISTINCT Dno
FROM s_mag)
s_mag(Ename, Dno, Sal) AS
   (SELECT Ename, Dno, Sal FROM emp
WHERE Job = "Sr Programmer")
```

Данная формулировка запроса позволяет избежать повторного обращения к одним и тем же данным и максимально использовать выгоды реляционной модели. Подзапрос s_mag выбирает служащих, являющихся старшими программистами, mag определяет, в каких отделах имеется хотя бы один из таких служащих, mag_avgsal вычисляет среднюю зарплату только для этих отделов. Несмотря на то, что данный алгоритм требует вычисления дополнительных таблиц, очевидно, что его выполнение более эффективно по сравнению с изначально представленным запросом. На примере, представленном в [59], показано, что преобразование методом декорреляции не дает аналогичного эффекта.

Наравне с преобразованием «магических множеств» для работы в рамках реляционной СУБД в алгоритм внесен ряд усовершенствований: в частности, после введения украшения с стала возможной обработка условий, не связанных с равенством, предлагается способ работы алгоритма для нерекурсивных запросов. Аналогичная проблема для работы магических множеств в среде Datalog рассматривалась в [60].

Значительной проблемой оставалась интеграция оптимизации методом «магических множеств» с оценочным оптимизатором, т.е. разработка алгоритмов выбора этого оптимизационного алгоритма из числа иных при обработке различных запросов оптимизатором. Тем более, что результат работы алгоритма очень сильно зависим от изначального представления запроса: если перед «магической» оптимизацией он уже подвергся перезаписи, его преобразованный вид будет отличаться от вида, полученного при обработке его начального представления.

Решение этой проблемы было представлено в [61]. Авторы предложили метод включения алгоритма «магических множеств» в схему оценочного оптимизатора в качестве одного из методов соединений, схожего с операцией полусоединения. Авторы отмечают: «Оценочная оптимизация предотвращает использование перезаписи на основе магических множеств, когда этого делать не следует, и позволяет выбрать один из лучших вариантов, когда перезапись на основе магических множеств целесообразно применять» [61]. В статье

также предлагается расширение алгебры запросов путем введения мультимножественной алгебраической операции Θ-полусоединения, которое позволяло бы моделировать магическую перезапись запросов.

Позже возобновился интерес к применению магических множеств для оптимизации запросов в среде Datalog, в частности – возможной избыточности магических фактов и правил. В ряде работ были предложены способы усовершенствования алгоритма магических множеств и проверки итогового представления запроса с целью устранения избыточностей[62-64].

Судя по доступным нам публикациям, дальнейшие попытки усовершенствовать метод «магических множеств» для реляционных баз данных не предпринимались. Более поздние работы касаются применения магических множеств для задач, отличных от оптимизации запросов, в том числе, интеграции данных [65], применения к запросам на языках, отличных от «классических» Datalog и SQL [66, 67] и т.п.. Отдельно следовало бы выделить работы о применении «магических множеств» для запросов к XML-данным [68, 69].

4. Лексическая оптимизация путем сокращения

Сокращение запроса - это повышение лаконичности и уменьшение числа условий, которое может производиться только при сохранении семантики запроса. Конечно, как показывает ряд примеров (см. предыдущий раздел), оно не может являться самоцелью, однако удаление семантически избыточных условий, дублирующих информацию, уже содержащуюся в запросе, будет способствовать ускорению его обработки.

Каждое условие в составе ограничения требует определенных системных ресурсов на свою реализацию. Следовательно, любые семантические повторы в составе ограничения приводят к росту издержек на обработку запроса. Аналогичным образом, выполнение тождественно ложных или тождественно истинных условий повышает расходы на исполнении запроса. Устранение подобных проявлений семантической избыточности запроса на стадии до его выполнения оптимизирует запрос.

Подобное преобразование условия ставит задачу распознавания эквивалентности запросов и их ограничений. Мы не будем подробно останавливаться на теоретических работах в этой области, и отметим лишь, что они актуальны и сегодня (см., например, [70-71]). Интерес к этой области также возрос в связи с тем, что было доказано сходство данной задачи с проблемой поиска допустимого решения в области искусственного интеллекта и теории операций [72-73].

Распознавание эквивалентности актуально не только при лексической оптимизации запроса, но и при использовании материализованных представлений, которые позволяют оптимизировать запрос путем поиска общих подвыражений, с запросами, ранее обработанными системой, чьи

представления хранятся в памяти и могут быть использованы при реализации нового запроса. Однако последний вид оптимизации находится вне темы нашего обзора, и мы ограничимся ссылкой на работы, посвященные непосредственно ему [см., например, 19, 74-75], а также краткий обзор [5, раздел 7.3].

Исследованиям в области упрощающей лексической оптимизации запроса способствовала, в том числе, работа [76], где было доказано, что любое конъюнктивное ограничение запроса вида

X1 AND X2 ... AND Xn

имеет минимальное эквивалентное представление, которое может быть определено с помощью последовательности операций. Эта задача является NP-сложной. Тем не менее, любое конъюнктивное ограничение может быть преобразовано к более краткой форме путем исключения избыточных условий.

Теоретическую основу упрощающей лексической оптимизации, алгоритмам которой посвящена настоящая часть статьи, по всей видимости, составила работа П. А. В. Холла (P. A. V. Hall) [41], в которой рассматривается применение приемов реляционной алгебры по преобразованию логических выражений в целях оптимизации запросов. Основные сформулированные им идеи лексической оптимизации путем упрощения запроса сводятся к следующему:

- Объединение последовательности проекций в одну;
- Исключение избыточных операций;
- Упрощение выражений, использующих пустые отношения и тривиальные условия;
- Вынос общих выражений «за скобки».

Ряд общих соображений по использованию реляционной алгебры при оптимизации запросов рассматривается также в [77]. При решении задач оптимизации запросов в нереляционных базах данных ряд авторов также применяли подход, представленный, в том числе, в [77]. В его рамках сокращаются избыточные условия, объединенные логическим конъюнктом с представлением ограничений в виде табло, т.е. табличных нотаций [78, 79]. Позже этот алгоритм был усовершенствован для ограничений, содержащих операции дизъюнкции и отрицания [80].

Следующий шаг в решении задачи сокращения запроса сделан в [43, раздел 4.5], где процедура перемещения предиката по дереву запроса используется среди прочего для сопоставления его с предикатами в других узлах на разных уровнях вложенности с целью исключения избыточности. Фактически речь идет о способах сопоставления ограничений запросов, основанных на [73].

Обзор некоторых родственных методов оптимизации запроса, основанных на передаче предикатов между блоками, представлен в [5, раздел 4.3].

Судя по опубликованным данным о работе оптимизаторов широко распространенных СУБД, с целью удаления избыточных условий в них реализуются только алгоритмы поиска общих подвыражений в конъюнктах условий. В частности это касается MySQL [81], PostgreSQL [82] и Oracle [21, p. 9]. Причем в случае PostgreSQL анализ открытого кода позволяет установить, что поиск общих подвыражений реализовывался без использования представления табло на основе сформулированных для различных пар условий правил. Несмотря на то, что этот алгоритм, в сущности, достаточно тривиален, долгое время не было сообщений о попытках его усовершенствования.

Отказ от обработки дизъюнктов ограничений можно рассмотреть на примере работы оптимизатора из описания MySQL 5.5 [12], где при рассмотрении приемов лексической оптимизации приводится преобразование:

```
((a AND b) AND c OR (((a AND b) AND (c AND d)))) = >
(a AND b AND c) OR (a AND b AND c AND d).
```

Однако очевидно, что данное преобразование не завершено и итоговое кратчайшее представление этой функции

```
a AND b AND c,
```

так как

```
X OR (X AND a) = > X.
```

Применимые для решения этой задачи способы минимизации логических функций были разработаны еще в начале 1950-х гг. [83, 84] (т.е. еще до начала специальных исследований в области оптимизации запросов [1]), позже они были применены для программной реализации [85-86]. Возможность использования алгоритма Квайна-Маккласки (Quin-McCluskey) [85-86] при оптимизации запросов допускалась в [87, p. 234], [88], [89, с. 171-172.].

Авторы данного обзора в своих недавних работах [90-92] предложили ряд алгоритмов, позволяющих оптимизировать запрос, используя алгоритмы алгебры логики и линейной алгебры, в том числе упомянутый выше алгоритм Квайна. При этом ограничения запросов рассматривались бы как логические функции или системы линейных неравенств.

Нами были предложены:

- алгоритм поглощения условий-дубликатов и их конъюнктов, основанный на распознавании дубликатов не только в рамках одного конъюнкта, как в ранее упомянутых работах, но и разных конъюнктах одного ограничения, представленного в дизъюнктивной нормальной форме;
- алгоритм минимизации запроса на основе дополненного алгоритма

Квайна (дополнения Маккласки нереализуемы для данного вида задач), при котором ограничение в ДНФ рассматривается как множество утверждений;

- алгоритм минимизации ограничения запроса как системы линейных неравенств.

Кроме алгоритмов, удаляющих избыточные условия, используется также методы упрощения запроса путем объединения и сокращения не отдельных предикатов и их групп, а вложенных подзапросов. Например, в системе Oracle [93, раздел 2] подобная процедура производится над подзапросами на основании «свойства включения» («блок запроса X включает другой блок запроса Y, если результат Y является подмножеством (не обязательно собственным) результата X» [93, p. 1368]).

Важным направлением в области оптимизации запросов путем сокращения является адаптация существующих и разработок новых алгоритмов для обработки запросов к нереляционным СУБД, в особенности, на языке XQuery, предназначенном для взаимодействия с XML-данными.

Однако в существующих работах к оптимизации XQuery-запросов в значительной мере применяются уже хорошо известные приемы. Работы [34, 35] содержат преобразования для алгебры XQuery, тяготеющие, однако, к приемам из [41]. Авторы [54] особо подчеркивают важность развития алгебраического представления запросов с целью применения для оптимизации XQuery-запросов приемов, доказавших свою эффективность для реляционных баз данных. В [30] даже описываются оптимизирующие преобразования над XQuery-запросом, преобразованным в реляционную форму.

5. Заключение

Несмотря на большое число разработанных методов оптимизации запросов, эта задача все еще остается актуальной для существующих СУБД. Как отмечает известный специалист в данной области С. Чаудхари: «Оптимизация запросов столь же, если не более, актуальна, чем в прошлом. Современные база данных включают в себя системы обработки транзакций в реальном времени, ERP-системы, системы управления взаимодействием с клиентами, аналитическую обработку в реальном времени, анализ данных с использованием хранилищ данных (Data-warehouses). Запросы, генерируемые этими приложениями, все сложнее, а базы данных – больше, чем когда бы то ни было. Таким образом, центральная роль оптимизации запросов, которая ищет различные стратегии обработки и выбирает лучший план исполнения, остается несомненной» [94, p. 961]. В том числе, определенного внимания заслуживают алгоритмы лексической оптимизации запросов.

В данном обзоре мы постарались рассмотреть вышедшие за прошедшие годы работы в области лексического подхода к оптимизации запросов и выделить

основные предметы исследований в этой области. Мы предлагаем следующую классификацию.

Перезапись запросов:

- Создание алгебраических представлений для управления запросами
- Перестановка и упорядочение элементов запроса с целью повышения скорости обработки
- Интеграция вложенных подзапросов

Улучшение подзапроса:

- Магические множества

Сокращение подзапросов:

- Удаление избыточных условий и групп условий в рамках одного ограничения
- Сопоставление и сокращений ограничений подзапросов на различных уровнях вложенности

Кроме того, общим для всех указанных направлений является адаптация существующих в них разработок к нереляционным базам данных, в том числе, к обработке запросов на XQuery, предназначенном для обращения к XML документам. Ряд существующих алгоритмов легко применимы к новым запросам, другие нуждаются в существенной доработке для новых задач.

Как показывает пример [90], более внимательного изучения заслуживают существующие математические алгоритмы, применимые для решения различных задач, связанных с оптимизацией запросов.

Список литературы

[1] Palermo F. A data base search problem // Proceedings of the 4th Symposium on Computer and Information Sciences, Virginia, USA, 1972. Restion: AFIPS Press, 1972. Pp. 67-101.

[2] Jarke M., Koch J. Query Optimization in Database Systems // ACM Computing Surveys (CSUR), 1984. March, Volume 16, Issue 2. Pp. 111-152.

[3] Mannino M. V., Chu P., Sager T. Statistical profile estimation in database systems // ACM Computing Surveys, 1988. September, Volume 20, Issue 3. Pp. 191-221.

[4] Ionnidis Y. E. Query Optimization // The Computer Science and Engineering Handbook. Boca Raton: CRC Press, 1996. Pp. 1038-1054.

[5] Chaudhari S. An Overview of Query Optimization in Relational Systems // Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. New York: SIGMOD, 1998. Pp. 34-43.

[6] Кузнецов С. Д. Методы оптимизации выполнения запросов в реляционных СУБД // http://www.citforum.ru/database/articles/art_26.shtml [Обращение 20 сентября 2012].

[7] Graefe G. Query Evaluation Techniques for Large Databases // ACM Computing Surveys, 1993. Volume 25, Issue 2. P. 73-169.
[8] Дейт К. Дж. Введение в системы баз данных. Москва – Санкт-Петербург – Киев: Издательский дом «Вильямс», 2001. 1072 с.
[9] Ozsu M. N., Valduriez P. Principles of Distributed Database Systems. Second Edition. New Jersey: Prentice Hall International, 1999. 666 pp.
[10] Ramakrishnan R., Gehrcke J. Database System Management. 2nd Edition. Singapore: The McGraw-Hill Book Co, 2000. 906 pp.
[11] Karayannidis N. Query Optimization Bibliography, http://www.dbnet.ece.ntua.gr/~nikos/edith/qopt_bibl/, [Обращение 20 сентября 2012].
[12] MySQL 5.5 Reference Manual. Chapter 7. Optimization. http://dev.mysql.com/doc/refman/5.5/en/optimization.html, [Опубликовано в 2010 году, обращение 20 сентября 2012].
[13] Веллин Л., Томсон Дж. MySQL. Учебное пособие. Перевод с английского. М.: Издательский дом «Вильямс», 2005. 292 с.
[14] Upgrading from Oracle Database 10 g to 11 g: What to expect from the Optimizer. An Oracle White Paper, November 2009. Redwood Shores: Oracle Corporation, 2009. 36 pp.
[15] Comparison of Materialized Views & Analytic Workspaces in Oracle Database 11 g. An Oracle White Paper, March 2008. Redwood Shores: Oracle Corporation, 2008. 27 pp.
[16] Markl V., Lohman G. M., Raman V. LEO: An autonomic query optimizer for DB2 // IBM System Journal, 2003. V. 42, № 1. Pp. 98-106.
[17] PostgreSQL 8.1.19 Documentation. Chapter VII: Internals, http://www.postgresql.org/docs/8.1/interactive/internals.html, [Обращение 20 сентября 2012].
[18] Ioannidis Y. The History of Histograms // Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany. Berlin: Morgan Kaufmann, 2003. Pp. 19-30.
[19] Halevy Y. Answering queries using views: A survey // The International Journal on Very Large Data Bases, December 2001. Volume 10 Issue 4. Pp. 270-294.
[20] Braga D., Ceri S., Grossniklaus M. Join Methods and Query Optimization // Lecture Notes on Computer Science, 2010. Issue 5950. Pp. 188-210.
[21] Query Optimization in Oracle Database 10g Release 2. An Oracle White Paper, June 2005. Redwood Shores: Oracle Corporation, 2005. 31 pp.
[22] Смирнов С. Н., Задворьев И. С. Работаем с Oracle: Учебное пособие. М: Гелиос АРВ, 2002. 496 сс.
[23] Shenoy S. T., Ozsoyoglu Z. M. A System for Semantic Query Optimization // SIGMOD Record, 1987. Volume 16, Number 3. Pp. 181-195
[24] Geng K., Dobbie G., Meng Y. Survey of XML Semantic Query Optimization // Proceedings 4th International Conference on Internet Computing for Science and Engineering (ICICSE), Harbin, 2009. Washington D. C.: IEEE Computer Society, 2009. Pp. 297-300
[25] Зверев Д. Л. Оптимизация потоков простых SQL-запросов: Диссертация кандидата технических наук: 05.13.11. Санкт-Петербург: Санкт-Петербургский Государственный Университет Аэрокосмического Приборостроения, 2005. 169 сс.
[26] XQuery 1.0 and XPath 2.0 Formal Semantics. World Wide Web Consortium (W3C), W3C Recommendation, 2007. http://www.w3.org/TR/xquery-semantics/, [Обращение 20 сентября 2012].

[27] Deutsch A., V. Tannen XML queries and constraints, containment and reformulation // Theoretical Computer Science, 2005. № 336. Pp. 57-87.
[28] Fan W., Xu J. Y., Ding B., Qin L., Rastogi R. Query Translation from XPath to SQL in the Presence of Recursive DTDs // Very Large Data Bases Journal, 2009. Issue 18. Pp. 857-883.
[29] Барашев Д. В., Горшкова Е. А., Новиков Б. А. Оптимизация представления XML документов в реляционной базе данных // Вторая Всероссийская научная конференция. Электронные библиотеки. Перспективные методы и технологии, электронные коллекции. 26-28 сентября, Протвино, 2000. С. 224-229.
[30] Manolescu I., Florescu D., Kossmann D. Answering xml queries on heterogeneous data sources // Proceedings of the 27th International Conference on Very Large Data Bases, San Francisco, 2001. San Francisco: Morgan Kaufmann Publishers Inc., 2001. Pp.241-250.
[31] Krishnamurthy R., Kaushik R., Naughton J. XML-SQL query translation literature: The state of the art and open problems // XML Database Symposium (XSym 2003) at VLDB 2003. Berlin, September 2003. Lecture Notes in Computer Science, Volume 2824, 2003. Pp. 1-18.
[32] Cluet S., Moerkotte G. Nested queries in object bases // Proceedings of the 4th International Workshop on Database Programming Languages: Object Models and Language. Manhattan, New York City, USA, 30 August-1 September, 1993. London: Springer-Verlag, 2004. Pp. 226-242.
[33] Steenhagen H. J., Apers P. M. G., Blanken H.M., de By R. A. From nested-loop to join queries in OODB // Proceedings of the 20th International Conference on Very Large Data Bases, 1994. Santiago: Morgan Kaufman, 1994 P. 618-629.
[34] Frasincar F., Houben G.-J., Pau C. XAL: An algebra for XML query optimization // Proceedings of the 13th Australasian Database Conference, Australian Computer Society, Inc. Darlinghurst, Australia, 2002. Volume 5. Melbourne: Australian Computer Society, 2002. Pp. 49-56.
[35] May N., Helmer S., Moerkotte G. Strategies for Query Unnesting in XML Databases // ACM Transactions on Database Systems (TODS) 2006. Volume 31, Issue 3. Pp. 968-1013.
[36] Re C., Simeon J., Fernandez M. F. A Complete and Efficient Algebraic Compiler for XQuery // Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April, 2006, Atlanta, GA, USA: IEEE Computer Society, 2006. Lecture Notes on Computer Society, 2007. Volume 4797. Pp. 81-96.
[37] Ghelli G., Onose N., Rose K., Siméon J. XML Query Optimization in the Presence of Side Effects // Proceedings of the 2008 ACM SIGMOD international conference on Management of data. New York, USA, 2008. New York: ACM, 2008. Pp. 339-352.
[38] Lukichev M., Barashev D. XML Query Algebra for Cost-based Optimization // SYRCODIS*07 The Fourth Spring Young Researchers Colloquium on Databases and Information Systems, Moscow, May 31 - June 1, 2007. http://ceur-ws.org/Vol-256/, [Обращение 20 сентября 2012].
[39] Лукичев М. С. Оптимизация запросов в слабоструктурированной модели данных. Диссертация кандидата физико-математических наук: 05.13.11. Санкт-Петербург: Санкт-Петербургский Государственный Университет, 2009. 120 с.
[40] Smith M., Chang P. Y. W. Optimizing the performance of a relational algebra database interface // Communications for the ACM, October, 1975. Volume 18, Issue 10. Pp. 568-579.

[41] Hall P. A. V. Optimization of single expressions in a relational data base system // IBM Journal of Research and Development. Volume 20, Number 3, 1976. Pp. 244-257.
[42] Chaudhuri S., Shim K. Including Group-By in Query Optimization // Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann, San Mateo, USA, 1994. San Francisco: Morgan Kaufmann Publishers Inc., 1994. Pp. 354-366.
[43] Levy Y., Mumick I. S., Sagiv Y. Query Optimization by Predicate Move Around // Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann, San Mateo, USA, 1994. Morgan Kaufmann Publishers Inc., 1994. Pp. 96-107.
[44] Wong E., Youssefi K. Decomposition - a strategy for query processing // ACM Transactions On Database Systems, September 1976. Volume 1, Number 3. Pp. 223–241.
[45] Yannakakis M. Algorithms for acyclic database schemes // Proceedings of Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings. IEEE Computer Society 1981. New York: IEEE Press, 1981. Pp. 82-94.
[46] McMahan B., Porter P., Pan G., M. Y. Vardi Projection Pushing Revisited // Proceedings of the 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004. Berlin: Springer, 2004. Pp. 441-458.
[47] McMahan B. J. Structural Heuristics for Query Optimization. Master of Science Degree Thesis. Houston: Rice University, 2004. 64 pp.
[48] Kim W. On optimizing an SQL-Like Nested Query // ACM Transactions on Database Systems (TODS), September, 1982. Volume 7 Issue 3. Pp. 443-469.
[49] Kisessling W. On Semantic Reefs and Efficient Processing of Correlation Queries Revisited // Proceedings of 11th International Conference for Very Large Data Bases. August 21-23, Stockholm, Sweden, 1985. New York: Morgan Kaufmann, 1985. Pp. 241-250.
[50] Ganski R. A., Wong H. K. T. Optimization of Nested SQL Queries Revisited // Proceedings of the ACM SIGMOD international conference on Management of data. San Francisco, May 1987. New York: ACM, 1987. Pp. 23-33.
[51] Muralikrishna M. Improved unnesting algorithms for join aggregate SQL queries // Proceedings of the 18th International Conference on Very Large Data Bases, August 23-27, Vancouver, Canada, 1992. San Francisco: Morgan Kaufmann Publishers Inc., 1992. Pp. 91-102.
[52] Khaitan P., Satish K. M., S. B. Korra, S. K. Jena Improved query plans for unnesting nested SQL queries // Proceedings of 2nd International Conference on Computer Science and its Applications, December 10-12, South Korea, 2009. Jeju Island : IEEE, 2009. Pp. 147-152.
[53] Fegaras L., D. Levine, S. Bose, V. Chaluvadi Query processing of streamed XML data // Proceedings of the eleventh international conference on Information and knowledge management. ACM Press, New York, USA, 2002. Berlin: Springer, 2004. Pp. 195-215.
[54] May N., Moerkotte G. Normalization and Translation of XQuery // Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization and Data Query Technologies. Hershey: Igi Global Publishing, 2010. Pp. 283-307.
[55] Pirahesh H., Hellerestein J., Hasan W. Extensible/rule based query rewrite optimization in Starburst // ACM SIGMOD Record, June 1, 1992. Volume 21, Issue 2. Pp. 39-48.
[56] MySQL Nested-Loop Join Algorithms, http://dev.mysql.com/doc/refman/5.5/en/nested-loop-joins.html, [Обращение 20 сентября 2012].

[57] Bancilhon F., Maierl D., Sagiv Y., Ullman J. D. Magic Sets and Other Strange Ways to Implement Logic Programs // Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Cambridge, Massachusetts, March 24-26, 1986. New York: ACM, 1986. Pp. 1-15.
[58] Mumick S., Finkelsteint S. J., Pirahesh H., Ramakrishnan R. Magic Conditions // ACM Transactions on Database Systems (TODS), March 1996. Volume 21 Issue 1. Pp. 107-155.
[59] Mumick I. S., Finkelstein S. J., Pirahesh H., Ramakrishnan R. Magic is Relevant // Proceedings of the 1990 ACM SIGMOD international conference on Management of data. New York: ACM, 1990. Pp. 247-258.
[60] Jezek K., Zima M. Query optimization in deductive programs with aggregates // Proceeding of the 4th International Conference Information Systems Modelling, Ostrava: MARQ, 2001. Pp. 85-92.
[61] Seshadri P., Hellerstein J. M., Pirahesh H., Cliff Leung T. Y., Ramakrishnan R., Srivastava D., Stuckey P. J., Sudarshan S. Cost-Based Optimization for Magic: Algebra and Implementation // ACM SIGMOD Record, June 1996. Volume 25, Issue 2, Pp. 28-33.
[62] Sagiv Y. Is there anything better than magic? // Logic Programming, Proceedings of the 1990 North American Conference, Austin, Texas, October 29 - November 1, 1990. Austin: MIT Press 1990. Pp. 235-254.
[63] Sippu S., Soisalon-Soininen E. An Analysis of Magic Sets and Related Optimization Strategies for Logic Queries // Journal of the ACM, November 1996. Volume 43, № 6. Pp. 1046-1088.
[64] Azevedo P. J. Magic sets with full sharing // The Journal of Logic Programming, 1997. Volume 30, № 3. Pp. 223-237.
[65] Faber W., Greco G., Leone N. Magic Sets and their application to data integration // Journal of Computer and System Sciences, June 2007. Volume 73, Issue 4. Pp. 584-609.
[66] Ruckhaus E., Ruiz E., Vidal M. E. OnEQL: An Ontology Efficient Query Language Engine for the Semantic Web // Proceedings of the Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS), Porto, Portugal, September 13th, 2007. Porto: CEUR Workshop Proceedings, 2007. Pp. 65-88.
[67] Alviano M., Faber W., Greco G., Leone N. Magic Sets for Disjunctive Datalog Programs // Artificial Intelligence, 2012. Volume 187. Pp. 156-192.
[68] Almendros-Jiménez M., Becerra-Terón A., Enciso-Banos F. J. Magic Sets for the XPath Language // Journal of Universal Computer Science, 2006. Volume 12, № 11. Pp. 1651-1678.
[69] Ozcan F., Seemann N., Wang L. XQuery Rewrite Optimization in IBM DB2 pureXML // IEEE, Data Engineering Bulletin, December 2008. Volume 34, Number 4. Pp. 25-32.
[70] Kolaitis P. G., Martin D.L., Thakur M.N. On the complexity of the containment problem for conjunctive queries with built-in predicates // Proceeding of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Seattle, Washington, USA, June 1-3, 1998. New York: ACM, 1998. Pp. 197-204.
[71] Benedikt M., Gottlob G. The Impact of Virtual Views on Containment // Proceedings of the The Very Large Data Bases, September 2010. Volume 3 Issue 1-2. Pp. 297-308.
[72] Kolaitis P. G., Vardi M. Conjunctive-query containment and constraint satisfaction // Proceeding of the 17th ACM SIGACT-SIGMOD-SIGART SIGART Symposium on Principles of Database Systems (PODS), Seattle, Washington, USA, June 1-3, 1998. New York: ACM, 1998. Pp. 205-213.

[73] Kolaitis P. G., Vardi M. Conjunctive-Query Containment and Constraint Satisfaction // Journal of Computer and System Sciences, 2000. № 61. Pp. 302-332.
[74] Goldstein J., Larson P.-A. Optimization Queries Using Materialized Views: A Practical Scalable Solution // ACM SIGMOD Record, June 2001. Volume 30 Issue 2. Pp.331-342.
[75] Пашинин О. В. Оптимизация запросов к базам данных // Математические структуры и моделирование. Выпуск 17, 2007. С. 100-107ю
[76] Chandra K., Merlin P. M. Optimal Implementation of Conjunctive Queries in Relational Databases // Proceedings of the 9th annual ACM symposium on Theory of computing, May, 1977. New York: ACM, 1977. Pp. 77-90.
[77] Stroet J. W. M., Engmann R. Manipulation of expressions in a relational algebra // Information Systems, 1979. Volume 4, Issue 4. Pp. 195-203.
[78] Aho V., Sagiv Y., Ullman J. D. Equivalences among relational expressions // Society for Industrial and Applied Mathematics Journal on Computing, 1979. Volume 8, Issue 2. Pp. 218-246.
[79] Aho V., Sagiv Y., Ullman J. D. Efficient optimization of a class of relational expressions // Journal ACM Transactions on Database Systems (TODS), December 1979. Volume 4, Issue 4. Pp.435-454.
[80] Sagiv Y., Yannakakis M. Equivalences among relational expressions with the union and difference operators // Journal of the ACM, October 1980. Volume 27, Issue 4. Pp. 633-655.
[81] 7.2.1.2 How MySQL Optimizes WHERE Clauses http://dev.mysql.com/doc/refman/5.5/en/where-optimizations.html, [Обращение 20 сентября 2012].
[82] Приложение PostgreSQL 8.3.3., http://www.postgresql.org/download/, [Обращение 20 сентября 2012]. Адрес файла, содержащего обсуждаемый код, в архиве postgresql-8.3.3\src\backend\optimizer\util\predtest.c.
[83] Veitch E. W. A Chart Method for Simplifying Truth Functions // ACM Annual Conference/Annual Meeting: Proceedings of the 1952 ACM Annual Meeting. Pittsburg: ACM, NY, 1952. Pp. 127-133.
[84] Karnaugh M. The Map Method for Synthesis of Combinational Logic Circuits // Transactions of the American Institute of Electrical Engineers, November 1953. Part I, № 72 (9). Pp. 593-599.
[85] Quin W. V. On Cores and Prime Implicants of Truth Functions // American Mathematics Monthly, 1959. V. 66, № 9. P. 755-760.
[86] McCluskey E. J. Minimization of Boolean Functions // The Bell System Technical Journal, November 1956. V. 35, Issue 5. Pp. 1236-1249.
[87] Wu M.-C. Query Optimizatiom for Selecting Using Bitmaps // ACM SIGMOD Record, June 1999. Volume 28 Issue 2. Pp. 227-238.
[88] Das Sarma A., Theobald M., Widom J. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. Technical Report. Stanford, 2007. http://ilpubs.stanford.edu:8090/800/, [Обращение 20 сентября 2012].
[89] Тарасенко П. Ф., Бухарова М. Ф. Технология «The Reporter» для построения отчетов по базам данных // Вестник Томского Государственного Университета, № 275, апрель 2002. С. 167-176.
[90] Mendkovich N., Kuznetcov S. New Algorithms for Lexical Query Optimization // Proceedings of the 31st International Conference on Information Technology Interfaces. Cavtat/Dubrovnik, Croatia, June 22-25, 2009. Zagreb: University of Zagreb, 2009. Pp. 187-192.

[91] Кузнецов С. Д., Мендкович Н. А. Новые алгоритмы лексической оптимизации запросов // Модели и анализ информационных систем, 2009. Т. 16, № 4. С. 22-33.
[92] Мендкович Н. А., Кузнецов С. Д. Оптимизация конъюнктов условий в составе запросов // Модели и анализ информационных систем, 2011. Т. 18, № 3. С. 144-154.
[93] Bellamkonda S., Ahmed R., Witkowski A., Amor A., Zait M., Lin Ch.-Ch. Enhanced Subquery Optimizations in Oracle / // Proceedings of the 35th international conference on Very large data base, August 2009. Volume 2 Issue 2. Pp. 1366-1377.
[94] Chaudhuri S. Query Optimizers: Time to Rethink the Contract? // Proceedings of the ACM SIGMOD International Conference on Management of Data, Providence, Rhode Island, USA, June 29 - July 2, 2009. New York: ACM, 2009. Pp. 961-968.

An Overwiew of Evolution of Lexical Query Optimization Techniques

Mendkovich N.A.
OOO «FREEnet Group», Moscow, Russia
mend@rambler.ru
Kuznetsov S.D.
ISP RAS, Moscow, Russia
kuzloc@ispras.ru

Abstract. The presented overview is concerned with lexical query optimization and covers papers published in the last four decades. The paper consists of five sections. The first section – Introduction – classifies query optimization techniques into semantic optimizations and lexical optimizations. Semantic optimizations usually relies on data integrity rules that are stores within metadata part of databases, and on data statistics. This kind of optimizations is discussed in many textbooks and papers. Lexical optimizations (more often called rewriting) use only a text of query and no other information about database and its structure. Lexical optimizations are further classified into query transformations, query amelioration, and query reduction. The second section of the paper discusses techniques of query transformation such as predicate pushdown, transformation of nested query into query with joins, etc. Query amelioration is a topic of the third section with a focus on magic set optimizations. The forth section covers query reduction optimizations. The section briefly describes traditional approaches (such as tableau -based) are briefly described and considers in more details three new algorithms proposed by authors. The fifth section concludes the paper.

Keywords: query optimization; query simplification; lexical query optimization; magic sets

References

[1]. Palermo F. A data base search problem. Proceedings of the 4th Symposium on Computer and Information Sciences, Virginia, USA, 1972. Restion: AFIPS Press, 1972. Pp. 67-101.
[2]. Jarke M., Koch J. Query Optimization in Database Systems. ACM Computing Surveys (CSUR), 1984. March, Volume 16, Issue 2. Pp. 111-152.
[3]. Mannino M. V., Chu P., Sager T. Statistical profile estimation in database systems. ACM Computing Surveys, 1988. September, Volume 20, Issue 3. Pp. 191-221.
[4]. Ionnidis Y. E. Query Optimization. The Computer Science and Engineering Handbook. Boca Raton: CRC Press, 1996. Pp. 1038-1054.
[5]. Chaudhari S. An Overview of Query Optimization in Relational Systems. Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. New York: SIGMOD, 1998. Pp. 34-43.
[6]. Kuznetsov S.D. Metody optimizacii vypolnenija zaprosov v reljacionnyh SUBD [Query optimization techniques for relational DBMSs]. http://www.citforum.ru/database/articles/art_26.shtml (in Russian).
[7]. Graefe G. Query Evaluation Techniques for Large Databases. ACM Computing Surveys, 1993. Volume 25, Issue 2. P. 73-169.

[8]. Date C. J. Vvedenie v sistemy baz dannyh [An Introduction to Database Systems]. Moskva – Sankt-Peterburg – Kiev: Izdatel'skij dom «Vil'jams», 2001. 1072 s. (in Russian).
[9]. Ozsu M. N., Valduriez P. Principles of Distributed Database Systems. Second Edition. New Jersey: Prentice Hall International, 1999. 666 pp.
[10]. Ramakrishnan R., Gehrcke J. Database System Management. 2nd Edition. Singapore: The McGraw-Hill Book Co, 2000. 906 pp.
[11]. Karayannidis N. Query Optimization Bibliography, http://www.dbnet.ece.ntua.gr/~nikos/edith/qopt_bibl/.
[12]. MySQL 5.5 Reference Manual. Chapter 7. Optimization. http://dev.mysql.com/doc/refman/5.5/en/optimization.html.
[13]. Vellin L., Tomson Dzh. MySQL. Uchebnoe posobie [Textbook on MySQL]. Perevod s anglijskogo. M.: Izdatel'skij dom «Vil'jams», 2005. 292 c.
[14]. Upgrading from Oracle Database 10 g to 11 g: What to expect from the Optimizer. An Oracle White Paper, November 2009. Redwood Shores: Oracle Corporation, 2009. 36 pp.
[15]. Comparison of Materialized Views & Analytic Workspaces in Oracle Database 11 g. An Oracle White Paper, March 2008. Redwood Shores: Oracle Corporation, 2008. 27 pp.
[16]. Markl V., Lohman G. M., Raman V. LEO: An autonomic query optimizer for DB2. IBM System Journal, 2003. V. 42, № 1. Pp. 98-106.
[17]. PostgreSQL 8.1.19 Documentation. Chapter VII: Internals, http://www.postgresql.org/docs/8.1/interactive/internals.html.
[18]. Ioannidis Y. The History of Histograms. Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany. Berlin: Morgan Kaufmann, 2003. Pp. 19-30.
[19]. Halevy Y. Answering queries using views: A survey. The International Journal on Very Large Data Bases, December 2001. Volume 10 Issue 4. Pp. 270-294.
[20]. Braga D., Ceri S., Grossniklaus M. Join Methods and Query Optimization. Lecture Notes on Computer Science, 2010. Issue 5950. Pp. 188-210.
[21]. Query Optimization in Oracle Database 10g Release 2. An Oracle White Paper, June 2005. Redwood Shores: Oracle Corporation, 2005. 31 pp.
[22]. Smirnov S. N., Zadvor'ev I. S. Rabotaem s Oracle: Uchebnoe posobie [Working with Oracle: textbook]. M: Gelios ARV, 2002. 496 s. (in Russian).
[23]. Shenoy S. T., Ozsoyoglu Z. M. A System for Semantic Query Optimization. SIGMOD Record, 1987. Volume 16, Number 3. Pp. 181-195
[24]. Geng K., Dobbie G., Meng Y. Survey of XML Semantic Query Optimization. Proceedings 4th International Conference on Internet Computing for Science and Engineering (ICICSE), Harbin, 2009. Washington D. C.: IEEE Computer Society, 2009. Pp. 297-300
[25]. Zverev D. L. Optimizacija potokov prostyh SQL-zaprosov [Optimization of simple SQL query' streams]: Dissertacija kandidata tehnicheskih nauk: 05.13.11. Sankt-Peterburg: Sankt-Peterburgskij Gosudarstvennyj Universitet Ajerokosmicheskogo Priborostroenija, 2005. 169 s. (in Russian).
[26]. XQuery 1.0 and XPath 2.0 Formal Semantics. World Wide Web Consortium (W3C), W3C Recommendation, 2007..
[27]. Deutsch A., V. Tannen XML queries and constraints, containment and reformulation. Theoretical Computer Science, 2005. № 336. Pp. 57-87.

[28]. Fan W., Xu J. Y., Ding B., Qin L., Rastogi R. Query Translation from XPath to SQL in the Presence of Recursive DTDs. Very Large Data Bases Journal, 2009. Issue 18. Pp. 857-883.
[29]. Barashev D. V., Gorshkova E. A., Novikov B. A. Optimizacija predstavlenija XML dokumentov v reljacionnoj baze dannyh [Optimization of XML document's representation within a relational database]. Vtoraja Vserossijskaja nauchnaja konferencija. Jelektronnye biblioteki. Perspektivnye metody i tehnologii, jelektronnye kollekcii. 26-28 sentjabrja, Protvino, 2000. C. 224-229 (in Russian).
[30]. Manolescu I., Florescu D., Kossmann D. Answering xml queries on heterogeneous data sources. Proceedings of the 27th International Conference on Very Large Data Bases, San Francisco, 2001. San Francisco: Morgan Kaufmann Publishers Inc., 2001. Pp.241-250.
[31]. Krishnamurthy R., Kaushik R., Naughton J. XML-SQL query translation literature: The state of the art and open problems. XML Database Symposium (XSym 2003) at VLDB 2003. Berlin, September 2003. Lecture Notes in Computer Science, Volume 2824, 2003. Pp. 1-18.
[32]. Cluet S., Moerkotte G. Nested queries in object bases. Proceedings of the 4th International Workshop on Database Programming Languages: Object Models and Language. Manhattan, New York City, USA, 30 August-1 September, 1993. London: Springer-Verlag, 2004. Pp. 226-242.
[33]. Steenhagen H. J., Apers P. M. G., Blanken H.M., de By R. A. From nested-loop to join queries in OODB. Proceedings of the 20th International Conference on Very Large Data Bases, 1994. Santiago: Morgan Kaufman, 1994 P. 618-629.
[34]. Frasincar F., Houben G.-J., Pau C. XAL: An algebra for XML query optimization. Proceedings of the 13th Australasian Database Conference, Australian Computer Society, Inc. Darlinghurst, Australia, 2002. Volume 5. Melbourne: Australian Computer Society, 2002. Pp. 49-56.
[35]. May N., Helmer S., Moerkotte G. Strategies for Query Unnesting in XML Databases. ACM Transactions on Database Systems (TODS) 2006. Volume 31, Issue 3. Pp. 968-1013.
[36]. Re C., Simeon J., Fernandez M. F. A Complete and Efficient Algebraic Compiler for XQuery. Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April, 2006, Atlanta, GA, USA: IEEE Computer Society, 2006. Lecture Notes on Computer Society, 2007. Volume 4797. Pp. 81-96.
[37]. Ghelli G., Onose N., Rose K., Siméon J. XML Query Optimization in the Presence of Side Effects. Proceedings of the 2008 ACM SIGMOD international conference on Management of data. New York, USA, 2008. New York: ACM, 2008. Pp. 339-352.
[38]. Lukichev M., Barashev D. XML Query Algebra for Cost-based Optimization. SYRCODIS*07 The Fourth Spring Young Researchers Colloquium on Databases and Information Systems, Moscow, May 31 - June 1, 2007. http://ceur-ws.org/Vol-256/ (in Russian).
[39]. Lukichev M. S. Optimizacija zaprosov v slabostrukturirovannoj modeli dannyh [Query optimization in a semistructured data model]. Dissertacija kandidata fiziko-matematicheskih nauk: 05.13.11. Sankt-Peterburg: Sankt-Peterburgskij Gosudarstvennyj Universitet, 2009. 120 s. (in Russian).
[40]. Smith M., Chang P. Y. W. Optimizing the performance of a relational algebra database interface. Communications for the ACM, October, 1975. Volume 18, Issue 10. Pp. 568-579.

[41]. Hall P. A. V. Optimization of single expressions in a relational data base system. IBM Journal of Research and Development. Volume 20, Number 3, 1976. Pp. 244-257.
[42]. Chaudhuri S., Shim K. Including Group-By in Query Optimization. Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann, San Mateo, USA, 1994. San Francisco: Morgan Kaufmann Publishers Inc., 1994. Pp. 354-366.
[43]. Levy Y., Mumick I. S., Sagiv Y. Query Optimization by Predicate Move Around. Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann, San Mateo, USA, 1994. Morgan Kaufmann Publishers Inc., 1994. Pp. 96-107.
[44]. Wong E., Youssefi K. Decomposition - a strategy for query processing. ACM Transactions On Database Systems, September 1976. Volume 1, Number 3. Pp. 223–241.
[45]. Yannakakis M. Algorithms for acyclic database schemes. Proceedings of Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings. IEEE Computer Society 1981. New York: IEEE Press, 1981. Pp. 82-94.
[46]. McMahan B., Porter P., Pan G., M. Y. Vardi. Projection Pushing Revisited. Proceedings of the 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004. Berlin: Springer, 2004. Pp. 441-458.
[47]. McMahan B. J. Structural Heuristics for Query Optimization. Master of Science Degree Thesis. Houston: Rice University, 2004. 64 pp.
[48]. Kim W. On optimizing an SQL-Like Nested Query. ACM Transactions on Database Systems (TODS), September, 1982. Volume 7 Issue 3. Pp. 443-469.
[49]. Kisessling W. On Semantic Reefs and Efficient Processing of Correlation Queries Revisited. Proceedings of 11th International Conference for Very Large Data Bases. August 21-23, Stockholm, Sweden, 1985. New York: Morgan Kaufmann, 1985. Pp. 241-250.
[50]. Ganski R. A., Wong H. K. T. Optimization of Nested SQL Queries Revisited. Proceedings of the ACM SIGMOD international conference on Management of data. San Francisco, May 1987. New York: ACM, 1987. Pp. 23-33.
[51]. Muralikrishna M. Improved unnesting algorithms for join aggregate SQL queries. Proceedings of the 18th International Conference on Very Large Data Bases, August 23-27, Vancouver, Canada, 1992. San Francisco: Morgan Kaufmann Publishers Inc., 1992. Pp. 91-102.
[52]. Khaitan P., Satish K. M., S. B. Korra, S. K. Jena Improved query plans for unnesting nested SQL queries. Proceedings of 2nd International Conference on Computer Science and its Applications, December 10-12, South Korea, 2009. Jeju Island : IEEE, 2009. Pp. 147-152.
[53]. Fegaras L., D. Levine, S. Bose, V. Chaluvadi. Query processing of streamed XML data. Proceedings of the eleventh international conference on Information and knowledge management. ACM Press, New York, USA, 2002. Berlin: Springer, 2004. Pp. 195-215.
[54]. May N., Moerkotte G. Normalization and Translation of XQuery. Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization and Data Query Technologies. Hershey: Igi Global Publishing, 2010. Pp. 283-307.
[55]. Pirahesh H., Hellerestein J., Hasan W. Extensible/rule based query rewrite optimization in Starburst. ACM SIGMOD Record, June 1, 1992. Volume 21, Issue 2. Pp. 39-48.
[56]. MySQL Nested-Loop Join Algorithms, http://dev.mysql.com/doc/refman/5.5/en/nested-loop-joins.html.

[57]. Bancilhon F., Maierl D., Sagiv Y., Ullman J. D. Magic Sets and Other Strange Ways to Implement Logic Programs. Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Cambridge, Massachusetts, March 24-26, 1986. New York: ACM, 1986. Pp. 1-15.
[58]. Mumick S., Finkelsteint S. J., Pirahesh H., Ramakrishnan R. Magic Conditions. ACM Transactions on Database Systems (TODS), March 1996. Volume 21 Issue 1. Pp. 107-155.
[59]. Mumick I. S., Finkelstein S. J., Pirahesh H., Ramakrishnan R. Magic is Relevant. Proceedings of the 1990 ACM SIGMOD international conference on Management of data. New York: ACM, 1990. Pp. 247-258.
[60]. Jezek K., Zima M. Query optimization in deductive programs with aggregates. Proceeding of the 4th International Conference Information Systems Modelling, Ostrava: MARQ, 2001. Pp. 85-92.
[61]. Seshadri P., Hellerstein J. M., Pirahesh H., Cliff Leung T. Y., Ramakrishnan R., Srivastava D., Stuckey P. J., Sudarshan S. Cost-Based Optimization for Magic: Algebra and Implementation. ACM SIGMOD Record, June 1996. Volume 25, Issue 2, Pp. 28-33.
[62]. Sagiv Y. Is there anything better than magic?. Logic Programming, Proceedings of the 1990 North American Conference, Austin, Texas, October 29 - November 1, 1990. Austin: MIT Press 1990. Pp. 235-254.
[63]. Sippu S., Soisalon-Soininen E. An Analysis of Magic Sets and Related Optimization Strategies for Logic Queries. Journal of the ACM, November 1996. Volume 43, № 6. Pp. 1046-1088.
[64]. Azevedo P. J. Magic sets with full sharing. The Journal of Logic Programming, 1997. Volume 30, № 3. Pp. 223-237.
[65]. Faber W., Greco G., Leone N. Magic Sets and their application to data integration. Journal of Computer and System Sciences, June 2007. Volume 73, Issue 4. Pp. 584-609.
[66]. Ruckhaus E., Ruiz E., Vidal M. E. OnEQL: An Ontology Efficient Query Language Engine for the Semantic Web. Proceedings of the Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS), Porto, Portugal, September 13th, 2007. Porto: CEUR Workshop Proceedings, 2007. Pp. 65-88.
[67]. Alviano M., Faber W., Greco G., Leone N. Magic Sets for Disjunctive Datalog Programs. Artificial Intelligence, 2012. Volume 187. Pp. 156-192.
[68]. Almendros-Jiménez M., Becerra-Terón A., Enciso-Banos F. J. Magic Sets for the XPath Language. Journal of Universal Computer Science, 2006. Volume 12, № 11. Pp. 1651-1678.
[69]. Ozcan F., Seemann N., Wang L. XQuery Rewrite Optimization in IBM DB2 pureXML. IEEE, Data Engineering Bulletin, December 2008. Volume 34, Number 4. Pp. 25-32.
[70]. Kolaitis P. G., Martin D.L., Thakur M.N. On the complexity of the containment problem for conjunctive queries with built-in predicates. Proceeding of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Seattle, Washington, USA, June 1-3, 1998. New York: ACM, 1998. Pp. 197-204.
[71]. Benedikt M., Gottlob G. The Impact of Virtual Views on Containment. Proceedings of the The Very Large Data Bases, September 2010. Volume 3 Issue 1-2. Pp. 297-308.
[72]. Kolaitis P. G., Vardi M. Conjunctive-query containment and constraint satisfaction. Proceeding of the 17th ACM SIGACT-SIGMOD-SIGART SIGART Symposium on Principles of Database Systems (PODS), Seattle, Washington, USA, June 1-3, 1998. New York: ACM, 1998. Pp. 205-213.

[73]. Kolaitis P. G., Vardi M. Conjunctive-Query Containment and Constraint Satisfaction. Journal of Computer and System Sciences, 2000. № 61. Pp. 302-332.
[74]. Goldstein J., Larson P.-A. Optimization Queries Using Materialized Views: A Practical Scalable Solution. ACM SIGMOD Record, June 2001. Volume 30 Issue 2. Pp.331-342.
[75]. Pashinin O. V. Optimizacija zaprosov k bazam dannyh [Optimization of queries to relational databases] . Matematicheskie struktury i modelirovanie. Vypusk 17, 2007. S. 100-107 (in Russian).
[76]. Chandra K., Merlin P. M. Optimal Implementation of Conjunctive Queries in Relational Databases. Proceedings of the 9th annual ACM symposium on Theory of computing, May, 1977. New York: ACM, 1977. Pp. 77-90.
[77]. Stroet J. W. M., Engmann R. Manipulation of expressions in a relational algebra. Information Systems, 1979. Volume 4, Issue 4. Pp. 195-203.
[78]. Aho V., Sagiv Y., Ullman J. D. Equivalences among relational expressions. Society for Industrial and Applied Mathematics Journal on Computing, 1979. Volume 8, Issue 2. Pp. 218-246.
[79]. Aho V., Sagiv Y., Ullman J. D. Efficient optimization of a class of relational expressions. Journal ACM Transactions on Database Systems (TODS), December 1979. Volume 4, Issue 4. Pp.435-454.
[80]. Sagiv Y., Yannakakis M. Equivalences among relational expressions with the union and difference operators. Journal of the ACM, October 1980. Volume 27, Issue 4. Pp. 633-655.
[81]. 7.2.1.2 How MySQL Optimizes WHERE Clauses http://dev.mysql.com/doc/refman/5.5/en/where-optimizations.html.
[82]. Prilozhenie PostgreSQL 8.3.3., http://www.postgresql.org/download/. Adres fajla, soderzhashhego obsuzhdaemyj kod, v arhive postgresql-8.3.3\src\backend\optimizer\util\predtest.c.
[83]. Veitch E. W. A Chart Method for Simplifying Truth Functions. ACM Annual Conference/Annual Meeting: Proceedings of the 1952 ACM Annual Meeting. Pittsburg: ACM, NY, 1952. Pp. 127-133.
[84]. Karnaugh M. The Map Method for Synthesis of Combinational Logic Circuits. Transactions of the American Institute of Electrical Engineers, November 1953. Part I, № 72 (9). Pp. 593-599.
[85]. Quin W. V. On Cores and Prime Implicants of Truth Functions. American Mathematics Monthly, 1959. V. 66, № 9. P. 755-760.
[86]. McCluskey E. J. Minimization of Boolean Functions. The Bell System Technical Journal, November 1956. V. 35, Issue 5. Pp. 1236-1249.
[87]. [87].Wu M.-C. Query Optimizatiom for Selecting Using Bitmaps. ACM SIGMOD Record, June 1999. Volume 28 Issue 2. Pp. 227-238.
[88]. Das Sarma A., Theobald M., Widom J. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. Technical Report. Stanford, 2007. http://ilpubs.stanford.edu:8090/800/.
[89]. Tarasenko P. F., Buharova M. F. Tehnologija «The Reporter» dlja postroenija otchetov po bazam dannyh [The Reporter technology for database reporting]. Vestnik Tomskogo Gosudarstvennogo Universiteta, № 275, aprel' 2002. S. 167-176 (in Russian).
[90]. Mendkovich N., Kuznetsov S. New Algorithms for Lexical Query Optimization. Proceedings of the 31st International Conference on Information Technology Interfaces. Cavtat/Dubrovnik, Croatia, June 22-25, 2009. Zagreb: University of Zagreb, 2009. Pp. 187-192.

[91]. Kuznetsov S. D., Mendkovich N. A. Novye algoritmy leksicheskoj optimizacii zaprosov [New Algorithms for Lexical Query Optimization]. Modeli i analiz informacionnyh sistem, 2009. T. 16, № 4. S. 22-33 (in Russian).
[92]. Mendkovich N. A., Kuznetsov S. D. Optimizacija kon#junktov uslovij v sostave zaprosov [Optimization of query condition' conjucts]. Modeli i analiz informacionnyh sistem, 2011. T. 18, № 3. S. 144-154 (in Russian).
[93]. Bellamkonda S., Ahmed R., Witkowski A., Amor A., Zait M., Lin Ch.-Ch. Enhanced Subquery Optimizations in Oracle /. Proceedings of the 35th international conference on Very large data base, August 2009. Volume 2 Issue 2. Pp. 1366-1377.
[94]. Chaudhuri S. Query Optimizers: Time to Rethink the Contract?. Proceedings of the ACM SIGMOD International Conference on Management of Data, Providence, Rhode Island, USA, June 29 - July 2, 2009. New York: ACM, 2009. Pp. 961-968.

Тематическое моделирование текстов на естественном языке

Антон Коршунов, Андрей Гомзин
{korshunov, gomzin}@ispras.ru

Аннотация. Тематическое моделирование — способ построения модели коллекции текстовых документов, которая определяет, к каким темам относится каждый из документов. Переход из пространства терминов в пространство найденных тематик помогает разрешать синонимию и полисемию терминов, а также эффективнее решать такие задачи, как тематический поиск, классификация, суммаризация и аннотация коллекций документов и новостных потоков. Наибольшее применение в современных приложениях находят подходы, основанные на Байесовских сетях — ориентированных графических вероятностных моделях, позволяющих учитывать авторство документов, связи между словами, темами, документами и авторами, а также другие типы сущностей и метаданных. В статье приведён сравнительный обзор различных моделей, описаны способы оценивания их параметров и качества результатов, а также приведены примеры открытых программных реализаций.

Ключевые слова: тематическое моделирование; тематический поиск; классификация документов; графические вероятностные модели; Байесовские сети; скрытое размещение Дирихле; уменьшение размерности; анализ текста; извлечение информации; машинное обучение.

1. Введение

В 1958 году Герхард Лисовски и Леонард Рост завершили работу по составлению каталога религиозных текстов на иврите, призванных помочь учёным определить значения терминов, которые были давно утрачены [20]. Путём кропотливой ручной работы они собрали воедино все возможные контексты, в которых появлялся каждый из терминов. Следующей задачей было научиться игнорировать несущественные различия в формах слов и выделять те различия, которые влияют на семантику. Замыслом авторов было дать возможность исследователям языка проанализировать различные отрывки и понять семантику каждого термина в его контексте.

Трудности, с которыми столкнулись Лисовски и Рост полвека назад, часто возникают и сегодня при автоматическом анализе текстов. Одна и та же концепция может выражаться любым количеством различных терминов (*синонимия*), тогда как один термин часто имеет разные смыслы в различных контекстах (*полисемия*). Таким образом, необходимы способы различать

варианты представления одной концепции и определять конкретный смысл многозначных терминов. Кроме того, нужно уметь представлять данные в доступной для человека форме, чтобы дать возможность понять неизвестный ему смысл термина. Теоретически обоснованным и активно развивающимся направлением в анализе текстов на естественном языке, призванным решать перечисленные задачи, является тематическое моделирование коллекций текстовых документов.

Построение тематической модели может рассматриваться как задача одновременной кластеризации документов и слов по одному и тому же множеству кластеров, называемых темами. В терминах кластерного анализа *тема* — это результат би-кластеризации, то есть одновременной кластеризации и слов, и документов по их семантической близости. Обычно выполняется *нечёткая кластеризация*, то есть документ может принадлежать нескольким темам в различной степени. Таким образом, сжатое семантическое описание слова или документа представляет собой вероятностное распределение на множестве тем. Процесс нахождения этих распределений и называется *тематическим моделированием* [27].

Как правило, количество тем, встречающихся в документах, меньше количества различных слов во всем наборе. Поэтому скрытые переменные — темы — позволяют представить документ в виде вектора в пространстве скрытых (*латентных*) тем вместо представления в пространстве слов. В результате документ имеет меньшее число компонент, что позволяет быстрее и эффективнее его обрабатывать. Таким образом, тематическое моделирование тесно связано с другим классом задач, известным как *уменьшение размерности* данных [14]. Кроме того, найденные темы могут использоваться для семантического анализа текстов.

Задача извлечения скрытых тем из коллекции текстовых документов имеет множество применений. Помимо кластеризации и классификации документов, найденные темы могут применяться для определения релевантности документа заданной теме или запросу, определения тематического сходства документа с другими документами и их фрагментами, построения тематических профилей авторов, разбиения документа на тематически однородные фрагменты и т.д.

В силу своей универсальности и расширяемости, современные способы тематического моделирования находят применение в широком спектре приложений [5, 21, 22, 27]:

- кластеризация, классификация, ранжирование, аннотирование и суммаризация отчётов, научных публикаций, переписки, блогов, студенческих работ и т.д.;
- тематический поиск документов и связанных с ними объектов: рисунков, авторов, организаций, журналов, конференций;

- фильтрация спама;
- рубрикация коллекций изображений, видео, музыки;
- поиск генетических паттернов в различных популяциях и определение пропорции этих паттернов у конкретного индивидуума;
- коллаборативная фильтрация в сервисах рекомендаций;
- построение тематических профилей пользователей форумов, блогов и социальных сетей для поиска тематических сообществ и определения наиболее активных их участников;
- анализ новостных потоков и сообщений из социальных сетей для определения актуальных событий реального мира и реакции пользователей на них.

Иными словами, тематическое моделирование позволяет автоматически систематизировать и реферировать электронные архивы такого масштаба, который человек не в силах обработать.

Дальнейшее изложение строится следующим образом. В разделе 2 вводится понятие векторного представления документов и описываются ранние подходы к поиску тем, основанные на кластеризации. Раздел 3 содержит описание метода латентно-семантического индексирования (LSI), который рассматривает исходный набор данных как матрицу «документ-термин» и использует матричные разложения для извлечения скрытых тем. В разделе 4 рассмотрено применение Байесовских сетей для тематического моделирования текстов на примерах вероятностного латентно-семантического индексирования (PLSI) и скрытого размещения Дирихле (LDA). Здесь же коротко описаны способы оценивания оптимальных значений параметров моделей для обучающего и тестового набора документов. Примеры более сложных вероятностных моделей, позволяющих устранить ограничения и недостатки первых подходов, приведены в разделе 5. В разделе 6 описаны основные способы оценки качества результатов тематического моделирования. Наконец, раздел 7 содержит примеры программных реализаций алгоритмов тематического моделирования.

2. Кластеризация и классификация документов

Задача *определения и отслеживания тем (Topic Detection and Tracking, TDT)* возникла в 1996-1997 годах. В работе [1] понятие темы тесно связано с понятием события: *тема* — это событие или действие вместе со всеми непосредственно связанными событиями и действиями. Задача заключается в извлечении событий из потока информации.

Для представления документов принято пользоваться *векторной моделью (Vector Space Model, VSM)*, в которой каждому слову сопоставляется вес в

соответствии с выбранной весовой функцией. Располагая таким представлением для всех документов, можно, например, находить расстояние между точками пространства и тем самым решать задачу подобия документов — чем ближе расположены точки, тем больше похожи соответствующие документы.

Классическим методом назначения весов словам является *TF-IDF*:

$$TFIDF(t,d,D) = TF(t,d) \times IDF(t,D) \tag{1}$$

TF *(term frequency)* — нормализованная частота слова в тексте:

$$TF(t,d) = \frac{freq(t,d)}{\max_{w \in D} freq(w,d)} \tag{2}$$

Здесь $freq(t,D)$ — число вхождений слова *t* в документе *d*.

IDF *(inverse document frequency)* — обратная частота документов:

$$IDF(t,D) = \log \frac{|D|}{|\{d \in D : t \in d\}|} \tag{3}$$

Здесь в числителе — количество документов в наборе, а в знаменателе — количество документов, в которых встречается слово *t*.

В зависимости от решаемой задачи используются различные модификации TF-IDF. Например, в одном из решений, описанных в [1] используются следующие веса:

$$w(t,D) = (1 + \log_2 TF(t,D)) \times \frac{IDF(t)}{\left\|\vec{d}\right\|} \tag{4}$$

Здесь $\left\|\vec{d}\right\|$ - норма вектора, представляющего документ *D*.

В более поздних исследованиях [4] использовались следующие модификации TF-IDF:

$$TF' = \frac{TF}{TF + 0.5 + 1.5\frac{len_d}{len_{avg}}} \tag{5}$$

Здесь len_d — длина документа *d*, len_{avg} — средняя длина документа.

$$IDF' = \frac{\log(IDF)}{\log(N+1)} \quad (6)$$

Для сравнения векторов документов в [1] и [4] применялись такие метрики, как косинус, дивергенция Кульбака-Лейблера и другие методы (взвешенная сумма компонентов документа, простые языковые модели). Всего существует более 70 способов расчёта похожести векторов [29].

В [1] рассматривается два типа задач: обнаружение событий из набора данных за определенный период времени и обнаружение событий в режиме реального времени.

Первый тип задач заключается в разбиении исходных данных на группы, соответствующие событиям, а также в определении, описывает ли текстовый документ из набора какое-либо событие. Основной идеей всех решений было использование алгоритмов кластеризации [2, 30] (*инкрементальная кластеризация, метод K-средних* и др). При этом предполагается, что каждый кластер содержит документы, описывающие какое-либо событие.

Задача второго типа — для нового документа определить, описывает ли он событие, которое уже встречалось в исходных данных. Для отслеживания событий использовались алгоритмы классификации [3] (*метод k-ближайших соседей, решающие деревья* и др). Классификация производилась с использованием двух классов: *YES* — документ описывает событие, *NO* — не описывает.

Таким образом, в ранних исследованиях тема отождествлялась с событием. В реальной жизни тема может описывать иные сущности, а не только события. Поэтому в более поздних работах задачи определения событий и тем стали различаться. Еще один недостаток описанных методов в том, что анализируемые документы относятся только к одной теме или событию. Однако один документ может затрагивать несколько тем. К тому же, векторное представление документов не позволяет разрешать синонимию и полисемию терминов (см. раздел 1).

Для решения перечисленных проблем было предложено рассматривать набор векторов терминов из документов как общую терм-документную матрицу и применять к ней особые разложения (метод LSI).

3. Латентно-семантическое индексирование

80-е годы прошлого столетия ознаменовались активным развитием систем информационного поиска по коллекциям документов разнообразной природы. Следуя принципу «от простого к сложному», первыми были реализованы подходы, основанные на поиске точных совпадений частей документов с запросами пользователей. Довольно скоро, однако, стало очевидно различие между *релевантностью* (соответствием) документа запросу и точным

совпадением их частей. Зачастую документы, релевантные запросу с точки зрения пользователя, не содержали терминов из запроса и поэтому не отображались в результатах поиска (проблема синонимии). С другой стороны, большое количество документов, слабо или вовсе не соответствующих запросу, показывались пользователю только потому, что содержали термины из запроса (проблема полисемии).

Самым простым решением этих проблем кажется добавление к запросу *уточняющих* терминов для более точного описания интересующего контекста. Однако предположение о том, что индекс поисковой системы содержит все возможные уточняющие термины, на практике выполняется довольно редко.

В 1988 г. Dumais et al [36] предложили метод *латентно-семантического индексирования (latent semantic indexing, LSI)*, призванный повысить эффективность работы информационно-поисковых систем путём проецирования документов и терминов в пространство более низкой размерности, которое содержит *семантические концепции* исходного набора документов.

Основная идея метода состоит в оценивании корреляции терминов путём анализа их совместной встречаемости в документах. К примеру, в коллекции всего 100 документов, содержащих термины «*доступ*» и/или «*поиск*». При этом только 95 из них содержат оба термина вместе. Логично предположить, что отсутствие термина «*поиск*» в документе с термином «*доступ*» ошибочно и возвращать данный документ по запросу, содержащему только термин «*доступ*». Разумеется, подобные выводы можно делать не только из простой попарной корреляции терминов.

С другой стороны, анализируя корреляцию терминов в запросе, можно более точно определять интересующий пользователя смысл основного термина и повышать позиции документов, соответствующих этому смыслу, в результатах поиска.

Таким образом, при латентно-семантическом индексировании документов задача состоит в том, чтобы спроецировать часто встречающиеся вместе термины в одно и то же измерение семантического пространства, которое имеет пониженную размерность по сравнению с оригинальной *терм-документной матрицей*, которая обычно довольно разрежена. Элементы этой матрицы содержат веса терминов в документах, назначенные с помощью выбранной весовой функции (см. раздел 2). В качестве примера можно рассмотреть самый простой вариант такой матрицы, в которой вес термина равен 1, если он встретился в документе (независимо от количества появлений), и 0 если не встретился (рис. 1).

	d_1	d_2	d_3	d_4	d_5	d_6
ship	1	0	1	0	0	0
boat	0	1	0	0	0	0
ocean	1	1	0	0	0	0
voyage	1	0	0	1	1	0
trip	0	0	0	1	0	1

Рис. 1. Терм-документная матрица.

Наиболее распространенный вариант LSI основан на использовании разложения терм-документной матрицы по сингулярным значениям — так называемом *сингулярном разложении (Singular Value Decomposition, SVD)*, которое хорошо зарекомендовало себя в факторном анализе.

Согласно *теореме о сингулярном разложении*, любая вещественная прямоугольная матрица может быть разложена на произведение трех матриц:

$$A = TSD^T, \tag{7}$$

где матрицы T и D — ортогональные, а S — диагональная матрица, элементы на диагонали которой называются *сингулярными значениями* матрицы A.

Такое разложение обладает замечательной особенностью: если в матрице S оставить только k наибольших сингулярных значений, а в матрицах T и D — только соответствующие этим значениям столбцы, то произведение получившихся матриц S, T и D будет наилучшим приближением исходной матрицы A к матрице $\hat{A}$ ранга k.

Если в качестве матрицы A взять терм-документную матрицу, то матрица $\hat{A}$, содержащая только k первых линейно независимых компонент A, отражает основную структуру различных зависимостей, присутствующих в исходной матрице. Структура зависимостей определяется весовыми функциями терминов.

Таким образом, каждый термин и документ представляются при помощи векторов в общем семантическом пространстве размерности k. Близость между любой комбинацией терминов и/или документов легко вычисляется при помощи скалярного произведения векторов. Для задач информационного поиска запрос пользователя рассматривается как набор терминов, который

проецируется в семантическое пространство, после чего полученное представление сравнивается с представлениями документов в коллекции.

Как правило, выбор k зависит от поставленной задачи и подбирается эмпирически. Если выбранное значение k слишком велико, то метод теряет свою мощность и приближается по характеристикам к стандартным векторным методам. Слишком маленькое значение k не позволяет улавливать различия между похожими терминами или документами.

В качестве альтернативы SVD в тематическом моделировании [37] также применяется *неотрицательная матричная факторизация (non-negative matrix factorization, NMF)*. Данный способ разложения матриц накладывает ограничение на результирующие матрицы (факторы): все их элементы должны быть положительными либо нулевыми [38].

Следующим этапом развития тематического моделирования стали подходы, позволяющие моделировать *вероятности* скрытых тем в документах и терминов в темах (см. раздел 4). В результатах работы LSI эти вероятности для каждой темы и документа распределены равномерно, что не соответствует характеристикам реальных коллекций документов. В отличие от так называемых *дискриминативных* подходов (к которым относится LSI), в вероятностных подходах сначала задаётся модель, а затем с помощью терм-документной матрицы оцениваются её скрытые параметры, которые затем могут быть использованы для генерации моделируемых распределений. Из этого следуют преимущества вероятностного моделирования документов:

- результаты работы представляются в терминах теории вероятностей и поэтому могут быть с минимальными затратами встроены в другие вероятностные модели и проанализированы стандартными статистическими методами;
- новые порции входных данных не требуют повторного обучения модели;
- использование Байесовского непараметрического моделирования позволяет избежать подбора входных параметров, что делает такие модели более гибкими;
- вероятностные модели могут быть с лёгкостью расширены путём добавления переменных, а также новых связей между наблюдаемыми и скрытыми переменными.

4. Вероятностные тематические модели

Вероятностное тематическое моделирование — это набор алгоритмов, позволяющих анализировать слова в больших наборах документов и извлекать из них темы, связи между темами и изменение их во времени [5]. При этом

документ рассматривается как набор слов, порядок которых не имеет значения. Для каждого документа определено распределение θ_d его слов по темам, т.е. вероятность θ_d^t для каждой темы встретить ее в данном документе, причём $\sum_t \theta_d^t = 1$. В документе на рис. 2 это распределение изображено справа в виде гистограммы.

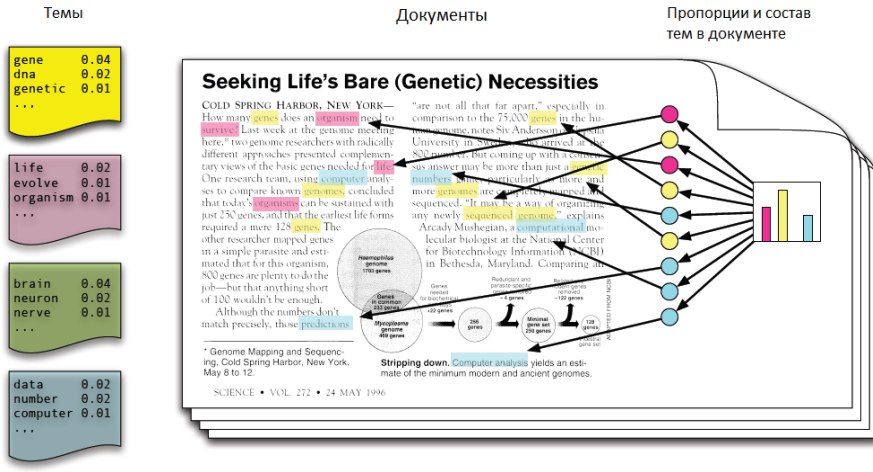


Рис. 2. Интуитивное представление тем и тематических моделей.

Тема представляется в виде распределения φ_t слов из фиксированного словаря, т.е. каждое слово входит в тему с некоторой вероятностью φ_t^w, причём $\sum_w \varphi_t^w = 1$. На рис. 2 распределения φ_t изображены слева.

Вероятностные модели являются *генеративными (порождающими)*, то есть их можно использовать для генерации документов. Описание модели, как правило, начинается со способа генерации документов — *генеративного процесса*. Однако основной целью тематического моделирования является не генерация, а извлечение тем из имеющегося набора документов. То есть нужно определить скрытые структуры (темы, распределение тем в каждом документе и т.д.) при данных словах в документах. Другими словами, это задача, обратная генерации: выяснить, с помощью каких скрытых структур вероятнее всего могли бы быть сгенерированы исходные документы.

Задача *оценивания модели* заключается в том, чтобы найти значения параметров модели, при которых наблюдаемая обучающая выборка максимально правдоподобна. Задача *вывода по модели* состоит в определении значений скрытых переменных (например, скрытых вероятностей тем) для нового документа, изначально не входившего в состав обучающей выборки. Поскольку эти задачи отличаются лишь исходными данными и поэтому часто решаются похожими способами, то далее будет использоваться общий термин *оценивание параметров модели* для обеих задач.

Вводные обзоры по вероятностному тематическому моделированию даны в [5, 26, 27]. В них рассмотрены многочисленные модификации ранних моделей, используемые для современных приложений информационного поиска. Текущий и последующие разделы призваны дополнить предыдущие обзоры описанием недавно предложенных подходов к построению тематических моделей, а также оцениванию их параметров и качества результатов.

4.1. Графические модели. Плоское графическое представление

Вероятностные тематические модели по своей природе являются графическими и поэтому часто представляются в виде интуитивно понятного и наглядного *плоского графического представления* [28]. Каждая случайная величина обозначается кругом. Наблюдаемые величины (значения которых известны) закрашиваются, скрытые (значения которых надо найти) остаются незакрашенными. Стрелка (направленное ребро) из первой вершины во вторую обозначает условную зависимость второй величины от первой.

Зависимость величины Y от величины X означает, что совместную вероятность $P(X,Y)$ можно представить в виде $P(Y|X)P(X)$. Если X и Y независимы, то $P(X,Y) = P(X)P(Y)$.

Прямоугольник, включающий в себя некоторый подграф с указанным в правом нижнем углу числом N, обозначает совокупность N экземпляров данного подграфа. Прямоугольники могут быть вложенными.

Пример модели представлен на рис. 3. Здесь изображены величины и зависимости. Величины c и w — наблюдаемые, остальные — скрытые. В модели присутствует C величин α, D величин φ, MD величин c и θ, ND величин w, y, z. Стрелками показаны зависимости: например, z зависит от соответствующей y и от всех θ.

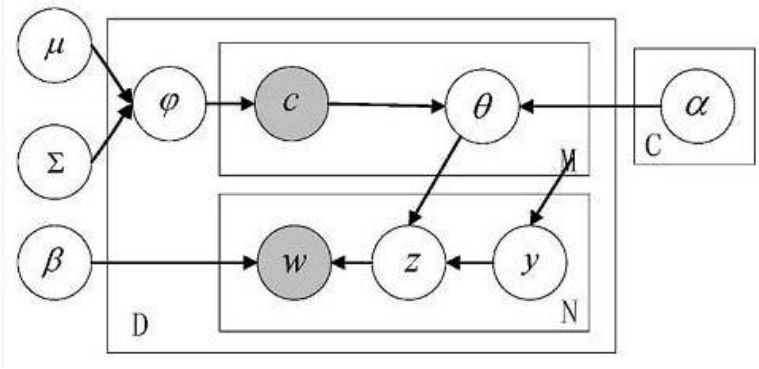


Рис. 3. Пример графической вероятностной модели.

4.2. Вероятностное латентно-семантическое индексирование

Одной из первых вероятностных тематических моделей является *вероятностное латентно-семантическое индексирование (Probabilistic Latent Semantic Indexing, PLSI),* предложенное Томасом Хоффманом в 1999 году [6, 7].

В основе PLSI лежит т.н. *аспектная модель*, которая связывает скрытые переменные тем $z \in Z = \{z_1, \dots, z_k\}$ с каждой наблюдаемой переменной — словом или документом. Таким образом, каждый документ может относиться к нескольким темам с некоторой вероятностью, что является отличительной особенностью этой модели по сравнению с подходами, не позволяющими вероятностного моделирования.

Генеративный процесс следующий:

1. Выбрать документ d согласно распределению $p(d)$
2. Выбрать тему $i \in \{1, \dots, k\}$ на основе распределения $\theta_{di} = p(z = i|d)$
3. Выбрать слово *v* — значение переменной *w* на основе распределения $\varphi_{iv} = p(w = v|z = i)$

Совместная вероятностная модель над документами и словами определена следующим образом:

$$P(d,w) = P(d)\sum_{z \in Z} P(w \mid z)P(z \mid d) \tag{8}$$

Также модель может быть представлена в виде:

$$P(d,w) = \sum_{z \in Z} P(z)P(w \mid z)P(d \mid z) \tag{9}$$

Представление (8) является асимметричным, представление (9) — симметричным (рис. 4).

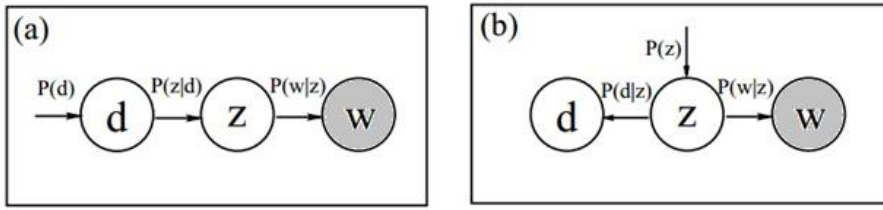


Рис. 4. Графическое представление модели вероятностного латентно-семантического индексирования с асимметричной (a) и симметричной (b) параметризацией.

4.2.1 Оценивание параметров модели вероятностного латентно-семантического индексирования

Для определения оптимальных значений скрытых параметров модели используется стандартная процедура оценки максимального правдоподобия — *EM-алгоритм (Expectation Maximization)* [8]. Он применяется к симметричному представлению модели.

На E-шаге алгоритма оценивается вероятность $P(z|d,w)$:

$$P(z \mid d, w) = \frac{P(z)[P(d \mid z)P(w \mid z)]^{\beta}}{\sum_{z' \in Z} P(z')[P(d \mid z')P(w \mid z')]^{\beta}} \tag{10}$$

где $\beta<1$ — задаваемый параметр [6, 7].

На M-шаге алгоритма вычисляются:

$$P(w \mid z) = \frac{\sum_{d} n(d, w)P(z \mid d, w)}{\sum_{d, w'} n(d, w')P(z \mid d, w')} \tag{11}$$

$$P(d \mid z) = \frac{\sum_{w} n(d, w)P(z \mid d, w)}{\sum_{d', w} n(d', w)P(z \mid d', w)} \tag{12}$$

$$P(z) = \frac{\sum_{d, w} n(d, w)P(z \mid d, w)}{\sum_{d, w} n(d, w)} \tag{13}$$

Несмотря на очевидные преимущества перед более ранними подходами, модель PLSI не лишена недостатков. Во-первых, она содержит большое число параметров, которое растет в линейной зависимости от числа документов. Как следствие, модель склонна к переобучению и неприменима к большим наборам данных. Во-вторых, невозможно вычислить вероятность документа, которого нет в наборе данных. В-третьих, отсутствует какая-либо закономерность при генерации документов из сочетания полученных тем. Данные недостатки устранены в модели LDA.

4.3. Скрытое размещение Дирихле

Скрытое размещение Дирихле (Latent Dirichlet Allocation, LDA) — генеративная графическая вероятностная модель, предложенная Дэвидом Блеем и соавторами в 2003 году [9]. Процесс генерации документа похож на генеративный процесс в PLSI. Каждый документ генерируется независимо:

1. Случайно выбрать для документа его распределение по темам θ_d

2. Для каждого слова в документе:
    a. Случайно выбрать тему из распределения θ_d, полученного на 1-м шаге
    b. Случайно выбрать слово из распределения слов в выбранной теме φ_t

Схема модели скрытого размещения Дирихле изображена на рис. 5.

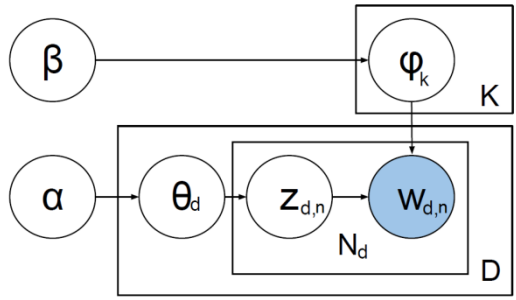


Рис. 5. Графическое представление модели скрытого размещения Дирихле.

Имеется набор из D документов. Каждый документ состоит из N_d слов, w_{dn} соответствует наблюдаемым переменным — словам в документе. Это единственные наблюдаемые переменные в модели, остальные переменные — скрытые. Переменная z_{dn} принимает значение темы, выбранной на шаге 2а для слова w_{dn}. Для каждого документа d переменная θ_d представляет собой распределение тем в этом документе.

В классической модели LDA количество тем фиксировано изначально и задаётся в явном виде параметром K. φ_k — распределение слов в теме k. Можно подобрать оптимальное значение K, варьируя его и измеряя способность модели предсказывать неизвестные данные, например, документы из тестовой выборки (см. раздел 6.1). Однако для того чтобы определять оптимальное количество тем в документе автоматически, были предложены более совершенные способы (см. раздел 5.2).

В модели LDA предполагается, что параметры θ_d и φ_t распределены следующим образом: $\theta \sim Dir(\alpha)$, $\varphi \sim Dir(\beta)$, где α и β — задаваемые вектора-параметры (т.н. *гиперпараметры*) распределения *Дирихле*. Из Байесовской теории вероятностей известно, что распределение Дирихле является *сопряжённым априорным* к *мультиномиальному* распределению, которое обычно используется для моделирования текстов. Знание сопряжённых семейств распределений существенно упрощает вычисление апостериорных вероятностей при оценке параметров модели (см. раздел 4.3.1).

Как правило, все компоненты параметров α и β распределения Дирихле берутся равными, поскольку отсутствует априорная информация о распределении слов в темах и тем в документах. Предложены подходы, позволяющие восстановить оптимальные значения гиперпараметров модели по обучающей выборке [10, 24]. На практике, как правило, используются значения, наиболее характерные для конкретных данных. К примеру, значение параметра, близкое к нулю, позволяет после оценивания параметров модели получить мультиномиальное распределение, в котором в котором б*о*льшая часть плотности вероятности сосредоточена на небольшом наборе значений. Это хорошо соотносится со *степенным* распределением, которое часто наблюдается в текстах на естественном языке.

4.3.1 Оценивание параметров модели скрытого размещения Дирихле

Генеративный процесс LDA соответствует следующему совместному распределению наблюдаемых и скрытых переменных [5]:

$$P(\varphi_{1:K}, \theta_{1:D}, z_{1:D}, w_{1:D}) = \prod_{i=1}^{K} P(\varphi_i) \prod_{d=1}^{D} P(\theta_d) (\prod_{n=1}^{N} P(z_{dn} \mid \theta_d)\, p(w_{dn} \mid \varphi_{1:K}, z_{dn})) \qquad (14)$$

Для определения оптимальных значений скрытых переменных модели нужно найти так называемое апостериорное распределение, т.е. условное распределение:

$$P(\varphi_{1:K}, \theta_{1:D}, z_{1:D} \mid w_{1:D}) = \frac{P(\varphi_{1:K}, \theta_{1:D}, z_{1:D}, w_{1:D})}{P(w_{1:D})} \qquad (15)$$

Числитель дроби (15) легко вычисляется согласно (14), знаменатель представляет собой маргинальную вероятность наблюдаемых переменных, т.е. вероятность наблюдать *w* при всех возможных параметрах модели. Теоретически он вычисляется как сумма вероятностей совместного распределения по всем значениям скрытых переменных. Но число всевозможных назначений тем *z* словам *w* экспоненциально зависит от размера документа, поэтому на практике используются другие методы для оценивания (15).

Алгоритмы оценивания (15) делятся на две категории: на основе сэмплирования и вариационные методы. Алгоритмы первой группы пытаются собрать конечную выборку переменных, чтобы приблизить апостериорное распределение (15) эмпирическим распределением. Как правило, алгоритм принадлежит классу *методов Монте-Карло для марковских цепей (Markov Chain Monte Carlo, MCMC)*. Примером такого алгоритма является *сэмплирование по Гиббсу* [10], которое состоит в том, чтобы на каждом шаге фиксировать все переменные, кроме одной, и выбирать оставшуюся переменную согласно распределению вероятности этой переменной при

условии всех остальных (эта вероятность выводится в [10]). Недостаток этих методов в том, что они недетерминированы, так как выборки берутся случайно. Кроме того, марковская цепь может неопределённо долго сходиться к нужному распределению.

Методы второй группы — вариационные алгоритмы, детерминированная альтернатива методам на основе сэмплирования. Такие алгоритмы сначала задают параметризованное семейство распределений над скрытыми переменными, а затем с помощью EM-алгоритма ищут распределение из этого семейства, наиболее близкое к апостериорному распределению (15). Таким образом, задача оценивания параметров сводится к задаче оптимизации. При этом сознательно игнорируются некоторые зависимости между переменными. На рис. 6 приведен пример модели LDA для вариационного оценивания.

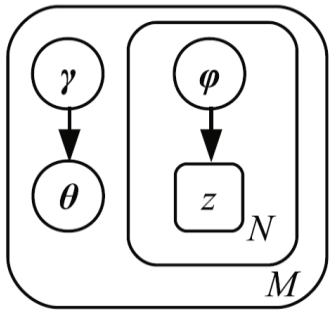


Рис. 6. Вариационная модель для оценивания параметров скрытого размещения Дирихле.

5. Второе поколение вероятностных тематических моделей

Модели, описанные ранее, применимы лишь к задачам, для которых верны следующие предположения об исходных данных [5]:

1. Последовательность слов в документе не имеет значения
2. Последовательность документов не имеет значения
3. Количество тем известно и не меняется

Если же какие-либо из данных условий не удовлетворяют поставленной задаче, то требуются более сложные модели. Например, при генерации документов, понятных человеку, последовательность слов в генерируемом документе имеет большое значение. Второе предположение может быть неверным при анализе тем в документах из большого временного промежутка. Например, тема, описывающая какую-нибудь научную область, может иметь разное распределение и состав в разные промежутки времени: с течением

времени из-за смены приоритетов и терминологии какие-то термины начинают встречаться чаще, а какие-то реже. Третье предположение работает только в том случае, если в задаче априорно известно количество тем в документах, что на практике выполняется редко.

В данном разделе рассмотрены модификации LDA и другие модели, позволяющие снять некоторые из перечисленных ограничений и расширить область применения тематического моделирования. Обзоры разнообразных подходов, предложенных за 10 лет активного развития тематического моделирования, приведены в [5, 26, 27].

5.1. Иерархическое скрытое размещение Дирихле

Одним из недостатков скрытого размещения Дирихле является тенденция к извлечению слишком общих тем для заданного набора документов. В случае, когда некоторая концепция имеет ряд аспектов (смыслов), которые часто употребляются совместно с основной концепцией, в результатах работы LDA с большой вероятностью будет присутствовать тема, которая включает как основной смысл концепции, так и все её аспекты. Часто необходимо, чтобы в отдельные темы была выделена не только основная концепция, но и различные её аспекты. В таких случаях используются иерархические тематические модели, позволяющие моделировать иерархию тем — от более общих до узких.

Модель иерархического скрытого размещения Дирихле (Hierarchical Latent Dirichlet Allocation, hLDA), описанная в работе [11], основана на вложенном процессе китайского ресторана (Nested Chinese Restaurant Process, nCRP). Логично сначала рассмотреть стандартный процесс китайского ресторана (Chinese Restaurant Process, CRP), который генерирует распределение M объектов (клиентов) по неограниченному числу разделов (столов).

Пусть некоторый китайский ресторан имеет неограниченное (счетное) количество столов. В него по очереди заходят M клиентов. Первый клиент садится за первый стол. Очередной клиент с номером m выбирает стол согласно распределению:

$$p(\textit{занятый стол}\, i \mid \textit{клиенты}\, \overline{1, \mathrm{m}-1}) = \frac{m_i}{\gamma + m - 1}$$
$$p(\textit{первый свободный стол} \mid \textit{клиенты}\, \overline{1, \mathrm{m}-1}) = \frac{\gamma}{\gamma + m - 1} \qquad (16)$$

Здесь γ - так называемый *концентрационный параметр* процесса.

Таким образом, если клиент садится за занятый стол, то с большей вероятностью он занимает стол с большим количеством клиентов, с каждым клиентом уменьшается вероятность занять новый стол. Причем распределение

объектов (клиентов) по разделам (столам) получается такое же, как из процесса Дирихле (см. раздел 5.2).

Процесс китайского ресторана можно расширить до вложенного процесса китайского ресторана [11]. Пусть в городе имеется бесконечное (счетное) число ресторанов. Каждый стол в ресторане содержит ссылку на другой ресторан. Пусть имеется один корневой ресторан, и в каждый ресторан ведет только одна ссылка. Таким образом, получается древовидная структура ресторанов.

Клиент прибывает в город на L дней. В первый вечер посещает корневой ресторан, выбирая стол согласно (16). На следующий день он идет в ресторан, определенный выбранным в корневом ресторане столом, снова выбирает стол согласно (16) и так далее. Каждый день клиент посещает один из ресторанов. Таким образом, он посетит L ресторанов. После того, как город посетят M клиентов, коллекция их путей по ресторанам будет представлять конечное поддерево глубины L бесконечного дерева ресторанов.

Полученное дерево может быть использовано для моделирования иерархии тем. В модели иерархического скрытого размещения Дирихле [11] каждому ресторану из процесса китайского ресторана соответствует тема. Генеративный процесс следующий:

1. Пусть c_1 — корневой ресторан
2. Для каждого уровня дерева $l \in \{2, \dots, L\}$:
    a. Выбрать стол в ресторане c_{l-1} согласно (16). Установить c_l — ресторан, на который ссылается выбранный стол
3. Случайно выбрать для документа его распределение по L темам $\theta_d \sim Dir(\alpha)$
4. Для каждого слова в документе:
    a. Случайно выбрать $z \in \{1, \dots, L\}$ согласно распределению θ_d
    b. Случайно выбрать слово из распределения слов в теме, соответствующему ресторану c_z

Схема модели hLDA изображена на рис. 7. Здесь T — дерево ресторанов, получаемое с помощью вложенного процесса китайского ресторана, $c_1, c_2, \dots, c_L$ — путь по ресторанам, причем значение c_i зависит от $c_1, c_2, \dots, c_{i-1}$. Способ оценивания параметров модели описан в [11].

Описанная модификация расширяет модель LDA, добавляя возможность существования неограниченного количества тем. Однако количество тем, описывающих один документ, по-прежнему постоянно и равно L.

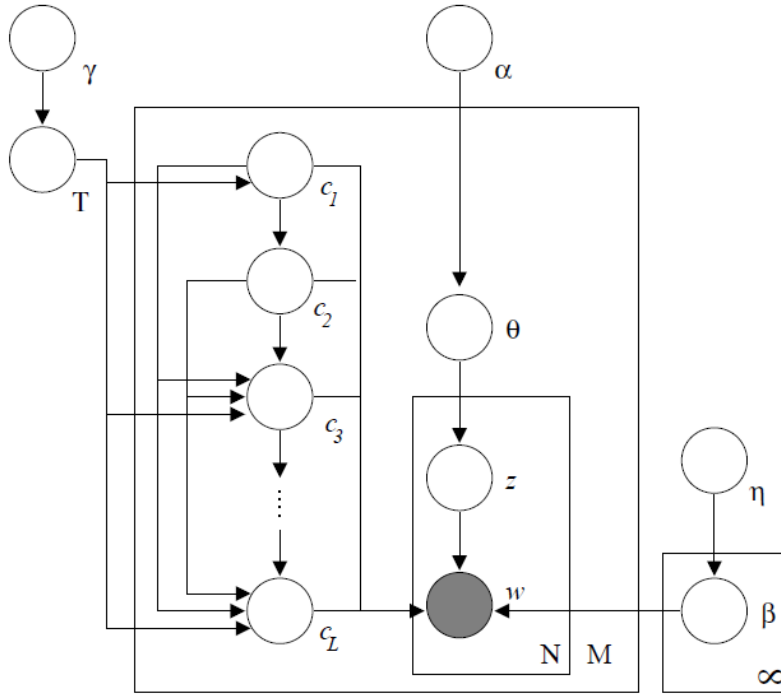


Рис. 7. Графическое представление модели иерархического скрытого размещения Дирихле.

5.2. Непараметрические модели

Зачастую невозможно предсказать, сколько тем встретится в наборе данных, например, при анализе потока неизвестных документов, новостей и т.д. Поэтому в этих случаях используются непараметрические модели.

Иерархический процесс Дирихле (Hierarchical Dirichlet Process, HDP) является Байесовской непараметрической моделью [12, 13], которая может быть использована для тематического моделирования с потенциально бесконечным числом тем.

Случайный процесс $G \sim DP(\alpha, H)$ называется *процессом Дирихле* с *базовым распределением* H и *параметром концентрации* α, если для произвольного конечного разбиения $A_1, A_2, \ldots, A_r$ вероятностного пространства над H выполняется:

$$(G(A_1), G(A_2), \ldots, G(A_r)) \sim Dir(\alpha H(A_1), \alpha H(A_2), \ldots, \alpha H(A_r)) \qquad (17)$$

Здесь $G(Ai)$ и $H(Ai)$ — маргинальные вероятности G и H над A_i.

Процесс Дирихле может быть представлен как процесс «*ломания палки*». Интуитивная интерпретация процесса предполагает, что имеется палка длины 1. Сначала ломают ее в точке β_1 (из бета-распределения (18) с параметром α) и получают пропорцию π_1. Затем повторяют процесс для оставшейся 1- β_1 части палки, получая π_2, π_3 и т.д. Пропорция π_k кластера k определена следующим образом:

$$\beta_k \sim Beta(1,\alpha)$$
$$\pi_k = \beta_k \prod_{l=1}^{k-1}(1-\beta_l) \qquad (18)$$

Иерархический процесс Дирихле можно понимать как процесс Дирихле над процессом Дирихле. Иначе говоря, при генерации случайного элемента из иерархического процесса Дирихле сначала выбирается, из какого процесса Дирихле верхнего уровня следует генерировать элемент, после чего к выбранному процессу Дирихле применяется стандартная схема генерации элемента. Он моделирует документы, в которых имеются некоторые темы, общие для всего набора данных, а также более специфичные темы. В работе [13] предложен способ онлайнового вариационного оценивания параметров процесса Дирихле.

5.3. Модели, учитывающие временной фактор

С течением времени темы могут появляться, изменяться и исчезать. Для решения задач, где нужно не только найти темы, но и проследить их динамику, используются темпоральные тематические модели.

В данном разделе рассмотрены две модели: в первой время делится на отрезки, каждый отрезок рассматривается как атомарная единица, во второй время рассматривается как непрерывная величина.

Динамическая тематическая модель, предложенная в [15], отслеживает эволюцию тем в последовательно организованном корпусе документов. В этой статье документы группируются по годам (корпус содержит документы за 100 лет), и документы каждого года генерируются согласно темам, произошедшим от тем из прошлого года.

Схема предложенной модели представлена на рис. 8. Для каждого интервала времени имеются свои распределения тем в документе θ и распределения слов в темах β. Причем эти распределения зависят от соответствующих распределений в предыдущем временном отрезке. Генеративный процесс следующий:

1. Сгенерировать темы $\beta_t|\beta_{t-1} \sim \mathcal{N}(\beta_{t-1}, \sigma^2 I)$

2. Сгенерировать $\alpha_t|\alpha_{t-1}\sim\mathcal{N}(\alpha_{t-1},\sigma^2 I)$
3. Для каждого документа:
    a. Сгенерировать распределение $\theta\sim\mathcal{N}(\alpha_t,a^2 I)$
    b. Для каждого слова сгенерировать тему из θ и слово из соответствующего β

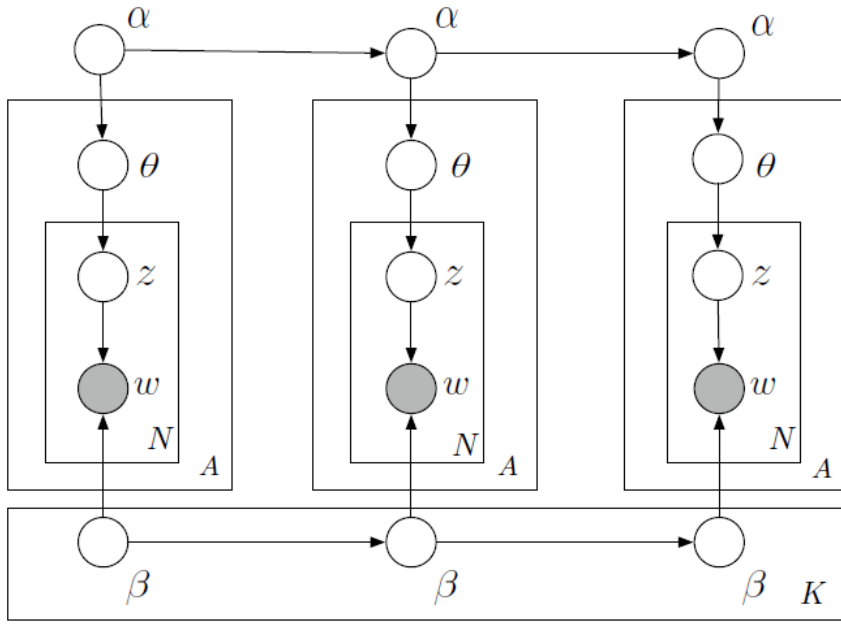


Рис. 8. Графическое представление динамической тематической модели.

В отличие от описанной выше модели, в работе [16] документы не делятся на дискретные группы, т.е. временной фактор непрерывный. Оценивание скрытых параметров модели позволяет найти темы, которые учитывают как одновременную встречаемость слов (как в стандартном LDA), так и локальность слов во времени.

В стандартную модель LDA для каждого документа добавляется наблюдаемая переменная — время t, которое зависит от темы и от переменной ψ, которая служит параметром априорного распределения t.

Процесс генерации аналогичен модели LDA. Основное отличие в том, что при генерации каждого слова генерируется переменная t из бета-распределения ψ. Схема модели изображена на рис. 9.

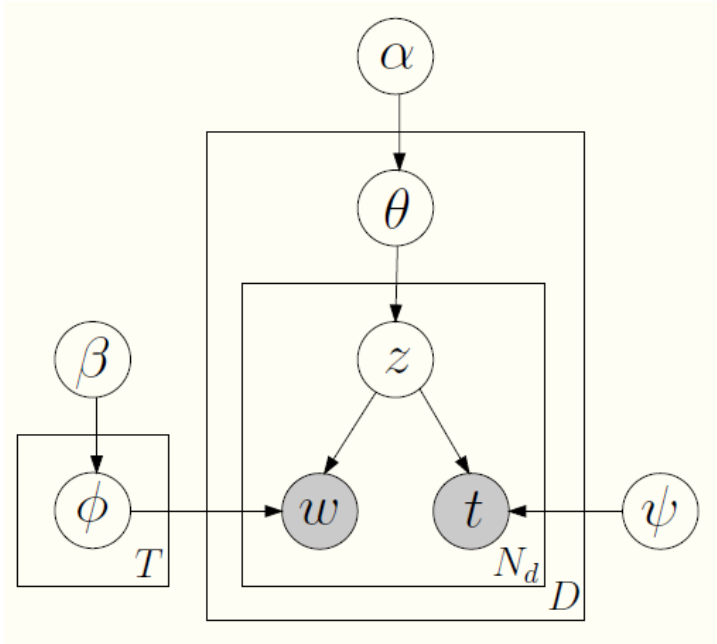


Рис. 9. Графическое представление тематической модели с учетом непрерывного временного фактора.

5.4. Онлайн-модели

В современном мире информация все чаще представляется в виде потоков. Поэтому необходимы алгоритмы, работающие не с фиксированным набором данных, а с данными, обновляющимися в режиме реального времени. Стандартная модель LDA не подходит для задачи поиска тем в режиме реального времени, потому что при появлении в потоке нового документа нужно полностью пересчитывать параметры модели на всех данных.

В работе [17] рассматривается онлайновое оценивание параметров модели LDA, при котором каждый документ обрабатывается один раз. Метод представляет собой модификацию вариационного оценивания. Все документы разбиваются на небольшие порции. Запускается итерационный процесс, на каждом шаге которого обрабатываются только данные из соответствующей порции и вычисляется оптимальный параметр постулируемого апостериорного распределения, задаваемого вариационным алгоритмом. После каждого шага этот параметр обновляется с учетом значения, вычисленного на предыдущей порции. Как правило, параметр вычисляется как взвешенное среднее текущего и предыдущего значений.

Кроме вариационного онлайнового оценивания параметров LDA существуют методы, являющиеся онлайн-модификациями сэмплирования по Гиббсу. В

работе [18] рассматривается алгоритм сэмплирования по Гиббсу, который выбирает значение темы z_i для слова w_i один раз для каждого слова. При этом сначала вычисляются параметры z для первых нескольких документов согласно стандартному алгоритму. А затем запускается итерационный процесс: для каждого последующего слова w_i выбирается назначение темы z_i, при условии значений z для предыдущих слов. Существенной проблемой данного алгоритма является зависимость от качества работы первого этапа — сэмплирования тем для первых документов, — так как все последующие темы выбираются на их основе. В связи с этим также рассматривается модификация алгоритма, в которой после сэмплирования темы для очередного слова для некоторых предыдущих слов производится повторное сэмплирование, то есть z_i выбирается заново.

5.5. Модели, учитывающие пользовательские метки

Многие социальные сервисы позволяют пользователям реагировать на объекты или оставлять дополнительную информацию. Эта информация может быть представлена в разных формах: в виде комментариев, оценок, меток и др. Ее можно использовать для улучшения качества результатов алгоритмов, работающих с содержимым социальных сетей.

В работе [19] рассматривается модификация LDA, которая учитывает метки (или *тэги*), которые пользователи назначают текстовым объектам.

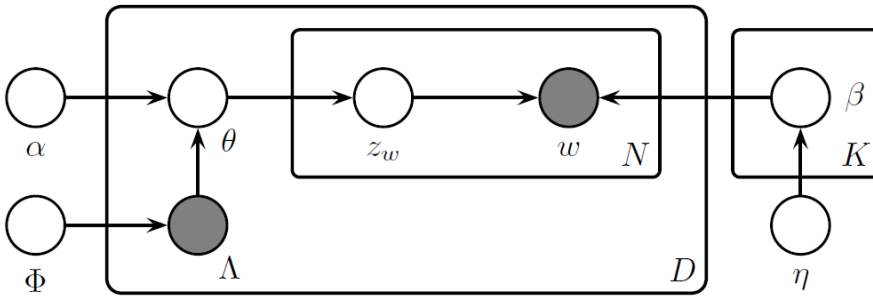


Рис. 10. Графическое представление тематической модели с учетом меток.

Схема предложенной модели представлена на рис. 10. Основная её особенность в том, что для каждого документа появляется наблюдаемая переменная Λ, которая соответствует тэгам пользователей. Λ — это бинарный вектор размерности K, где K — количество различных тэгов на всем наборе данных. При этом i-я компонента Λ равна 1, если документ помечен соответствующим i-м тэгом, 0 — в противном случае. В данной модели темы отождествляются с тэгами, поэтому количество различных тэгов совпадает с

количеством тем K. Распределение тем в документе θ зависит не только от априорного распределения, но и от наблюдаемых меток: в каждом документе вычисляется распределение только по тем темам, которые соответствуют тэгам с $\Lambda_i=1$, для остальных тем вероятность равна 0. Для оценивания скрытых параметров в работе [19] используется сэмплирование по Гиббсу, в котором каждая переменная z_w выбирается только из соответствующего набора тем.

6. Методы оценивания качества результатов

6.1. Обобщающая способность модели

Самым распространённым способом оценивания качества вероятностных тематических моделей является расчёт *перплексии* [31] на тестовом наборе данных D_{test} из M документов:

$$\text{Перплексия}(D_{test}) = \exp\left\{-\frac{\sum_{d=1}^{M}\log p(w_d|\text{модель})}{\sum_{d=1}^{M} N_d}\right\} \qquad (19)$$

Способы расчёта вероятности нового документа w_d при условии известных параметров модели рассмотрены в работе [32].

Если немного изменить способ расчёта перплексии и оценивать вероятности для каждого слова w_{dn} из тестового набора документов:

$$\text{Перплексия}(D_{test}) = \exp\left\{-\sum_{d=1}^{M}\sum_{n=1}^{N_d}\log p(w_{dn}|\text{модель})\right\}, \qquad (20)$$

то полученное значение соответствует полезному размеру словаря модели. Например, значение 100 означает, что набор вероятностей полученной модели эквивалентен случайному выбору каждого слова из словаря размером в 100 слов [14]. Таким образом, меньшее значение перплексии означает, что модель лучше описывает (обобщает) тестовые данные. Кроме того, минимизируя значение этого критерия, можно экспериментально подобрать оптимальное число различных тем в коллекции документов.

Альтернативным подходом является оценивание вероятности второй части документа при условии наличия первой [35]. Для этого каждый документ разделяют на 2 части: первую часть считают обучающими данными, а с помощью второй тестируют качество модели.

6.2. Эффективность приложений

Тестирование моделей на уровне приложений позволяет оценить их применимость к конкретной задаче и данным. К примеру, в работе [21] приведены результаты использования 4 различных тематических моделей в задаче фильтрации спама, а также сформулированы рекомендации для их применения к разным наборам данных. Авторы [25] исследовали

производительность моделей LSI и LDA в сочетании с популярным алгоритмом классификации *метод опорных векторов (Support Vector Machine, SVM)* в применении к классической задаче классификации документов. Продемонстрировано значительное увеличение точности классификации с использованием тематических моделей по сравнению с обычным векторным представлением.

6.3. Интерпретируемость

Приложения, которые предполагают непосредственное взаимодействие пользователя с результатами тематического моделирования, должны также учитывать их *интерпретируемость*. В исследовании [33] было показано, что модели с наименьшей перплексией (см. раздел 6.1) обычно хуже интерпретируются обычными людьми. Однако предложенный авторами метод оценивания интерпретируемости моделей предполагает активное участие пользователей и не применим в общем случае.

Авторами [34] были предложены методы автоматического оценивания *связности* найденных тем с помощью внешних баз знаний (*WordNet, Wikipedia, Google*). Наибольшую согласованность с мнениями экспертов показал метод, основанный на расчёте значения *поточечной взаимной информации (pointwise mutual information, PMI)* для пар терминов (w_i, w_j), составляющих тему, на полном корпусе статей Википедии (~2 млн. статей с ~1 млрд. слов):

$$PMI(w_i, w_j) = \log \frac{p(w_i, w_j)}{p(w_i)p(w_j)} \tag{21}$$

7. Программные реализации

Некоторые реализации вероятностных тематических моделей представлены в табл. 1. Кроме того, на веб-странице автора LDA и других тематических моделей Дэвида Блея [23] доступны реализации оригинальных методов, описанных им и его коллегами в статьях.

Фреймворки, позволяющие описать произвольную модель и оценить её оптимальные параметры, описаны в табл. 2.

В табл. 3 представлены библиотеки, упрощающие использование методов тематического моделирования с помощью графического интерфейса.

Табл. 1. Свободные реализации вероятностных тематических моделей.

Название	**Язык**	**Алгоритмы оценивания параметров**	**Ссылка**
LDA-C	C	вариационный EM	www.cs.princeton.edu/~blei/lda-c
Mallet	Java	сэмплирование по Гиббсу	mallet.cs.umass.edu/topics.php
GibbsLDA ++	C/C++	сэмплирование по Гиббсу	gibbslda.sourceforge.net
Gensim	Python	сэмплирование по Гиббсу	radimrehurek.com/gensim
Matlab Topic Modeling Toolbox	Matlab	сэмплирование по Гиббсу	psiexp.ss.uci.edu/research/ programs_data/toolbox.htm
Stanford Topic Modeling Toolbox	Scala	коллапсированное сэмплирование по Гиббсу, коллапсированная вариационная Байесовская аппроксимация	nlp.stanford.edu/software/tmt
GraphLab	C++	коллапсированное сэмплирование по Гиббсу	docs.graphlab.org/topic_modeling.html
Yahoo LDA	C++	сэмплирование по Гиббсу (MapReduce)	github.com/shravanmn/ Yahoo_LDA
Mahout	Java	вариационный EM (MapReduce)	cwiki.apache.org/confluence/display/ MAHOUT/Latent+Dirichlet+Allocation

Табл. 2. Фреймворки для вероятностного моделирования.

Название	**Язык**	**Алгоритмы оценивания параметров**	**Ссылка**
PyMC	Pyhton	методы Монте-Карло для марковских цепей	github.com/pymc-devs/pymc
Factorie	Scala	методы Монте-Карло для марковских цепей, вариационный EM	factorie.cs.umass.edu
Open BUGS	C	сэмплирование по Гиббсу	openbugs.info/w

Табл. 3. Библиотеки для тематического моделирования с графическим интерфейсом.

Название	Ссылка
WinBUGS	www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml
Topic Modeling Tool	code.google.com/p/topic-modeling-tool

8. Заключение

В статье были продемонстрированы эволюция и современное состояние тематического моделирования текстов на естественном языке, которое является перспективным инструментом для обработки больших коллекций документов в приложениях информационного поиска и анализа текстов.

Известно, что бо́льшая часть текстов на сегодняшний день создаётся и публикуется пользователями Интернета в свободной форме. Вместе с тем, именно автоматизированный анализ растущих объёмов подобной информации способен дать заинтересованным организациям уникальную возможность для отслеживания актуальных трендов, а также понимания заинтересованности потребителей в тех или иных продуктах, товарах или услугах.

В связи с этим можно выделить следующие перспективные направления развития тематического моделирования:

- Анализ пользовательского контента (блоги, форумы, социальные сети, рецензии, отзывы и др.);
- Использование дополнительных данных: социального профиля и связей пользователей, метаданных документов и связей между ними, внешних энциклопедий и онтологий, статистики просмотра страниц, записей о реакциях пользователей, временных меток и т.д.;
- Разработка методов обучения и вывода по моделям для распределённой обработки больших потоков данных в реальном времени;
- Разработка способов применения тематического моделирования к новым типам данных (изображения, аудио- и видеофайлы, геном, финансовая статистика и т.д.).

Список литературы

[1] James Allan, Jaime Carbonell, George Doddington, Jonathan Yamron, and Yiming Yang. *Topic Detection and Tracking Pilot Study. Final Report*. Proceedings of the Broadcast News Transcription and Understanding Workshop (Sponsored by DARPA), Feb. 1998

[2] A.K. Jain, M.N. Murty, P.J. Flynn. *Data Clustering: A Review*; ACM Computing Surveys, Vol. 31, No. 3, September 1999

[3] Fabrizio Sebastiani. *Machine Learning in Automated Text Categorization*, ACM Computing Surveys, Vol. 34, No.1, pp.1-47, 2002.

[4] Allan, J. and Lavrenko, V. and Malin, D. and Swan, R. *Detections, bounds, and timelines: UMass and TDT-3.* In Proceedings of Topic Detection and Tracking Workshop, pages 167–174.p. 167-174, Vienna, VA, 2000
[5] Blei, David M. (April 2012). *Introduction to Probabilistic Topic Models*. Comm. ACM 55 (4): 77–84.
[6] Thomas Hofmann. *Probabilistic Latent Semantic Analysis*. UAI 1999: 289-296
[7] Thomas Hofmann. *Probabilistic Latent Semantic Indexing*. SIGIR 1999: 50-57
[8] T.K. Moon. *The expectation-maximization algorithm*. IEEE Signal Processing Mag., vol. 13, pp. 47–60, Nov. 1996
[9] D. Blei, A. Ng, and M. Jordan. *Latent Dirichlet allocation*. Journal of Machine Learning Research, 3:993–1022, January 2003
[10] Gregor Heinrich. *Parameter estimation for text analysis*. Technical report, Fraunhofer IGD, 2005
[11] D. Blei, T. Griffiths, M. Jordan, and J. Tenenbaum. *Hierarchical topic models and the nested Chinese restaurant process*. Neural Information Processing Systems 16, 2003
[12] Yee Whye Teh, Michael I. Jordan, Matthew J. Beal and David M. Blei. *Hierarchical Dirichlet Processes*. Journal of the American Statistical Association, 101:476, 1566-1581, 2006
[13] C. Wang, J. Paisley, and D. Blei. *Online variational inference for the hierarchical Dirichlet process*. Artificial Intelligence and Statistics , 2011
[14] *Mining Text Data* (Springer) Ed. Charu Aggarwal, ChengXiang Zhai, March 2012
[15] D. Blei and J. Lafferty. *Dynamic topic models*. In Proceedings of the 23rd International Conference on Machine Learning, 2006
[16] Xuerui Wang, Andrew McCallum. *Topics over time: a non-Markov continuous-time model of topical trends.* KDD 2006: 424-433
[17] M. Hoffman, D. Blei, and F. Bach. *Online learning for latent Dirichlet allocation*. Neural Information Processing Systems, 2010
[18] Kevin Robert Canini, Lei Shi, Thomas L. Griffiths. *Online Inference of Topics with Latent Dirichlet Allocation*. Journal of Machine Learning Research - Proceedings Track 5: 65-72 (2009)
[19] D. Ramage, D. Hall, R. Nallapati, and C. D. Manning. Labeled LDA. *A supervised topic model for credit attribution in multi-labeled corpora.* In Empirical Methods in Natural Language Processing, pages 248–256, 2009
[20] G. Lisowsky and L. Rost. *Konkordanz zum hebräischen Alten Testament*. Deutsche Bibelgesellschaft, 1958.
[21] Lee, S., Song, J., and Kim, Y. *An Empirical Comparison of Four Text Mining Methods*. Journal of Computer Information Systems, (51:1), 2010, pp. 1-10
[22] D. Blei and J. Lafferty. *Topic Models*. In A. Srivastava and M. Sahami, editors, Text Mining: Classification, Clustering, and Applications. Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, 2009
[23] David M. Blei topic modeling page - http://www.cs.princeton.edu/~blei/topicmodeling.html
[24] D. Mimno and A. McCallum. *Topic models conditioned on arbitrary features with dirichlet-multinomial regression*. In UAI, 2008
[25] Zelong Liu, Maozhen Li, Yang Liu, Mahesh Ponraj. *Performance evaluation of Latent Dirichlet Allocation in text mining*. FSKD 2011: 2695-2698
[26] Steyvers, M. & Griffiths, T. *Probabilistic topic models*. In T. Landauer, D McNamara, S. Dennis, and W. Kintsch (eds), Latent Semantic Analysis: A Road to Meaning. Laurence Erlbaum, 2007

[27] Ali Daud, Juanzi Li, Lizhu Zhou, Faqir Muhammad. *Knowledge discovery through directed probabilistic topic models: a survey*. In Proceedings of Frontiers of Computer Science in China. 2010, 280-301. — перевод на русский К. В. Воронцов, А. В. Темлянцев и др.
[28] Buntine W. L. *Operations for learning with graphical models*. Journal of Artificial Intelligence Research, 1994, 2: 159 – 225
[29] S. Choi, S. Cha, C. C. Tappert. *A Survey of Binary Similarity and Distance Measures*, Journal of Systemics, Cybernetics and Informatics, Vol 8 No 1 2010, pp 43-48
[30] Rui Xu, Donald C. Wunsch II. *Survey of clustering algorithms*. IEEE Transactions on Neural Networks 16(3): 645-678 (2005)
[31] L. Bahl, J. Baker, E. Jelinek, and R. Mercer. *Perplexity — a measure of the difficulty of speech recognition tasks*. In Program, 94th Meeting of the Acoustical Society of America, volume 62, page S63, 1977
[32] H. Wallach, I. Murray, R. Salakhutdinov, and D. Mimno. *Evaluation methods for topic models*. In Proceedings of the 26th International Conference on Machine Learning (ICML 2009), 2009
[33] Jonathan Chang, Jordan L. Boyd-Graber, Sean Gerrish, Chong Wang, David M. Blei. *Reading Tea Leaves: How Humans Interpret Topic Models*. NIPS 2009: 288-296
[34] Newman, Lau, Grieser, Baldwin. *Automatic Evaluation of Topic Coherence*. NAACL HLT 2010
[35] Rosen-Zvi, M., Griffiths, T., Steyvers, M., & Smyth, P. (2004). *The author-topic model for authors and documents*. Proc. of Conf. on Uncertainty in Artificial Intelligence (UAI'04) (pp. 487–494)
[36] Dumais, S. T., Furnas, G. W., Landauer, T. K. and Deerwester, S. (1988). *Using latent semantic analysis to improve information retrieval*. In Proceedings of CHI'88: Conference on Human Factors in Computing, New York: ACM, 281-285
[37] Sanjeev Arora, Rong Ge, Ankur Moitra. *Learning Topic Models - Going beyond SVD*. CoRR abs/1204.1956 (2012)
[38] Daniel D. Lee and H. Sebastian Seung (1999). *Learning the parts of objects by non-negative matrix factorization*. Nature 401 (6755): 788–791

Topic modeling in natural language texts

Anton Korshunov, Andrey Gomzin
{korshunov, gomzin}@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Topic modeling is a method for building a model of a collection of text documents. The model is able to determine topics for each of documents. Shifting from term space to space of extracted topics helps resolving synonymy and polysemy of terms. Besides, it allows for more efficient topic-sensitive search, classification, summarization, and annotation of document collections and news feeds. The paper shows an evolution of topic modeling techniques. The earlier methods are based on clustering. These algorithms use some similarity function defined on two documents. The next generation of topic modeling techniques is based on Latent Semantic Indexing (LSA). Words co-occurrences in documents are analyzed here. Currently, the most popular are approaches based on Bayesian networks — directed probabilistic graphical models which incorporate different kinds of entities and metadata: document authorship, connections between words, topics, documents, and authors, etc. The paper contains a comparative survey of different models along with methods for parameter estimation and accuracy measurement. The following topic models are considered in the paper: Probabilistic Latent Semantic Indexing, Latent Dirichlet Allocation, non-parametric models, dynamic models, and semi-supervised models. The paper describes well-known quality evaluation metrics: perplexity and topic coherence. Freely available implementations are listed as well.

Keywords: topic modeling; topic-sensitive search; document classification; probabilistic graphical models; Bayesian networks; latent Dirichlet allocation; dimensionality reduction; text mining; information retrieval; machine learning.

References

[1]. James Allan, Jaime Carbonell, George Doddington, Jonathan Yamron, and Yiming Yang. Topic Detection and Tracking Pilot Study. Final Report. Proceedings of the Broadcast News Transcription and Understanding Workshop (Sponsored by DARPA), Feb. 1998

[2]. A.K. Jain, M.N. Murty, P.J. Flynn. Data Clustering: A Review; ACM Computing Surveys, Vol. 31, No. 3, September 1999

[3]. Fabrizio Sebastiani. Machine Learning in Automated Text Categorization, ACM Computing Surveys, Vol. 34, No.1, pp.1-47, 2002.

[4]. Allan, J. and Lavrenko, V. and Malin, D. and Swan, R. Detections, bounds, and timelines: UMass and TDT-3. In Proceedings of Topic Detection and Tracking Workshop, pages 167–174.p. 167-174, Vienna, VA, 2000

[5]. Blei, David M. (April 2012). Introduction to Probabilistic Topic Models. Comm. ACM 55 (4): 77–84.

[6]. Thomas Hofmann. Probabilistic Latent Semantic Analysis. UAI 1999: 289-296

[7]. Thomas Hofmann. Probabilistic Latent Semantic Indexing. SIGIR 1999: 50-57

[8]. T.K. Moon. The expectation-maximization algorithm. IEEE Signal Processing Mag., vol. 13, pp. 47–60, Nov. 1996

[9]. D. Blei, A. Ng, and M. Jordan. Latent Dirichlet allocation. Journal of Machine Learning Research, 3:993–1022, January 2003
[10]. Gregor Heinrich. Parameter estimation for text analysis. Technical report, Fraunhofer IGD, 2005
[11]. D. Blei, T. Griffiths, M. Jordan, and J. Tenenbaum. Hierarchical topic models and the nested Chinese restaurant process. Neural Information Processing Systems 16, 2003
[12]. Yee Whye Teh, Michael I. Jordan, Matthew J. Beal and David M. Blei. Hierarchical Dirichlet Processes. Journal of the American Statistical Association, 101:476, 1566-1581, 2006
[13]. C. Wang, J. Paisley, and D. Blei. Online variational inference for the hierarchical Dirichlet process. Artificial Intelligence and Statistics , 2011
[14]. Mining Text Data (Springer) Ed. Charu Aggarwal, ChengXiang Zhai, March 2012
[15]. D. Blei and J. Lafferty. Dynamic topic models. In Proceedings of the 23rd International Conference on Machine Learning, 2006
[16]. Xuerui Wang, Andrew McCallum. Topics over time: a non-Markov continuous-time model of topical trends. KDD 2006: 424-433
[17]. M. Hoffman, D. Blei, and F. Bach. Online learning for latent Dirichlet allocation. Neural Information Processing Systems, 2010
[18]. Kevin Robert Canini, Lei Shi, Thomas L. Griffiths. Online Inference of Topics with Latent Dirichlet Allocation. Journal of Machine Learning Research - Proceedings Track 5: 65-72 (2009)
[19]. D. Ramage, D. Hall, R. Nallapati, and C. D. Manning. Labeled LDA. A supervised topic model for credit attribution in multi-labeled corpora. In Empirical Methods in Natural Language Processing, pages 248–256, 2009
[20]. G. Lisowsky and L. Rost. Konkordanz zum hebräischen Alten Testament. Deutsche Bibelgesellschaft, 1958.
[21]. Lee, S., Song, J., and Kim, Y. An Empirical Comparison of Four Text Mining Methods. Journal of Computer Information Systems, (51:1), 2010, pp. 1-10
[22]. D. Blei and J. Lafferty. Topic Models. In A. Srivastava and M. Sahami, editors, Text Mining: Classification, Clustering, and Applications. Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, 2009
[23]. David M. Blei topic modeling page - http://www.cs.princeton.edu/~blei/topicmodeling.html
[24]. D. Mimno and A. McCallum. Topic models conditioned on arbitrary features with dirichlet-multinomial regression. In UAI, 2008
[25]. Zelong Liu, Maozhen Li, Yang Liu, Mahesh Ponraj. Performance evaluation of Latent Dirichlet Allocation in text mining. FSKD 2011: 2695-2698
[26]. Steyvers, M. & Griffiths, T. Probabilistic topic models. In T. Landauer, D McNamara, S. Dennis, and W. Kintsch (eds), Latent Semantic Analysis: A Road to Meaning. Laurence Erlbaum, 2007
[27]. Ali Daud, Juanzi Li, Lizhu Zhou, Faqir Muhammad. Knowledge discovery through directed probabilistic topic models: a survey. In Proceedings of Frontiers of Computer Science in China. 2010, 280-301.
[28]. Buntine W. L. Operations for learning with graphical models. Journal of Artificial Intelligence Research, 1994, 2: 159 – 225
[29]. S. Choi, S. Cha, C. C. Tappert. A Survey of Binary Similarity and Distance Measures, Journal of Systemics, Cybernetics and Informatics, Vol 8 No 1 2010, pp 43-48
[30]. Rui Xu, Donald C. Wunsch II. Survey of clustering algorithms. IEEE Transactions on Neural Networks 16(3): 645-678 (2005)

[31]. L. Bahl, J. Baker, E. Jelinek, and R. Mercer. Perplexity — a measure of the difficulty of speech recognition tasks. In Program, 94th Meeting of the Acoustical Society of America, volume 62, page S63, 1977
[32]. H. Wallach, I. Murray, R. Salakhutdinov, and D. Mimno. Evaluation methods for topic models. In Proceedings of the 26th International Conference on Machine Learning (ICML 2009), 2009
[33]. Jonathan Chang, Jordan L. Boyd-Graber, Sean Gerrish, Chong Wang, David M. Blei. Reading Tea Leaves: How Humans Interpret Topic Models. NIPS 2009: 288-296
[34]. Newman, Lau, Grieser, Baldwin. Automatic Evaluation of Topic Coherence. NAACL HLT 2010
[35]. Rosen-Zvi, M., Griffiths, T., Steyvers, M., & Smyth, P. (2004). The author-topic model for authors and documents. Proc. of Conf. on Uncertainty in Artificial Intelligence (UAI'04) (pp. 487–494)
[36]. Dumais, S. T., Furnas, G. W., Landauer, T. K. and Deerwester, S. (1988). Using latent semantic analysis to improve information retrieval. In Proceedings of CHI'88: Conference on Human Factors in Computing, New York: ACM, 281-285
[37]. Sanjeev Arora, Rong Ge, Ankur Moitra. Learning Topic Models - Going beyond SVD. CoRR abs/1204.1956 (2012)
[38]. Daniel D. Lee and H. Sebastian Seung (1999). Learning the parts of objects by non-negative matrix factorization. Nature 401 (6755): 788–791

Виртуальная ГИС-лаборатория как инструмент анализа пространственных данных[1]

А.В. Кошкарев, А. А. Медведев,Ю. С. Вишняков,
С. А. Поликарпов, А. Н. Сотников
akoshkarev@yandex.ru , a.a.medvedeff@gmail.com infom@ras.ru ,
polik@vmail.ras.ru , asotnikov@jscc.ru

Аннотация. На основе технологической платформы UniHUB (http://www.unihub.ru), разработанной в Институте системного программирования РАН, в составе Дата-центра РАН создана веб-лаборатория, нацеленная на интеграцию данных дистанционного зондирования в интересах наук о Земле.
Информационная система ориентирована на научное и образовательное сообщество и предназначена для совместной научной работы ее участников в едином рабочем пространстве, обеспечивая поиск источников пространственных данных, формирование хранилищ данных, доступ к данным, в том числе к ресурсам внешних открытых веб-сервисов, к приложениям и обучающим материалам. Облачная распределенная информационная среда UniHUB использована для решения географических задач, включая обработку космических изображений и цифровых моделей рельефа с использованием методов и технологий пространственного анализа и геомоделирования средствами ГИС с открытым исходным кодом Quantum GIS.

Ключевые слова: геоинформационная система; дистанционное зондирование; веб-лаборатория; цифровые модели рельефа; открытые ГИС.

1. Введение

Одна из основных тенденций развития геоинформационных технологий, наметившаяся еще в начале 90-х гг. прошлого века и определяющая пути их эволюции в долгосрочной перспективе — переход к новым формам организации, управления и использования пространственных данных, инфраструктурное обеспечение геоинформационной индустрии, создание национальных и иных инфраструктур пространственных данных (ИПД). С точки зрения перспектив развития информационно-телекоммуникационных

[1] Работа поддержана Российским фондом фундаментальных исследований (проект 11-07-12029-офи-м-2011)

технологий в их приложениях к геоинформатике развитие ИПД можно рассматривать как часть общей тенденции миграции пространственных данных, сервисов и приложений в сетевую среду, в том числе и в первую очередь в среду Интернета. Стала обычной публикация в сети данных и карт, уже давно в ней размещены веб-версии комплексных региональных и национальных атласов (Канада, Украина и др.).

Массовому распространению и использованию сетевых сервисов в повседневной жизни общества в немалой степени способствовало появление публичных картографических веб-сервисов, начало которым положено в 2005 г. проектами компании Google, включая Google Earth («Планета Земля») и Google Maps («Карты Google»), аналогами которых являются «виртуальные глобусы» World Wind (НАСА, США), Bing Maps (Microsoft Corp., США) и др. Особое место среди них занимает сервис OpenStreetMap, некоммерческие картографические ресурсы которого создаются усилиями сообщества пользователей и волонтеров и являются альтернативой аналогичным государственным и коммерческим ресурсам. Другим примером использования геоинформационных технологий широким кругом пользователей в бытовых целях могут служить мобильные и юбиквитные (доступные всюду и всегда) сервисы, основанные на определении местоположения объекта, оснащенного мобильным устройством (смартфоном, ноутбуком, автонавигатором и т.п.). В последние годы наметилась тенденция переноса ресурсоемких приложений, связанных с оперативной обработкой пространственных данных или с их экстремально большими объемами, в среду облачных вычислений. Эта тенденция, впрочем, характерна не только для рассматриваемой предметной области [1]. Технологические решения, обеспечивающие интероперабельность (взаимосовместимость) компонентов ИПД в распределенной сетевой среде, основаны на коммерческом программном обеспечении геоинформационных систем (ГИС) и на свободно распространяемых программных продуктах, в том числе с открытым исходным программным кодом. В ряде стран использование последних становится элементом государственной научно-технической политики. Растет доступность пространственных данных, значительная часть которых, включая базовые пространственные данные ИПД, архивы данных дистанционного зондирования Земли (ДЗЗ) из космоса, становится открытой, то есть распространяемой безвозмездно. Это создает необходимые предпосылки для их широкого использования, в том числе научно-образовательным сообществом, в интересах решения научных и прикладных задач.

2. Задачи

Научное исследование территории предполагает анализ и оценку существующих источников информации, поиск необходимых исходных данных, включая данные дистанционного зондирования, которые в последние годы стали базовой информацией географических исследований. Их наличие и

доступность, однако, не решает проблемы их эффективного использования, порождая задачи их обработки и интерпретации. Набор дополнительных источников информации будет определяться тематикой исследования и поставленными целями и конечными задачами. Важен и уровень профессиональных знаний и умений каждого отдельного исследователя, его опыт работы и многие другие обстоятельства. Тем не менее, для исследований в области создания распределенной среды обработки данных ДЗЗ можно выделить ряд общих информационных потребностей исследователей и научных коллективов [2].

На первом этапе исследования это потребности в космических и аэроснимках района исследования; научных методиках и теории использования ДЗЗ; современных программно-технологических сервисах. Все они могут быть удовлетворены путем создания распределенной среды.

На втором этапе (непосредственного проведения исследования) часто возникает необходимость в получении дополнительных количественных данных, изучении существующих методик решения подобных исследовательских задач и обработки полученных результатов, т.е. потребности в первичных данных (например, данных измерений состояния различных природных объектов и их характеристик); аналитических и экспериментальных методиках и методах обработки этих данных.

Они удовлетворяются за счет обращения в специализированные базы и банки данных и к возможностям современных ГИС и к другим программным средствам реализации различных аналитических (расчетных) функций.

На третьем, заключительном этапе возникает информационная потребность в отыскании наилучших способов представления полученных результатов, которая может быть удовлетворена за счет обращения к полнофункциональным ГИС для моделирования и представления полученных результатов.

Таким образом, удовлетворение информационных потребностей исследователей для решения задач в области дистанционного зондирования может и должно осуществляться за счет разработки специализированной системы, реализующей все эти функции путем перераспределения поисковых заданий исследователей к соответствующим серверам.

Одна из главных предметных областей применения создаваемой системы – изменения природной среды и ее компонентов, оценка природных ресурсов и состояния окружающей среды. Ее средства могут обеспечить:

- контроль рационального использования природных ресурсов и охрану природы;

- комплексный мониторинг природной среды;

- экологическое обоснование проектов;

- оценку инвестиционной привлекательности территорий;

- информационное обеспечениеей прединвестиционных, предпроектных и проектных работ;

- разработку и государственную экспертизу региональных схем социально-экономического развития, территориального планирования, капитального строительства, проектов освоения природных ресурсов;

- подготовку и издание информационно-справочных и аналитических материалов;

- подготовку общественных слушаний.

3. ГИС-лаборатория на платформе UniHUB

Для решения поставленных задач распределенной обработки данных ДЗЗ на платформе UniHUB (http://www.unihub.ru) была создана ГИС-лаборатория. В рамках принятой терминологии ГИС-лаборатория является группой пользователей UniHUB со всей доступной группам функциональностью для организации совместной работы [3]. Совместная работа виртуальной лаборатории и ее участников организуется с помощью единого рабочего пространства и взаимодействия информационных процессов на основе портального решения и веб-служб. Она является универсальной «точкой входа» в рабочую область авторизованных пользователей с возможностью доступа к файлам и программам, формирования файловых хранилищ данных и документов, разделения прав доступа к данным, документам и программным приложениям, управления работами (рассылка заданий и контроль исполнения, ведения архива и контроля версий документов).

Пользователь лаборатории имеет возможность найти необходимую информацию в файловых хранилищах пространственных данных — каталоге данных ДЗЗ, цифровых моделей рельефа, ГИС-проектах и др., а также сохранить информацию в пользовательской рабочей области, осуществив подбор тематических данных и других картографических материалов различного масштаба.

Используемый вариант организации информационной среды — клиент-серверная архитектура: пространственные данные хранятся на сервере, и многие пользователи имеют возможность обращаться к одному массиву данных. Упрощается администрирование данных, появляется возможность параллельного совместного редактирования и ведения нескольких версий. Вместе с тем, в соответствии с архитектурой UniHUB, ГИС и ДЗЗ-инструментарий выделены и доступны как самостоятельные клиентские приложения вне ГИС-лаборатории (рис. 1).

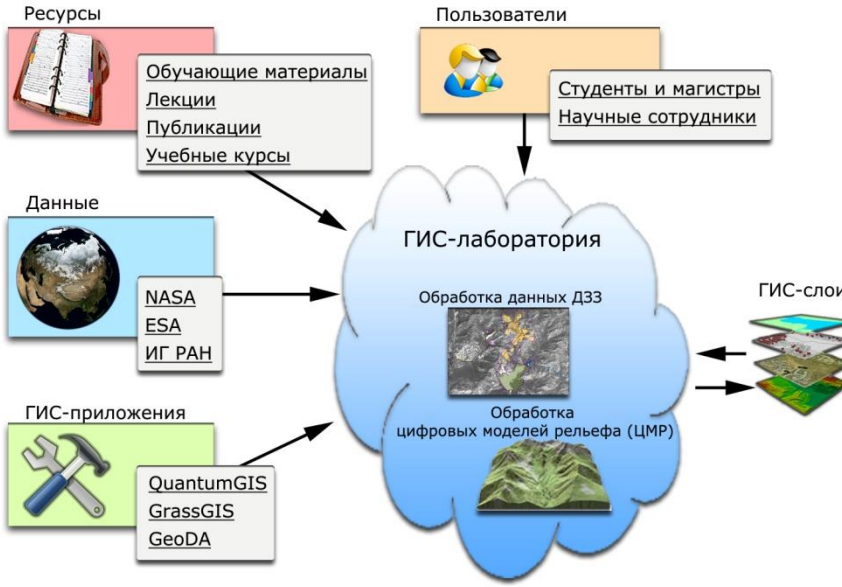


Рис. 1. Схема организации ГИС-лаборатории

Лаборатория базируется не только на программных приложениях, но и обучающих материалах и исходных данных, что позволяет использовать ресурсы лаборатории в образовательных целях.

Преимуществами такого решения являются:

- универсальность — приложение может быть легко настроено в соответствии с задачей и набором необходимых пользователю функций;

- простота использования — пользователи самостоятельно способны освоить приложение в короткий срок.

Все данные, необходимые пользователю, извлекаются не только из файловых хранилищ лаборатории, но и из открытых распределенных веб-сервисов, распространяющих данные ДЗЗ. Соответствующие данные направляются на обработку в соответствии с выбранными процедурами, на анализ или для поиска новых дополнительных данных.

4. Организация пользовательского доступа к пространственным данным

Принципиальной проблемой, определяющей успех работы лаборатории, в целом является вопрос обеспечения полноты покрытия данными, которые

доступны пользователям, обращающимся к системе с запросами на выполнение некоторых конкретных действий.

С точки зрения самих данных, в хранилище лаборатории и на распределенных серверах, подключенных к ней, находятся принципиально два разных типа данных, но полученных на основе дистанционного зондирования Земли. Первый тип — цифровые модели рельефа местности разного пространственного разрешения и охвата. Ко второму типу относятся космические снимки, а также каталог аэрофотоснимков.

Для работы в ГИС-лаборатории доступны глобальные и семиглобальные цифровые модели рельефа:

- глобальная ЦМР ASTGTM (ASTER GDEM), созданная в результате стереофотограмметрической обработки снимков космического аппарата ASTER (США) (шаг сетки ок. 30 м);
- семиглобальная ЦМР SRTM, полученная в результате съемки земной поверхности многоразовым транспортным космическим кораблем NASA «Индевор» (Endeavour) в феврале 2000 г. (3 угловые секунды или ок. 100 м в низких широтах, в пределах области от 60° с.ш. до 60° ю.ш.);
- глобальная ЦМР GTOPO30, созданная Геологической съемкой США USGS (30 угловых секунд или около 1 км в низких широтах), а также подобные ей GTOPO5, GTOPO2 и GTOPO1;
- ETOPO5 (5 угловых минут или ок. 10 км).

В целях оперативности доступа в хранилище ГИС-лаборатории доступны данные космической и аэрофотосъемки:

- Космические снимки Landsat глобального охвата с разрешением 150 м.;
- Синтезированные космические снимки Landsat за 1990 г. глобального охвата с разрешением 30 м.;
- Синтезированные космические снимки Landsat за 2000 г. глобального охвата с разрешением 30 м.;
- Каталог аэрофотоснимков (метаданные) ИГ РАН;

Для удобства пользовательского доступа к хранимым данным дистанционного зондирования был реализован механизм, позволяющий осуществить поиск данных посредством задания местоположения на географической карте. Программная реализация пользовательского интерфейса выполнена на базе Flash-приложения (рис. 2).

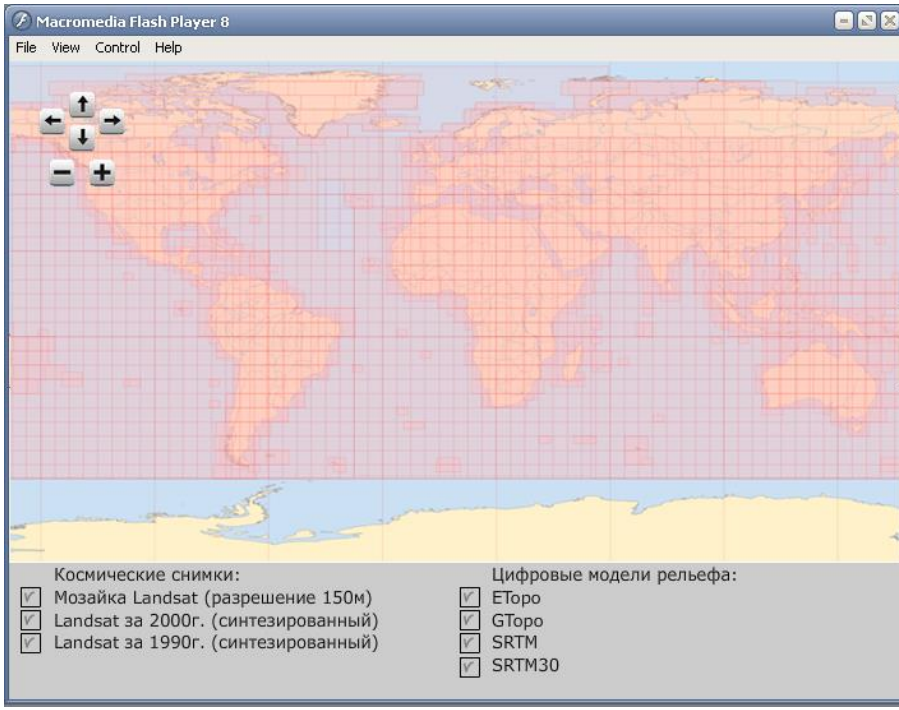


Рис. 2. Программное приложение для загрузки данных

Пользователь, воспользовавшись интерактивной картой, может легко выбрать соответствующие данные, а также необходимый ему пространственный охват загружаемых ресурсов.

С технологической точки зрения, загружаемые данные попадают из единого хранилища в рабочую область пользователя, в так называемую «пользовательскую сессию». После этой процедуры необходимые наборы данных доступны для последующей обработки.

5. Инструменты ГИС-лаборатории по работе с пространственными данными

Одним из основных программных приложений, установленных в лаборатории для обработки пространственных данных, является Quantum GIS (http://www.qgis.org), которое является программным обеспечением с открытым исходным кодом, созданное в мае 2002 года, а с июня того же года ставшее проектом на площадке SourceForge. В настоящее время Quantum GIS (QGIS) работает на большинстве платформ: Unix, Windows, и OS X. QGIS легок в использовании, имеет приятный и простой графический интерфейс,

поддерживает множество растровых и векторных форматов данных, выпускается на условиях лицензии GNU General Public License (GPL). Это означает, что пользователь всегда будет иметь доступ к программному обеспечению ГИС, которое является бесплатным и может свободно адаптироваться [4].

QGIS позволяет использовать множество распространенных функций ГИС, обеспечиваемых встроенными инструментами и модулями; среди них:

- просмотр данных,
- оформление и компоновка карт,
- управление данными: создание, редактирование и экспорт,
- анализ данных,
- публикация карт в сети Интернет,
- расширение функциональности Quantum GIS с помощью модулей расширения.

QGIS во многом основана на GRASS GIS, но имеет более понятный для неискушенного пользователя интерфейс. GRASS GIS был также установлен в рабочем пространстве ГИС-лаборатории в целях более глубокой обработки пространственных данных. GRASS GIS включает инструменты для пространственного моделирования, визуализации растровых и векторных данных, управления геоданными и их анализа, обработки спутниковых снимков и аэрофотоснимков.

GRASS GIS включает более 400 встроенных модулей для анализа и около 100 дополнительных модулей, поддерживаемых сообществом, а также справочные материалы, которые можно свободно загрузить с основного веб-сайта GRASS http://grass.osgeo.org, или с многочисленных «зеркал» по всему миру. GRASS GIS в настоящее время используется по всему миру в академических и коммерческих кругах, а также многих правительственных учреждениях и экологических консалтинговых компаниях. Она работает на различных популярных аппаратных платформах и является свободным программным обеспечением с открытым исходным кодом, распространяемым на условиях GPL.

Для расширения функциональных возможностей ГИС-лаборатории по работе с пространственными данными был также установлен кросс-платформенный программный пакет с открытым исходным кодом OpenGeoDa. Это бесплатный программный пакет, который дает возможность производить пространственный анализ данных, визуализацию, пространственную автокорреляцию и пространственное моделирование. Пакет специализируется на анализе пространственных данных и их визуализации, основываясь на методах динамического связывания и пространственных выборках.

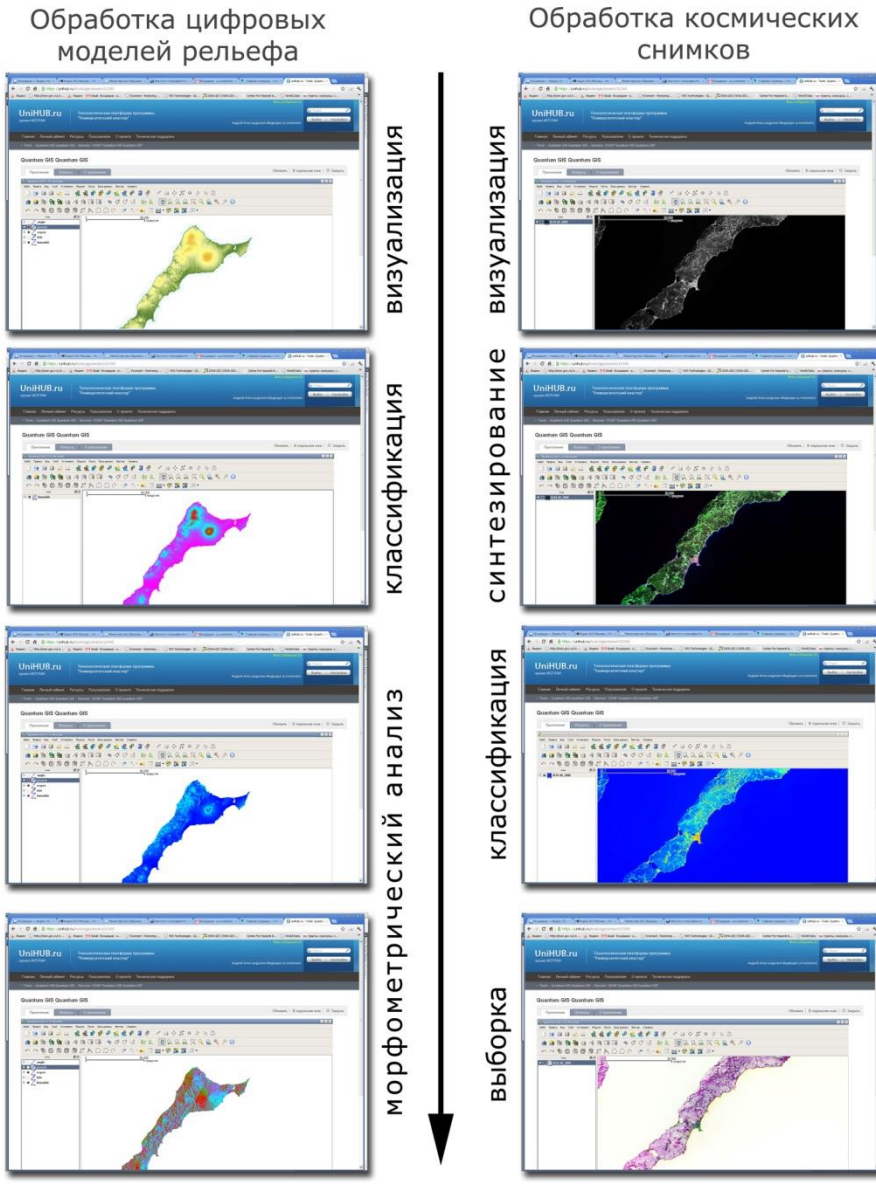


Рис. 3. Процессы и результаты обработки данных

Результаты работы с пространственными данными OpenGeoDa способна представлять не только в виде карт, но и в виде гистограмм, графиков

распределения, точечных диаграмм. OpenGeoDa имеет преимущества по функциональным возможностям в сравнении с аналогичными программными продуктами в инструментах пространственного анализа, многомерного анализа данных и др.

В результате работы на кроссплатформенных геоинформационных системах участниками ГИС-лаборатории созданы карты экспозиций склонов, общей кривизны, индекса пересеченности, углов наклона, визуализированы цифровые модели рельефа. В лаборатории были также обработаны космические снимки с различных платформ, в результате чего были получены данные о современном использовании земель (рис. 3).

6. Заключение

За время работы ГИС-лаборатории в ее использовании приняли участие и прошли обучение студенты и магистры Географического факультета МГУ им. М. В. Ломоносова. Занятия по работе в лаборатории внедрены в учебный процесс и присутствуют в учебном плане студентов 3-его курса. В работе лаборатории также приняли участие студенты Московского государственного университета геодезии и картографии (МИИГАиК) в рамках прохождения производственной практики в Институте географии РАН. В ходе прохождения практики и обучения на базе лаборатории студентами получены навыки обработки цифровых моделей рельефа, а в частности получения морфометрических характеристик рельефа местности.

В процессе работы ГИС-лаборатории получен ряд новых результатов в формировании стыкующихся информационных потоков данных дистанционного зондирования от разных источников, «конструировании» новых знаний на основе хранилищ массивов и организации информационного интерфейса для ученых и студентов по конкретным направлениям прикладных исследований в области наук о Земле.

В рамках лаборатории реализована концепция построения распределенной информационной системы. Пользователи ГИС-лаборатории обеспечены различными наборами данных, в том числе из внешних узлов, различными инструментами их обработки, и, главное, экспертной информационной поддержкой.

Список литературы

[1] Гайсарян С. С., Самоваров О. И., Аветисян А. И., Иванников В. П. «Университетский кластер»: интеграция образования, науки и индустрии // Открытые системы. — № 5. — 2010. — http://www.osp.ru/os/2010/05/13003059/ .

[2] Поликарпов С. А., Вишняков Ю. С., Жижченко А. Б., Сотников А. Н., Кошкарев А. В., Медведев А.А. Распределенная система для интеграции данных дистанционного зондирования в интересах наук о Земле // Тезисы Совещания APN (MAIRS/NEESPI/SIRS) «Экстремальные проявления глобального изменения климата на территории Северной Азии». — SCERT, 2012. — С. 49-51.

[3] Самоваров О. И. «Концепция Web-лабораторий (ИСП РАН)», Семинар «Облачные сервисы», 2011. — http://unicluster.ru/23-report-seminar-cloud-services.html .
[4] Кошкарев А. В., Медведев А. А., Серебряков В. А. Данные и сервисы Академической ИПД. — ИнтерКарто-ИнтерГИС-18: Устойчивое развитие территорий: теория ГИС и практический опыт. Материалы международной конференции / Редкол.: С. П. Евдокимов (отв. ред.) [и др.]. Смоленск, 26-28 июня, 2012 г. — Смоленск, 2012. . — С. 153-156.

Virtual GIS laboratory as a tool for spatial data analysis

Koshkarev A. V., Medvedev A. A. - Russian Acad Sci, Inst Geog, Moscow, Russia
Vishnyakov Yu. S., Polikarpov S. A. - Russian Acad Sci, Dorodnicyn Comp Ctr, Moscow, Russia
Sotnikov A. N. - Russian Acad Sci, Joint Supercomp Ctr, Moscow, Russia

Abstract: Based on UniHUB technology by ISP RAS, Web-laboratory (hub) for remote sensing data integration is currently being deployed. Huge amount of remote sensing data is currently available. Data are being gathered by separate researchers and scientific projects as well as by complex global databases of major international programs and information centers. Thus we observe broad migration of geographic data, services and applications to network environment, particularly to Internet. But our information system focuses not only on data collecting but mainly on creating problem-oriented community of scientists (i.e. experts, researchers, students) online. It provides a search mechanism for sources of spatial data via graphical interface, data access, web applications and training materials. In context of UniHUB our Web-laboratory exists as a user group. Anyone can easily contribute new materials to it as well as take advantage of assistance from competent colleagues. Cloud environment of UniHUB is used to solve geographic problems, such like handling of satellite images and digital elevation models by methods and techniques of spatial analysis and geomodeling in open source Quantum GIS. Currently our Web-laboratory is widely used among students. But we expect it to have much more applications such like assessments in the field of natural resources, state of environment etc.

References:

[1]. Gajsaryan S. S., Samovarov O. I., Avetisyan A. I., Ivannikov V. P. «Universitetskij klaster»: integratsiya obrazovaniya, nauki i industrii ["University Cluster": education, science and industry integration]. Open Systems, 2010, no. 5, http://www.osp.ru/os/2010/05/13003059/ (in Russian).

[2]. Polikarpov S. A., Vishnyakov Yu. S., ZHizhchenko A. B., Sotnikov A. N.,Koshkarev A. V., Medvedev A. A. Distributed system for remote sensing data integration in the interest of Earth sciences. Proceedings of MAIRS/NEESPI/SIRS APN Workshop "Climate Change Induced Extremes in Nothern Asia", 2012, p. 49.

[3]. Samovarov O. I. Kontseptsiya Web-laboratorij (ISP RAN) [Web-laboratory idea (ISP RAN)]. "Cloud Services" Workshop, 2011, http://unicluster.ru/23-report-seminar-cloud-services.html (in Russian).

[4]. Koshkarev A. V., Medvedev A. A., Serebryakov V. A. Dannye i servisy Akademicheskoj IPD [Data and services for academic spatial data infrastructure]. Proceedings of International Conference "InterCarto-InterGIS 18. Sustainable development: GIS theory and practice", 2012, pp. 153-156 (in Russian).

Use of Multiple Features for Extracting Topics from News Clusters

A.A. Alekseev, N.V. Loukachevitch
a.a.alekseevv@gmail.com, louk_nat@mail.ru
Lomonosov Moscow State University

Annotation. In this paper we consider a method for extraction of alternative names of a concept or a named entity mentioned in a news cluster. The method is based on the structural organization of news clusters and exploits comparison of various contexts of words. The word contexts are used as basis for multiword expression extraction and main entity detection. At the end of cluster processing we obtain groups of near-synonyms, in which the main synonym of a group is determined.

Keywords: near-synonym detection; text-structure models; multi-document summarization

1. Introduction

As it is widely known, a natural language text contains a lot of sense-related words and expressions such as synonyms, antonyms, hyponyms, hypernyms and others. The presence of such expressions in texts contradicts to the standard representation of texts as a bag of words.

The modeling of semantic relatedness between words in a text can be based on linguistic or statistical approaches. A linguistic approach considers this phenomenon as a property of natural language texts called lexical cohesion and represents it as lexical chains of semantically-related words [1]. This representation is based on existing linguistic resources such as WordNet [2] or user-generated resources as Wikipedia [3]. The evident problem of such an approach is the possible absence of necessary knowledge in a utilized resource or the irrelevance of described information to the text contents.

Well-known statistical approaches such as Latent Dirichlet Allocation are often called “topic models” (LDA) [4, 5]. Topic Models [6] are based on the idea that documents are mixtures of topics where a topic is a probability distribution over words. In such models two probability distributions are usually considered:

- Topics-VS-Documents distribution,
- Words-VS-Topics distribution.

Extraction of such topics is based on the iterative application of statistical methods (for example, Gibbs Sampling) and the co-occurrence of words in the same

documents of a collection. The statistical inference of topics does not consider the existing lexical relations between words or the internal structure of texts.

In this paper we will consider a deterministic approach to extraction of thematically-related chains of words and expressions (thematic nodes) based on various factors including:

- thesaurus information,
- spelling resemblance,
- several types of context similarity,
- discourse-based similarity. To extract such a similarity we utilize some assumptions on the structure of natural language texts.

We will demonstrate our approach on the news cluster summarization task. News clusters can contain semantically-related words as within a single cluster document as in different documents of the cluster, which can lead to problems in news clusterization and further summarization of the cluster. For example, *the U.S. air base in Kyrgyzstan* may be called in documents of the same news cluster as *Manas base, Manas airbase, Manas, base at Manas International Airport, U.S. base, U.S. air base* and etc.

This paper is organized as follows: after related works are surveyed in section 2, we discuss in section 3 a theoretical basis of the proposed algorithm, in particular coherent text-structure model. Detailed algorithm description is provided in section 4. In section 5, we describe the evaluation procedure and present the results. Section 6 concludes the paper.

2. Related Approaches

A context-based analysis is the most popular and widely used way to detect semantically related expressions [7]. Intuitively, words that can be used in the same context have a high chance to have similar meanings. Based on this, many methods have been proposed focusing on two aspects – what type of context to consider and what similarity measure to employ [8].

Dang et al. [9] proposes two simple methods addressing the quality of contexts for near-synonym extraction. Different types of contexts have different synonymy contribution. For example, consider two context words *"carries"* and *"points"* in the following sentences:

(1) *"He carries a gun in the bag"*
(2) *"He carries a pistol in the bag"*
(3) *"He points his gun at us"*
(4) *"He points his pistol at us".*

Obviously, *"points"* is a better context than *"carries"* for determining that *"gun"* and *"pistol"* should be near-synonyms. Therefore, not only a context word co-occurrence frequency with a term is important, but also how many different terms

co-occur with this context word. This characteristic reflects the quality of a context word. The paper offers two formulas for the context quality estimation.

There are some more sophisticated approaches for context extraction and exploiting. An unsupervised learning algorithm for identification of paraphrases from a corpus of multiple English translations of the same source text is proposed in [10]. Part-of-speech templates of neighbouring words are considered as contexts in this work. An iterative algorithm starts from the same words and phrases extraction in multiple translations. These are "good" examples of paraphrasing. Afterwards, context templates are extracted for found coincidences. These are "good" context templates. All other context templates are considered as "bad" examples. A step of the algorithm lies in the generation of new "good" examples of paraphrasing on the basis of extracted "good" templates and so on. A set of paraphrases is produced as a result of the described procedure.

The problem of alternative names for named entities is partly solved by co-reference resolution techniques (*Russian President Dmitry Medvedev, President Medvedev, Dmitry Medvedev*) [7, 11]. In Entity Detection and Tracking Evaluations, mainly such entities as organizations, persons and locations are detected and provided with co-referential relations [12]. But main entities of a cluster can be events such as *air base closure* and *air base withdrawal*. Besides, the variability of entity names in news clusters refers not only to concrete entities, but also to concepts such as *ecology* or *economic problems*.

News clusters as sources of various paraphrases are studied in several works. In [13] the authors describe the procedure of corpus construction for paraphrase extraction in the terrorist domain. The study in [14] is devoted to creation of a corpus of similar sentences from news clusters as a source for further paraphrase analysis. These studies are aimed to obtain general knowledge about a domain or linguistic means of paraphrasing, but it is also important to extract similar expressions of various types from a news cluster and to use them to improve the processing of the same news cluster.

3. Text-Structure Model

3.1. Thematic Structure and Thematic Nodes

The processing of cluster texts is based on the structure of coherent texts, which have such properties as the topical structure and cohesion.

Van Dijk [15] describes the topical structure of a text, the macrostructure, as a hierarchical structure in a sense that the theme of a whole text can be identified and summed up to a single proposition. The theme of the whole text can usually be described in terms of less general themes, which in turn can be characterized in terms of even more specific themes. Every sentence of a text corresponds to a subtheme of the text.

The macrostructure of a natural language text defines its global coherence: “Without such a global coherence, there would be no overall control upon the local connections and continuations. Sentences must be connected appropriately according to the given local coherence criteria, but the sequence would go simply astray without some constraint on what it should be about globally” [15].

Thus, a natural language text should have the main theme. In the hierarchical thematic structure of the document the main theme should be elaborated, specified with subthemes corresponding to specific sentences. Because of the global connectivity of the thematic structure, a considerable number of subtheme participants should be related to main participants of the main theme (fig. 1). So we suppose that numerous lexical relations in a text should refer to the participants of the main theme. We call such a node of links to more important thematic element – *thematic node*.

Interactions between subtheme participants described in specific sentences should be also related to the main theme of the document. From here we conclude that if two entities C_1 and C_2 often co-occur in the same sentences of a text, it means that the text is devoted to the consideration of relations between these entities and they represent different elements of the text theme [16, 17]. At the same time, if two lexical expressions C_1 and C_2 are rarely met in the same sentences, but co-occur very frequently in neighbour sentences then we can suppose that they are elements of a lexical chain, and there exists a semantic relation between them.

So we think that an important step to reveal the thematic structure of a document is to reconstruct thematic nodes. In comparison with LDA topics, thematic nodes do not comprise words co-occurring in the same documents or the same sentences - each thematic node is supposed to collect words and expressions corresponding to a separate participant of the situation described in the text.

If to compare with standard lexical chaining techniques [2], which try to construct chains of semantically related expressions in texts, thematic node elements are supposed to be related to its main element (center of the thematic node), and if two related expressions (for example, *doctor* and *patient*) co-occur in the same sentences of the text, it means that their relations represent the focus of the text contents, these expressions are related to different participants of the text topics and should be assigned to different thematic nodes. And on the contrary, if two expressions rarely co-occur in the same sentences, but frequently co-occur in neighbouring sentences, then they have to be considered as the elements of the same thematic node.

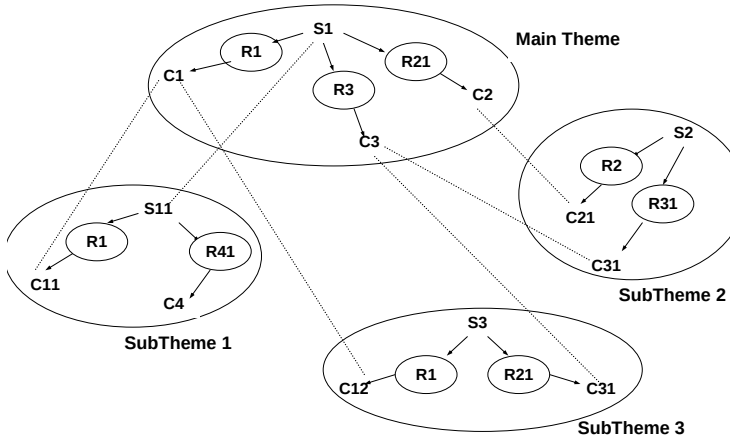


Fig.1. *The hierarchy of themes in a natural language text and links between levels of the hierarchical structure where S1, S11, S2, S3 are predicates describing a situation, C1… C4 are entities participating in the described situations, Ri are roles of entities.*

A news cluster is not a coherent text but cluster documents are devoted to the same theme. Therefore, statistical features of the topical structure are considerably enhanced in a thematic cluster, and on such a basis we try to extract unknown information from a cluster.

3.2. Hypothesis Validation

To check our idea that semantically related expressions are more often met in neighbouring sentences than in the same sentences we have carried out the following experiment. More than 20 large news clusters have been matched with terms of Socio-political thesaurus [17] and thesaurus-based near-synonyms have been detected. Such types of near-synonyms include (these examples are translations from Russian; in Russian the ambiguity of expressions is absent):

- nouns – thesaurus synonyms (*Kyrgyzstan – Kirghizia*),
- adjective – noun derivates (*Kyrgyzstan – Kyrgyz*),
- hypernym and hyponym nouns (*deputy – representative*),
- hypernym–hyponym noun - adjective (*national – Russia*),
- part-whole relations between nouns (*parliament – parliamentarian*),
- part-whole relations for adjective and noun (*American – Washington*).

For each cluster we considered all these pairs of expressions with a frequency filter: the frequencies of the expressions in a cluster should be more than a quarter of the

number of documents in the corresponding cluster. For these pairs we computed the ratio between their co-occurrence in the same sentence clauses F_{segm} and in neighbouring sentences F_{sent}. Table 1 shows the results of our experiment.

Type of relation	F_{segm}/F_{sent} ratio	Number of pairs
Synonymic Nouns	0.309	31
Noun-adjective derivation	0.491	53
Hyponym – Hypernym (nouns)	1.130	88
Hyponym – Hypernym (noun – adjective)	1.471	28
Meronym- holonym (nouns)	0.779	58
Meronym- holonym (noun – adjectives)	1.580	29
Other	1.440	21483

Table 1: Frequency ratio of related expressions within segments of sentences and neighbouring sentences

From the Table 1 we can see that the most closely-related expressions (synonyms, derivates) are much more frequent in neighbouring sentences than in the same clauses of sentences. Further, the more the distance in a sense between expressions is the more the ratio Fs_{egm}/F_{sent} is until the stabilization near the value equal to 1.5.

We can also see that noun-noun and noun-adjective pairs have different values of the ratio. We suppose that in many cases adjectives are components of noun groups, which can play own roles in a news cluster. Therefore, the first step in detection of thematic nodes should be extraction of multiword expressions.

4. Thematic Nodes Construction

Thus our aim in cluster processing is to reveal the main participants of the situation described in a cluster by means of thematic nodes extraction. We believe that such information about a cluster should improve performance of such operations as cluster refining and cluster summarization. The construction of thematic nodes is based on different types of similarities between expressions. Besides, the necessary condition of inclusion of two expressions in the same thematic nodes is their high co-occurrence frequency in neighbouring sentences (see section 3) in comparison with the same sentence co-occurrence frequency.

The cluster processing consists of three main stages. At the first stage noun and adjective contexts are accumulated. The second stage is devoted to multiword expression recognition. At the third stage thematic nodes are constructed.

In next sections we consider processing stages in more detail. As an example we use a news cluster devoted to Kyrgyzstan and the United States agreement denunciation on U.S. air base located at the Manas International Airport (19.02.2009). This news

cluster contains 195 news documents and is assembled on the basis of the clusterization algorithm described in [18].

4.1. Word Contexts Extraction

Sentences are divided into segments between punctuation marks. Contexts of a word W including nouns and adjectives situated in the same sentence segments as W are considered. The following types of contexts are extracted:

- Neighbouring words: neighbouring adjectives or nouns situated directly to the right or left from W (Near);
- Across-verb words: adjectives and nouns occurring in sentence segments with a verb, and the verb is located between W and these adjectives or nouns (AcrossVerb);
- Not-near words: adjectives and nouns that are not separated with a verb from W and are not direct neighbours to W (NotNear).

In addition, adjective and noun words that co-occur in neighbouring sentences are memorized (NS). For NS context extraction, only sentence fragments from the beginning up to a segment with a verb in a personal form are taken into consideration. It allows us to extract the most significant words from neighbouring sentences.Each context type has a numeric value equal to its frequency for each candidate pair. For example, if a candidate pair of objects A and B occurred 3 times directly near in an analysed news cluster, it means, that this candidate pair would have Near value equal to 3.

Along with the described context types, we exploited classical n-gramm contexts. We call such contexts – strict contexts: two words to the left and two words to the right in the fixed order around the word W. For example, if we extract strict contexts of word *"processing"*, then in the sentence *"Cluster processing consists of three main stages"* we will yield the string context: (*, *cluster, W, consist, of*), where * means a context element missing in the beginnings and endings of sentences. Thereon strict contexts for all the words are gathered and two candidate words can be compared by the number of identical strict contexts.

4.2. Extraction of Multiword Expressions

We consider recognition of multiword expressions as a necessary step before thematic nodes construction. An important basis for multiword expression recognition is the frequency of word sequences [19]. However, a news cluster is a structure where various word sequences are repeated a lot of times. We supposed that the main criterion for multiword expression extraction from clusters is the significant excess in a co-occurrence frequency of neighbour words in comparison with their separate occurrence frequency in segments of sentences (1):

$$Near > 2 * (AcrossVerb + NotNear) \quad (1)$$

In addition, the restrictions on frequencies of potential component words are imposed.

The search for candidate pairs is performed in order of the *"Near – 2*(AcrossVerb + NotNear)"* value decrease. If a suitable pair has been found, its component words are joined together into a single phrase and all contextual relationships are recalculated. The procedure starts again and repeats until at least one join is performed.

As a result, such expressions as Parliament of Kyrgyzstan, the U.S. military, denunciation of agreement with the U.S., Kyrgyz President Kurmanbek Bakiyev were extracted from the example cluster.

Two measures of quality were tested for multiword expression extraction in our previous work [20]. Firstly, the share of syntactically correct groups among all extracted expressions was evaluated. Secondly, a professional linguist was invited to select the most significant multiword expressions (5-10) for each cluster, and arrange them in the descending order of importance. The proposed algorithm for multiword expression extraction showed 91.4% precision and 72.6% recall result, which is enough for further thematic node construction.

4.3. Similarity Features

A set of the five main similarity features is used for determining of semantically related expressions and the following thematic nodes construction. Some of these features are based on context information, extracted directly from the news cluster under consideration. Others reflect formal resemblance information and information from pre-defined resources. Each similarity feature contributes some points to the overall score of a candidate pair. The scoring algorithm would be described in the next section in more details.

Context-dependent features:

The neighbouring sentence feature (**NSF**). This feature is based on the discourse model described in section 3 and reflects the expected co-occurrence of thematic node elements in the same and neighbouring sentences.

NSF feature is calculated on the basis of AcrossVerb, Near, NotNear and NS context features and their average distribution in the cluster. NSF feature estimates the excess of neighbouring sentence counts in comparison to across-verb, near and not-near contexts and the following value is the basis of this feature:

$$C = NS - 2 * (AcrossVerb + Near + NotNear) \quad (2)$$

The general formula for NSF feature score contribution has the next form:

$$\text{NSF} = Min\left[1, \frac{C}{Avg(C)}\right] \quad (3)$$

where AVG(C) is an average value of C among positive values in the whole cluster.

NSF feature is also our regulatory feature. It means that a candidate pair could not be included in the same thematic node if NSF feature has a negative value (see section 3.2). A candidate pair with negative NSF feature value has no overall score and is not considered by the algorithm. It is worth noting that such a feature has not been utilized before for such tasks as near-synonym detection, topic extraction or lexical chaining before.

***Strict context feature* (SC).** This feature is based on the comparison of fixed order contexts. The more identical templates a pair shares the more its similarity is. Contexts with missing information (or not full 4-gramm contexts) have a less weight than full contexts.

Strict contexts are scored using the following weighting formula: each word in a context n-gramm has the weight 0.25. For instance, the n-gramm (*, *, *consist, of*) would have the weight 0.5 and (*news, cluster, consist, of*) would have the weight equals 1.0, which is the maximum weight for a full context n-gramm.

Pairs \ Features	Context-independent		Context-dependent				SPlus	SCORE
	BS	**TS**	**NSF**	**SC1**	**SC2**	**SPS**		
"Kyrgyzstan" – "Kyrgyz"	1.00	1.00	0.07	+	1.00	1.00	+	**4.07**
"Airbase" – "Manas Airbase"	1.00	0.00	1.00	+	1.00	1.00	+	**4.00**
"Kyrgyz parliament" – "Parliament of Kyrgyzstan"	1.00	0.00	0.79	+	1.00	1.00	+	**3.79**
"Manas Airbase" – "Manas base"	1.00	0.00	0.71	+	1.00	0.94	+	**3.65**
"Airbase" – "Base"	0.00	0.00	1.00	+	1.00	1.00	+	**3.50**

Table 2: Example of candidate pairs ranking (top 5 pairs at the first iteration).

SC feature has a Boolean value: 0 or 1. The maximum value of 1 is assigned if SC feature has the value not less than 2. It means, that for satisfying of SC feature a candidate pair has to share not less than 2 identical context templates (with context template weights taken into account).

Cosine similarity feature (**S**calar **P**roduct **S**imilarity, **SPS**). Each type of context information, described in 4.1, represents a vector of frequencies assigned to each word. Dimensions in this vector reflect co-occurrence frequencies of a word under consideration with all other words, mentioned in the news cluster. When the context

vectors are calculated, they can be compared with classic cosine similarity metric reflecting the similarity of two expressions. SPS feature can be considered as more smoothed and flexible than SC feature, because they both were designed to analyse the sentence context.

SPS feature score has a decimal value and directly related to cosine similarity metrics of the corresponding vectors. It is calculated as the sum of cosine similarity values for all context vectors (AcrossVerb, Near and NotNear vectors) and restricted by 1 from above.

Context-independent features:

Formal resemblance feature (**B**eginning **S**imilarity, **BS**). Formal resemblance is a natural way for similarity detection. We exploited a simple formal resemblance metric – the same beginning of words. This feature allows recognition of such pairs as *Kyrgyzstan - Kyrgyz, Parliament of Kyrgyzstan - Kyrgyz Parliament* and etc.

The overall score contribution of BS feature has a Boolean value. So, this feature could add one point to the whole score of a candidate pair.

External resource similarity feature (**T**hesaurus **S**imilarity, **TS**). There are a lot of existing and pre-defined resources, which could give additional information about relations between words and phrases. Such information can be used in thematic nodes construction and make their recognition more reliable. Moreover, it is known that some types of relations between words and phrases are widely used for the text connectivity (for example, such relations as synonymy, hyponym-hypernym, meronym-holonym). To compute TS feature we utilized information from Russian language thesaurus RuThes [23]. Only directly established thesaurus relations (without any inference) were considered (synonyms, hyponyms, hypernyms, meronyms (parts), holonyms (wholes)), but what types of relations are the most appropriate for this purpose would be studied in our further work.

TS feature has also a Boolean value - 0 or 1.

The values of the similarity features for each pair of expressions are summed up to their overall similarity score. Each feature can add from 0 up to 1 point to the overall score. So, the score value is a decimal number located between 0 (minimum similarity) and 5 (maximum similarity).

Additionally, we use **SPlus** feature reflecting similarity through a third-party object or so called “plus one similarity”. If an expression A is similar to an expression C and an expression B is similar to C then we postulate that SPlus condition is satisfied for A and B expressions. By “similar” in this case we understand fulfilment of the following two conditions:

- C >= 0 (from (2) formula)
- One of the features BS, TS or SC has TRUE value.

So, if a candidate pair has no BS and TS features satisfied, but SPlus condition has TRUE value then an additional 0.5 point is added to the overall similarity score. At

last the pairs of similar expressions are ordered in the similarity score decrease, constructing the similarity ranking.

An instance of such ranking for the example cluster is provided in Table 2 (several pairs from the top of the list before the first iteration of the algorithm).

4.4. Algorithm

The algorithm constructs thematic nodes from the most similar pairs of expressions.

The supposed structure of the thematic nodes is as follows:

- a textual expression can belong to one or two thematic nodes; double links to thematic nodes provide the possibility to represent different aspects of the expression or its lexical ambiguity;
- every thematic node has its main element – the thematic center, which belongs only to one thematic node. The thematic center is the most frequent expression among thematic node elements.

The thematic node construction consists of the following steps:

- The pair of expressions with the maximal similarity score is taken;
- The most frequent element of the pair absorbs the second element with all its occurrences and contexts and becomes the representative of the pair, that is the thematic center of a new thematic node;
- The second participant of the pair can further be joined in a similar manner to another thematic node;

Merging of thematic nodes consisting from several elements is fulfilled in the same way as single expressions. The center of a more frequent thematic node becomes the center of a new, merged thematic node.

On the whole, each iteration of the algorithm consists of three main steps:

1. Candidate pairs ranking
2. Top-ranked pair selection
3. Joining procedure

The iterative process proceeds until the top-ranked pair score would be less than a pre-defined threshold. For example, a thematic node with the main expression *Manas base* is constructed as follows (top-ranked pairs at various iterations are presented; a more frequent object is the first object in a pair):

Iteration 2: ***airbase*** <- *Manas airbase*
Iteration 4: ***Manas base*** <- *Manas airbase*
Iteration 5: (***Manas base****, Manas airbase)* <- *base*
Iteration 6: (***Manas base,*** *Manas airbase, base)* <- *Manas*
Iteration 7: (***Manas base****, Manas airbase, base, Manas* <- ***(airbase,*** *Manas airbase)*
Iteration 41: (*Manas base**, Manas airbase, base, Manas, airbase)*** <- *base closure*

Iteration 51: (***Manas base****, Manas airbase, base, Manas, airbase, base closure) <- base withdrawal*

The following thematic nodes were obtained as a result of the described algorithm for the example cluster. We present top ten the most frequent thematic nodes ordered by frequency without any correction, the thematic centers are highlighted by the bold font (translation from Russian):

Manas base: *Manas airbase, base, Manas, airbase, base closure, base withdrawal*
Kyrgyzstan: *Kyrgyz, Parliament of Kyrgyzstan, Kyrgyz parliament, Kyrgyz president Kurmanbek Bakiyev, Bishkek, President of Kyrgyzstan*
USA: *United States, American (noun), American (adj), Pentagon, American military*
Deputy: *Paliament deputy, Legislator, Parlamentarian, Parliament, Parliamentary Committee*
Soldiers: *Military contingent, troops, Military base, American military, Martial contingent, Military-transport aircraft*
Country: *Territory of country, Russia, State, Russian, Territory of the republic*
Manas airport: *Manas international airport, Manas, Airport*
Withdrawal: *Airbase withdrawal, Closure*
Decision: *Government decision*
USA agreement denunciation: *Agreement denunciation, Denunciation, Contract denunciation, Agreement termination, Contract termination*

5. Summary-Based Evaluation

The main purpose of the proposed method was to improve the overall performance of various news cluster automatic processing tasks.

We selected the multi-document summarization task as the basis of evaluation; we suppose that constructed thematic nodes can allow avoiding undesirable repetitions in summary sentences and improve the quality of generated summaries.

In general, summarization is a task of creating a brief summary of text or a set of related texts. News cluster summarization is widely used in news services, such as Google.News, Yandex.News, Rambler.News etc. These services collect information from multiple news sources, divide this information into thematic categories (news clusters), process this information and afterwards present short texts describing a specific event to end-users of the service.

A lot of summarization algorithms have been developed. Some of them are comparatively simple and based on frequency features only [21]. Others exploit additional semantic information from the pre-defined resources [22] and use more

sophisticated algorithms for sentence ranking and selection [23]. And there is set of modern algorithms, which employ probabilistic language models (such as Latent Dirichlet Allocation, LDA) for summary creating and present state-of-the-art results along with other approaches [24]. Such summarization approaches are usually based on hierarchical Latent Dirichlet Allocation model (hLDA) [25], which is built on sentence level (not on document level as in classic LDA). It allows capturing of expected topic distributions in given sentences directly from the model. It is important, because we are considering extractive summarization approaches and sentence is an atomic unit in this case. Besides, news clusters could contain a relatively small number of documents, which may limit the variability of topics if they are evaluated on the document level.

Celikyilmaz et al. [26] propose to construct a hierarchical tree structure of candidate sentences. Each sentence is represented by a path in the tree, and each path can be shared by many sentences. The assumption is that sentences sharing the same path should be more similar to each other because they share the same topics. The tree-based sentence scoring and ranking algorithm is also provided in this work.

For our evaluation, we selected one of the most well-known summarization algorithms – Maximal Marginal Relevance (MMR) [21]. We substitute initial words in the cluster sentences with corresponding thematic nodes and suppose that this generalization operation can improve generated summaries.

5.1. MMR Method

Maximal Marginal Relevance Multi-Document summarization is a classic purely extractive summarization method, which is based on Maximal Marginal Relevance concept proposed for information retrieval [21]. In the original version it is a query-oriented summarization algorithm, but there is a variant of MMR for general summarization too.

MMR criterion means that the best sentence for a summary has to be maximally similar to the user query (or the whole text in case of general summarization) and maximally different from the already selected sentences of the summary.

The summary is constructed incrementally from a list of ranked sentences; the sentence which maximizes MMR is chosen at each iteration:

$$MMR = \arg\max_{s \in S} \left[\lambda \cdot Sim_1(s, Q) - (1 - \lambda) \cdot \max_{s_j \in E} Sim_2(s, s_j) \right]$$

where S is the set of candidates sentences and E is the set of selected sentences; λ represents an interpolation coefficient between sentence relevance and non-redundancy; Sim_1 is the similarity metric used in document retrieval and relevance ranking between sentences (documents) and a query; and Sim_2 can be the same as

Sim_1 or a different metric. In our work we used the classic cosine similarity metric as Sim_1 and Sim_2.

5.2. Pyramid Evaluation Method

Evaluation of automatically generated summaries is a very complicated procedure. The summary evaluation involves different aspects, the main of which are the summary content and coherence. In 2005 an algorithm for summary content evaluation – the Pyramid method was proposed [27]. The algorithm was successfully used in the large-scale evaluation of competitive summarization systems [11] and in our work we also utilized this method.

The method is based on extraction of all "information nuggets" from expert (manually created by experts) summaries, or Summary Content Units (SCUs). SCU describes some fact, which expert summaries take in. Therefore, an automatic summary has to reflect this fact too. Here is an example of summary content unit and its occurrences in different documents of news cluster from [27]:

SCU: *Mini-submarine trapped underwater*
contr1: *mini-submarine... became trapped... on the sea floor*
contr2: *a small... submarine... snagged... at a depth of 625 feet*
contr3: *mini-submarine was trapped... below the surface*
contr4: *A small... submarine... was trapped on the seabed*

The number of contributors (contr) is equal to the weight of the SCU, i.e. an SCU from four contributors has a weight of 4, an SCU from 3 contributors has the weight of 3 and etc. It means that an essential step in summary evaluation by the Pyramid method is creation of several expert summaries (4 summaries at DUC/TAC conference) and manual selection of content units from them. Each SCU after this process obtains a weight, which equals the number of expert summaries, where this SCU occurred.

So, all found summary content units form a pyramid. The upper levels are usually occupied by a comparatively small amount of the most significant summary content units. A lot of less important information units are placed at the lower levels of the pyramid. SCUs pyramid construction is a preliminary step in the summary evaluation. When this step is done, each automatic summary could be assessed for the presence of SCUs from the formed pyramid and the final summary score could be calculated on the basis of the following formula:

$$Sum\_Score = \frac{[Found\_SCU\_Weight]}{[Sum\_SCU\_Weight]} \quad (2)$$

where Found_SCU_Weight – the total weight of all SCUs, presented in a given automatic summary, Sum_SCU_Weight – the total weight of all SCUs, determined for the current cluster. Accordingly, the final Pyramid score for an automatic

summary is its total SCU weight divided by the maximum SCU weight available to a summary of average length (where the average length is determined by the mean SCU count of the expert summaries for this topic). This way of assessment reflects the coverage of expert SCUs by an automatic summary with taking into account SCU different weights.

The Pyramid method for summary content evaluation allows measuring the information coverage by automatic summary, regardless the synonyms and paraphrases used in news cluster documents.

5.3. Evaluation Procedure

To evaluate our approach we apply MMR summarization method to different input data. The original version of MMR method considers an input text (or texts) as a bag-of-words. No information about multiword expressions and related expressions is exploited. Our idea was to add such information to the input data and to examine the results of MMR algorithm. Accordingly, four variants of the input data structure were investigated:

1. Simple bag-of-words model with no additional information. It is a classic input data for MMR method. This version is considered as a baseline.
2. Bag-of-words model with information about multiword expressions added. All consecutive words included to a multiword expression, are considered as a single word. This model is important for evaluation of the influence of multiword expressions on the overall performance.
3. Bag-of-words model with thesaurus information added. Thesaurus-based thematic nodes were described in [12]. Elements of the same thematic node are substituted with their thematic center and considered as the same input word.
4. Bag-of-words model with cluster-based thematic nodes information (assembled by the proposed algorithm) added. Thematic node elements are also considered as the same input word (thematic center of this thematic node) with the weight propotional to the similarity score between a given element and the thematic center.

To test the MMR method with different input data models we took 10 news clusters on various topics. The Pyramid evaluation procedure was performed. For this purpose two-four expert summaries were created for each news cluster by professional linguists. Summary Content Units were manually extracted from these summaries. On the whole, 129 SCUs were extracted. Each SCU has the weight equal to the number of expert summaries, where it occurred. Thus, SCU pyramid was assembled. Afterwards, each automatic summary was manually assessed for SCUs references and the score of the examined summary was calculated (see previous section).

5.4. Results

Table 3 shows evaluation results of MMR method with various input data.

We can see that the adding of multiword expressions to the simple bag-of-words model decreases the overall performance of the MMR algorithm. This is due to the appearance of low-frequent multiword expressions and therefore the increase of diversity in input data. Low results were achieved by the MMR method on only thesaurus-based input data, which possibly can be explained by the differences between information described in the thesaurus and the real cluster structure.

The best results were obtained on cluster-based thematic nodes, taking into account multiword phrases and similarity between words and expressions

Input data model	Score
Simple bag-of-words	57,8%
Bag-of-words with multiword expressions	53,1%
Bag-of words with thesaurus information	52,6%
Bag-of-words with cluster-based thematic nodes	**59,8%**

Table 3: MMR method evaluation results for various input data models

6. Conclusion

In this paper we have proposed to use the discourse structure of natural language texts to extract sets of semantically similar expressions representing different participants of the text story – thematic nodes. We described two experiments on news clusters: multiword expression extraction and cluster-based thematic node construction. In addition to known methods of context comparison, we exploited the co-occurrence frequency in neighboring sentences to detect the semantic similarity of language expressions. We also combined several heterogeneous features for the thematic node construction:

- Formal resemblance features
- Information from the pre-defined resource (Russian language thesaurus RuThes [17])
- Context-based features

The evaluation of the introduced method showed that the cluster-based thematic nodes can improve the overall performance of the multi-document summarization algorithm.

In future we are going to use cluster-based thematic nodes for various operations as cluster refining, novelty detection, sub-clustering and etc.

References

[1] Loukachevitch N.: Multigraph representation for lexical chaining. In: Proceedings of SENSE workshop, pp. 67-76 (2009)

[2] Hirst G., St-Onge D.: Lexical Chains as representation of context for the detection and correction malapropisms. In: WordNet: An electronic lexical database and some of its applications / C. Fellbaum, editor. Cambrige, MA: The MIT Press (1998)

[3] Turdakov D., Lizorkin D. HMM Expanded to Multiple Interleaved Chains as a Model for Word Sense Disambiguation. In: Proceedings of the 23rd Pacific Asia Conference on Language, Information and Computations, pp. 549–559 (2009)

[4] Blei D., Ng A., Jordan M. Latent Dirichlet Allocation. In: Journal of Machine Learning Research, 3:993-1022 (2003)

[5] Griffiths T., Steyvers M. Finding scientific topics. In: Proceedings of the National Academy of Sciences of the United States of America, Vol. 101, No. Suppl 1. (6 April 2004), pp. 5228-5235 (2004)

[6] Allan J.: Introduction to Topic Detection and Tracking. In: Topic detection and tracking, Kluwer Academic Publishers Norwell, MA, USA, pp. 1-16 (2002)

[7] Duame H., Marcu D.: A large Scale Exploration of Global Features for a Joint Entity Detection and Tracking Model. In: Proceedings of Human Language Conference and Conference on Empirical Methods in Natural Language Processing, pp. 97-104 (2005)

[8] Yang H., Callan J.: A metric-based framework for automatic taxonomy induction. In: Proceedings of ACL-2009 (2009)

[9] Dang V., Xue X., Croft B. Context-based Quasi-Synonym Extraction. CIIR Technical Report (2009)

[10] [Barzilay R., McKeown K.: Extracting Paraphrases from a Parallel Corpus. In: Proceedings of the 39th Annual Meeting on Association for Computational Linguistics (2001)

[11] Passonneau R.J., Nenkova A., McKeown K.R., Sigelman S.: Applying the pyramid method in DUC 2005. In: Proceedings of the Document Understanding Conferences (DUC'2005), Vancouver, Canada (2005)

[12] Doddington G., Mitchell A., Przybocki M., Ramshaw, L., Strassel S., Weishedel R.: The Automatic Content Extraction (ACE): Task, Data, Evaluation. In: Proceedings of Fourth International Conference on Language Resources and Evaluation, LREC 2004 (2004)

[13] Barzilay R., Lee L.: Learning to Paraphrase: an Unsupervised Approach Using Multiple Sequence Alignment. In: Proceedings of HLT/NACCL-2003 (2003)

[14] Dolan B., Quirk Ch., Brockett Ch.: Unsupervised Construction of Large Paraphrase Corpora: Exploiting Massively Parallel News Sources. In: Proceedings of COLING-2004 (2004)

[15] Dijk van T.: Semantic Discourse Analysis. In: Teun A. van Dijk, (Ed.), Handbook of Dis-course Analysis, vol. 2., pp. 103-136, London: Academic Press (1985)

[16] Hasan R.: Coherence and Cohesive harmony. J. Flood, Understanding reading comprehension, Newark, DE: IRA, pp. 181-219 (1984)

[17] Loukachevitch N., Dobrov B.: Evaluation of Thesaurus on Sociopolitical Life as Information Retrieval Tool. In: M.Gonzalez Rodriguez, C. Paz Suarez Araujo (Eds.), Proceedings of Third International Conference on Language Resources and Evaluation (LREC2002), Vol.1, pp.115-121 (2002)

[18] Dobrov B., Pavlov A.: Basic line for news clusterization methods evaluation. In: Proceedings of the 5-th Russian Conference RCDL-2010 (2010) (in Russian)

[19] Witten I., Paynter G., Frank E., Gutwin C., Newill-Manning C.: KEA: practical automatic keyphrase extraction. In: Proceedings of the fourth ACM conference on Digital Libraries (1999)
[20] Alexeev A., Loukachevitch N. Automatic detection of near-synonyms in news clusters. In: Computational Linguistics and Intelligent Technologies: Proceedings of the International Conference Dialog`2011, pp. 32-40 (2011)
[21] Carbonell J., Goldstein J.: The use of MMR, diver-sity-based reranking for reordering documents and producing summaries. In: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Melbourne, Australia, pp. 335-336 (1998)
[22] Dobrov B., Loukachevitch N.: Summarization of News Clusters Based on Thematic Representation. In: Computational Linguistics and Intelligent Technologies: Proceedings of the International Conference Dialog`2009, pp. 299-305 (2009) (In Russian)
[23] Li J., Sun L., Kit C., Webster J.: A Query-Focused Multi-Document Summarizer Based on Lexical Chains. In: Proceedings of the Document Understanding Conference DUC-2007 (2007)
[24] Haghighi A., Vanderwende L.: Exploring Content Models for Multi-Document Summarization. In: Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the ACL, Boulder, Colorado, pp. 362–370 (2009)
[25] Blei D., Griffiths T., Jordan M., Tenenbaum J. Hierarchical topic models and the nested chinese restaurant process. In: Neural Information Processing Systems (NIPS) (2003)
[26] Celikyilmaz A., Hakkani-Tur D. A Hybrid Hierarchical Model for Multi-Document Summarization. In: Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, Uppsala, Sweden, pp. 815–824 (2010)
[27] Harnly A., Nenkova A., Passonneau R., Ram-bow O.: Automation of summary evaluation by the pyramid method. In: Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP'2005), Borovets, Bulgaria (2005)

Linkset–based Data Integration System for LOD Space

Kuznetcov Konstantin
K.Kuznetcov@gmail.com
Lomonosov Moscow State University

Abstract. This paper describes research-in-progress work on data integration system in Linked Open Data space. Proposed system uses concept of RDF identity links to interlink heterogeneous local data sources and integrate them into global data space.

Key Words: data integration; linked open data; semantic web; link generation.

1. Introduction

For the last few decades data integration has been one of the most actual problems of computer science. With the development of IT industry countless data sources emerged in the Internet. These data sources are heterogeneous in all possible ways. Effective usage of such data sources is impossible without automatical tools for data search, retrieval, publishing and transformation.

The original hypertextual web didn't suit well for automatical processing of data from heterogeneous sources spread across the web. This led to emergence of various microformats and web APIs, and finally the concept of Semantic Web. Sematic Web implies usage of standard stack of data formats and technologies intended to support data accumulation, structuring and exchange across the web. The most important of these technologies are RDF, RDFS, OWL and SPARQL.

From the practical point of view one of the most interesting Semantic Web initiatives is Linking Open Data project [8]. This project aims for quantitative filling of the web with data structured according to Semantic Web standards and for interlinking of semantic data sources. As a result, a global Linked Open Data space should be established, similar to hypertextual web of linked documents. Publishing data in Linked Open Data space encourages reuse of data, decreases data redundancy, maximizes its (real and potential) inter-connectedness and enables network effects to add value to data. Data providers can benefit from publishing their data in LOD space. Unfortunately, the process of publishing is not that simple and consists of several steps. Small organizations often cannot afford to transform their data into LOD- acceptable form and then support the published dataset. To the moment, there is no special software to support the full cycle of publishing and

managing linked open datasets. A system to support integration of data from independent data sources into the web of linked data is required.

2. Related work

To the moment, there are quite few solutions that support variuos steps required to include one's data into Linked Open Data space, even though they are based on existing hypertext web technologies. And there is no system that includes all the functionality recommended by LOD project. The most complex solution available now is Virtuoso Universal Server [11] platform. It provides tools for representing data from different sources (relational databases, RDF-storages, Web APIs, etc.) as a single virtual database and supports RDF data publishing. Virtuoso offers SPARQL access to its data and such features as RDF-crawler and simple reasoner. Virtuoso can be extended in multiple ways, e.g. published RDF-data can be accompanied with voiD descriptors. Unfortunately all the extensions that are useful for linked data publishing are made on instrumental level and not on data model level. Virtuoso is commercial software with limited open-source community edition. Among other open-source solutions it is worth to mention D2R Server [3], which supports RDF data publishing from relational databases and SPARQL querying. The MASTRO [4] and the data integration system developed in Dorodnicyn Computing Centre of RAS [2] provide richer semantic formalisms compared to D2R and Virtuoso. However, the first system is bound to a single federative database and the second one implies that its sources share some common URIs. And both of them don't provide any means for publishing or interlinking with other RDF datasets.

However, both Virtuoso and D2R Server do not go beyond simple RDF publishing. Their RDF-resource interlinking capabilities are limited to URI generation from templates. In many cases using such templated URIs cannot expose identity relations between RDF resources from different datasets and truly interlink these datasets. There are several applications for RDF data interlinking and link supporting, such as SILK [12], LIMES, SemMF and DSNotify. But at the moment none of these applications provide means for integration of one's RDF into the whole Linked Data space. I.e. there are no toos that can automatically discover new related datasets in the web, the set up and support links to the resources in these datasets. Some proposals for such systems are made in [10].

The possibilities of non-trivial usage of generated linksets are yet to be explored. Very few applications take advantage of this feature of Linked Open Data space. It is worth to mention SPLENDID system [6] here, which uses linksets statistics to optimize federative SPARQL queries. Some semantic search engines also utilize voiD descriptions of linksets.

3. Problem statement

This article proposes a concept of automated data integration system in Linked Open Data space. Proposed system should

- Form the single dataset from multiple heterogeneous sources of structured or unstructured information in similar knowledge domain and support/update formed dataset;
- Discover and store links between resources from system dataset and resources from different Linked Open Data sets available on the Internet in RDF format, as well as implicit links between resources within system dataset;
- Publish system dataset in the Internet in RDF format and provide access to it via user interface and API;
- Provide users and external applications with unified query interface to all of system's data sources;
- Support different data source types (including relational databases and SPARQL endpoints) and support on-fly connection of new data sources;
- Include flexible ontology of knowledge domain that follows Linking Open Data project recommendations and can be extended to support new data sources.

4. System architecture overview

Proposed system will follow modular architecture and will consist of following components:

- Ontology of informational objects and links of interest to system data consumers and providers;
- Linking subsystem, that should discover and store links between resources from system's data sources and/or resources from external Linked Data sources;
- Publishing subsystem, which should provide users and applications with access to resources from system's dataset according to LOD project recommendations;
- Data integration subsystem, which will contain mechanism to uniquely identify system's resources both within system and in Linked Open Data space and provide uniform access to all system's resources. This subsystem will include a set of adapters that provide unified SPARQL access to system's data sources of different types (relational databases, Web APIs, etc.);
- Harvesting and extraction subsystem with a set of harvester components, which will gather data from system's sources of unstructured data (text files, scanned documents, etc.), transform it into structured form and store it. This system is a subject of a future work.

4.1. Ontology

The system uses OWL ontology to semantically organize objects and links that match the concepts of interest from knowledge domain of system's data sources.

Ontology consists of core terms and imported modules which can be added in case when some resources in newly added data source require more precise definition. In Linked Open Data space ontology serves as system's data vocabulary, it is used to establish terminological outgoing links to external datasets and allows external applications to discover metadata to establish ingoing links. Following the principles of Linked Open Data, ontology is annotated in human language with such terms as rdfs:label or rdfs:comment. Ontology's terms should be defined in URI namespace controlled by the system. Ontology adapts common Linked Data vocabularies such as Dublin Core, FOAF, vCard, PRISM, SIOC, Creative Commons, BibTex, Schema.org. Core of ontology is based on ENIP RAS ontology.

4.2. Publishing subsystem

Publishing subsystem will serve as an entry point to the system for human users and Linked Data applications. It should dereference URIs of system's resources, i.e. return descriptions of the object or concept identified by these URIs. It can be achieved by using a mechanism called content negotiation. Depending on HTTP GET request header, publishing subsystem will return either HTML representation or RDF/XML (as required by Linked Data applications) representation of the resource.

Publishing subsystem will receive data from data integration subsystem. For dereferencing resource URI, following information should be requested from data integration subsystem:

- All the literal values of resource, all incoming and outgoing RDF links. This information can be retrieved with simple SPARQL queries with patterns {<URI> ?x ?y} and {?x ?y <URI>};
- Most likely the results of these simple requests will contain URIs of other system's resources. Linked Data applications often traverse URIs they find in RDF documents. Therefore to reduce the number of HTTP requests publishing subsystem should extend aforementioned requests to some depth, or by applying some explicitly stated rules;
- Information on ontology class to which the requested resource belongs and all its ancestors;
- Information on the dataset to which this resource belongs;

All the information retrieved from data integration subsystem will be represented as a set of RDF triples. In case of RDF document these triples will be merged into resulting RDF/XML document and returned to client. In other case the triples will be published as HTML+RDFa document generated from template. These templates can be specified in general form and then redefined for specific classes.

4.3. Data integration subsystem

A data integration subsystem will provide other subsystems or external agents with uniform access interface to all of the system's data sources. Requested information should be specified with SPARQL query. This subsystem will be responsible for

presenting system's data as single dataset in Linked Open Data space. There are several approaches to data integration systems – data warehousing, data mediation, peer-to-peer systems. Proposed system is supposed to work with multiple strongly autonomous data sources; therefore it adapts data mediation architecture. The drawback of such systems (e.g. Virtuoso) is huge amount in network interactions required to produce query answer. Proposed system uses Linked Open Data principles to reduce this drawback.

Data sources will be connected to the system via adapters, which are SPARQL endpoints capable of querying data sources in terms of system ontology. These adapters should be generic, configurable components (e.g. JDBC adapter, REST API adapter). As opposed to existing data integration systems with semantic capabilities (e.g. Virtuoso with its Sponger cartridges), resources from different data sources won't be merged into single dataset by providing same URI to identical resources. Instead, in the spirit of Linked Open Data, every data source should be considered to contain unique resources and get its own sub-namespace (like http://<system_URL>/datasets/<source_id>). Adapter should confront every resource from its data source with HTTP URI from this namespace. Therefore we will be able to track resource origin by its URI. When new data source will be added to the system, its adapter will be configured by specifying generic adapter settings (e.g. JDBC connection string), general Dublin Core description of the source, topic of interest categorization, licensing information, etc. Adapter configuration also includes the set of ontology classes and properties to which data in these sources belongs. This information can be entered manually or obtained with SPARQL ASK request. Next, adapter configuration will be published as voiD [1] descriptor of dataset. All such datasets are subsets (in terms of voiD) of system's whole dataset. However, all of them will be accessed via single SPARQL endpoint. Such structure preserves autonomy and independence of data sources while integrating them all together in Linked Open Data space.

Execution of queries in data integration subsystem will be carried out as follows. The first step is a SPARQL query rewriting according to the axioms of ontology, as described in [2]. Then algebraic query optimization techniques are applied. The result of this phase in terms of descriptive logic is the union of conjunctive queries with simple constraints. In the second step the set of relevant data sources for each atom of each conjunctive query is determined according to configuration of adapters. If there are no data sources relevant to an atom, the entire conjunctive query is dropped. As a result, a union of conjunctive queries with atoms of different data sets will be obtained.

Traditionally, the next step in data mediation process is construction of the physical query plan and its execution. During execution of query with atoms related to different data sources the results of subqueries to these data sources are joined. However, in the proposed system subquery results can be joined on literal field values and not on the URIs, because data sources are presented in a form of independent Linked Open Data sets and do not share common URIs. If subqueries

to different data sources are to be joined on URIs we will have to use the sets of links generated by linking subsystem between these data sources. Each conjunctive query is a graph pattern with vertices being either literal values or the URIs or variables, and edges are labeled with predicates in terms of different data sources. If two adjacent edges are labeled with predicates from different sources, it is necessary to refer to the linkset for this pair of sources and select resource pairs that satisfy a given part of graph pattern. By performing this operation on all the links in the conjunctive query, we will obtain the set of resource URIs that satisfies part of the pattern that defines relationships between different data sources. Then the subquery parts related to specific data sources will be executed by adapters with corresponding join variables being replaced by URIs from linksets. On this step traditional query optimization techniques can be applied again.

4.4. Linking subsystem

RDF documents published in the Linked Open Data space are required to contain outgoing links. These outgoing links are RDF triples with the subject being the URI of the resource from the local namespace and the URI of the object and / or predicate belonging to the namespace of another dataset. The most important type of outgoing links is identity links that point at URI aliases used by other data sources to identify the same real-world object or abstract concept. Identity links can use such predicates as owl: sameAs, rdfs: seeAlso or special SKOS terms. Although the uses of predicate owl: sameAs in the LOD space are often contrary to the semantics of OWL [7], its use is recommended by W3C Technical Architecture Group. Linking subsystem will be responsible for the discovery, storage and support of identity links. Properties of the link include pair of URIs, link generation time and method, date of last link check and similarity factor. When the link is published either owl:sameAs or rdfs:seeAlso predicate is used in the triple depending on similarity factor value.

Linking subsystem will work as follows. In the first step, the two data sources to be interlinked are found. For this pair an initially empty voiD linkset is created and published. When new data sources is added to the system the linksets between this new data source and all existing data sets from other sources are automatically created. A linkset between internal dataset and external Linked Open Data set will be created in one of the following cases:

- The user can manually select a pair of datasets for linking;
- Relevant datasets can be discovered using HTTP referrer technique described in [9];
- Relevant datasets can set be discovered by linking subsystem itself by traversing links in external dataset that is already linked to one of internal datasets.

When two target datasets for interlinking will have been selected, the subsystem will clusterize datasets by classes and determines pairs of clusters to be interlinked. This should be done to reduce the number of pairwise comparisons of datasets elements. In the case of two internal datasets both of them are described by the same

ontology, so that pairs of clusters contain instances of same ontology classes. In the case of linking to an external dataset the subsystem might select pairs of classes with help of different ontology mapping techniques [5], as well as using discovered or manually specified ontology mapping rules.

The third and final step of interlinking involves pairwise comparison of clusters elements to detect pairs of identity relations. These relations will be detected using SILK LSL language rules. In the case of internal data sources, rules will be declared together with the ontology and determine which instances of the same class are identical. In the case of an external dataset rules will be either specified manually, or derived from the existing rules and ontology mapping rules.

Complete binding is achieved by pairwise comparison of all elements of all datasets (both internal and external), but in practice such comparison is impossible. Link generation optimization requires additional study.

5. Conclusion

This paper proposes a concept of data integration system orientated towards Linked Open Data space. The novelty of this concept lies in its hybrid approach; the system proposed combines data mediation and data warehousing approaches by using locally stored linksets as indexes for a search engine hasn't been implemented yet. To the author's knowledge, such method hasn't been implemented yet. Besides, while there are works dedicated to bringing single data sources into the LOD space or dealing with multiple already present sources in LOD space, the idea of bringing multiple data sources into LOD space via single data integration system has received very little attention.

Currently, the proof-of-concept system is being developed in CC RAS as a part of a practical project dedicated to integration of data on protected sites and animal species. While participating in a group on this project, the author is working on query answering algorithms in presence of linksets. As a result of this project, a large set of data on national parks should emerge in the LOD space, and if incoming links from external datasets appear, the project would be considered to be successful.

Future works on this project might include the study of link network generation and support algorithms. The system can also be extended with modules to access external Semantic Web resource aggregators (sig.ma) and semantic search engines (sindice.com). Also, additional studies in the management of licensing and data access in the context of the Linked Open Data are required.

References

[1] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing linked datasets. In Proceedings of the WWW2009 Workshop on Linked Data on the Web, 2009.

[2] A. A. Bezdushny. Formal Model of Ontology-Based Data Integration Systems. Novosibirsk, 2008

[3] C. Bizer, R. Cyganiak. D2RQ — Lessons Learned. Position paper for the W3C Workshop on RDF Access to Relational Databases, 2007. http://www.w3.org/2007/03/RdfRDB/papers/d2rq-positionpaper/
[4] D. Calvanese, G. De Giacomo, D. Lembo et al. The MASTRO system for ontology-based data access. Semantic Web Journal, volume 2, number 1, pages 43-53, 2011
[5] J. Euzenat, A. Ferrara, et al. First results of the ontology alignment evaluation initiative 2011. In Proc. of 6th Ontology Matching Workshop (OM‘11), at International Semantic Web Conference (ISWC‘11), Bonn, Germany, 2011.
[6] O. Gorlitz, S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. Proceedings of the 2nd International Workshop on Consuming Linked Data, Bonn, Germany, 2011.
[7] H. Halpin, P. Hayes, J. McCusker, D. Mcguinness, and H. Thompson. When owl:sameas isn't the same: An analysis of identity in linked data. In Proceedings of the 9th International Semantic Web Conference, 2010
[8] T. Heath and C. Bizer. Linked Data: Evolving the Web into a Global Data Space (1st edition). Synthesis Lectures on the Semantic Web: Theory and Technology, 1:1, 1-136. Morgan & Claypool, 2011. http://linkeddatabook.com/editions/1.0/
[9] H. Muhleisen and A. Jentzsch: Augmenting the Web of Data using Referers Linked Data on the Web (LDOW2011), Mar. 2011
[10] A. Nikolov and M. d'Aquin. Identifying Relevant Sources for Data Linking using a Semantic Web Index, Workshop: 4th Workshop on Linked Data on the Web (LDOW 2011) at 20th International World Wide Web Conference (WWW 2011), Hyderabad, India, 2011.
[11] Virtuoso Universal Server, 2011. http://virtuoso.openlinksw.com/
[12] J. Volz, C. Bizer, M. Gaedke, and G. Kobilarov. Discovering and maintaining links on the web of data. In Proceedings of the International Semantic Web Conference, pages 650–665, 2009

Comparative Study Parallel Join Algorithms for MapReduce environment

A. Pigul
m05pay@math.spbu.ru
Saint Petersburg State University

Abstract. There are the following techniques that are used to analyze massive amounts of data: MapReduce paradigm, parallel DBMSs, column-wise store, and various combinations of these approaches. We focus in a MapReduce environment. Unfortunately, join algorithms is not directly supported in MapReduce. The aim of this work is to generalize and compare existing equi-join algorithms with some optimization techniques.

Key Words: parallel join algorithms, MapReduce, optimization.

1. Introduction

Data-intensive applications include large-scale data warehouse systems, cloud computing, data-intensive analysis. These applications have their own specific computational workload. For example, analytic systems produce relatively rare updates but heavy select operation with millions of records to be processed, often with aggregations.

Applications for large-scale data analysis use such techniques as parallel DBMS, MapReduce (MR) paradigm, and columnar storage. Applications of this type process multiple data sets. This implies need to perform several join operation. It's known join operation is one of the most expensive operations in terms both I / O and CPU costs.

Unfortunately, join algorithms is not directly supported in MapReduce. There are some approaches to solve this problem by using a high-level language PigLatin, HiveQL for SQL queries or implementing algorithms from research papers. The aim of this work is to generalize and compare existing equi-join algorithms with some optimization techniques.

This paper is organized as follows the section 2 describe state of the art. Join algorithms and some optimization techniques were introduced in 3 section. Performance evaluation will be described in 4 section. Finally, future direction and some discussion of experiments will be given.

2. Related Work

2.1. Architectural Approaches

Column storage is one of the architectural approaches to store data in columns, that the values of one field are stored physically together in a compact storage area. Column storage strategy improves performance by reducing the amount of unnecessary data from disk by excluding the columns that are not needed. Additional gains may be obtained using data compression. Storage method in columns outperforms row-based storage for workloads typical for analytical applications, which are characterized by heavy selection operation from millions of records, often with aggregation and by infrequent update operation. For this class of workloads I/O is major factor limited the performance. Comparison of column-wise and row-wise stores approaches is presented in [1].

Another architectural approach is a software framework MapReduce. Paradigm MapReduce was introduced in [11] to process massive amounts of unstructured data.

Originally, this approach was contrasted with a parallel DBMS. Deep analysis of the advantages and disadvantages of these two architectures was presented in [25,10].

Later, hybrid systems appeared in [9, 2]. There are three ways to combine approaches MapReduce and parallel DBMS.

- MapReduce inside a parallel DBMS. The main intention is to move computation closer to data. This architecture can be exemplified with hybrid database Greenplum with MAD approach [9].
- DBMS inside MapReduce. The basic idea is to connect multiple single node database systems using MapReduce as the task coordinator and network communication layer. An example is a hybrid database HadoopDB [2].
- MapReduce aside of the parallel DBMS. MapReduce is used to implement an ETL produced data to be stored in parallel DBMS. This approach is discussed in [28] Vertica, which also supports the column-wise store.

Another group of hybrid systems combines MapReduce with column-wise store. MapReduce and column-wise store are effective in data-intensive applications. Hybrid systems based on this two techniques may be found in [20,13].

2.2. Algorithms for Join Operation

Detailed comparison of relational join algorithms was presented in [26]. In our paper, the consideration is restricted to a comparison of joins in the context of MapReduce paradigm.

Papers which discuss equi-join algorithms can be divided into two categories which describe join algorithms and multi join execution plans.

The former category deals with design and analyses join algorithm of two data sets. A comparative analysis of two-way join techniques is presented in [6, 4, 21]. The cost model for two-way join algorithms in terms of cost I/O is presented in [7, 17].

The basic idea of multi-way join is to find strategies to combine the natural join of several relations. Different join algorithms from relation algebra are presented in [30]. The authors introduce the extension of MapReduce to facilitate implement relation operations. Several optimizations for multi-way join are described in [3, 18]. Authors introduced a one-to-many shuffling strategy. Multi-way join optimization for column-wise store is considered in [20, 32].

Theta-Joins and set-similarity joins using MapReduce are addressed in [23] and [27] respectively.

2.3. Optimization techniques and cost models

In contrast to the sql queries in parallel database, the MapReduce program contains user-defined map and reduce functions. Map and reduce functions can be considered as a black-box, when nothing is known about these functions, or they can be written on sql-like languages, such as HiveQL, PigLatin, MRQL, or sql operations can be extracted from functions on semantic basis. Automatic finding good configuration settings for arbitrary program offered in [16]. Theoretical designing cost models for arbitrary MR program for each phase separately presented in [15]. If the MR program is similar to the semantics of SQL, it allows us to construct a more accurate cost model or adapt some of the optimization techniques from relational databases. HadoopToSQL [22] allows to take advantage of two different data storages such as SQL database and the text format in MapReduce storage and to use index at right time by transforming the MR program to SQL. Manimal system [17] uses static analysis for detection and exploiting selection, projection and data compression in MR programs and if needed to employ B+ tree index.

New SQL-like query language and algebra is presented in [12]. But they are needed cost model based on statistic. Detailed construction of the model to estimate the I/O cost for each phase separately is given in [24]. Simple theoretical considerations for selecting a particular join algorithm are presented in [21]. Another approach [7] for selecting join algorithm is to measure the correlation between the input size and the join algorithm execution time with fixed cluster configuration settings.

3. Join algorithms and optimization techniques

In this section we consider various techniques of two-way joins in MapReduce framework. Join algorithms can be divided into two groups: Reduce-side join and Map-side join. The pseudo code presented in Listings, where R – right dataset, L – left dataset, V – line from file, Key – join key, that was parsed from a tuple, in this context tuple is V.

3.1. Reduce-Side join

Reduce-side join is an algorithm which performs data pre-processing in Map phase, and direct join is done during the Reduce phase. Join of this type is the most general without any restriction on the data. Reduce-side join is the most time-consuming, because it contains an additional phase and transmits data over the network from one phase to another. In addition, the algorithm has to pass information about source of data through the network. The main objective of the improvement is to reduce the data transmission over the network from the Map task to the Reduce task by filtering the original data through semi-joins. Another disadvantage of this class of algorithms is the sensitivity to the data skew, which can be addressed by replacing the default hash partitioner with a range partitioner.

There are three algorithms in this group:

- General reducer-side join,
- Optimized reducer-side join,
- the Hybrid Hadoop join.

General reducer-side join is the simplest one. The same algorithms are called Standard Repartition Join in [6]. The abbreviation is GRSJ and pseudo code is presented in Listing 1.

This algorithm has both Map and Reduce phases. In the Map phase, data are read from two sources and tags are attached to the value to identify the source of a key/value pair. As the key is not effecting by this tagging, so we can use the standard hash partitioner. In Reduce phase, data with the same key and different tags are joined with nested-loop algorithm. The problems of this approach are that the reducer should have sufficient memory for all records with a same key; and the algorithm sensitivity to the data skew.

```
Map (K: null, V from R or L)
    Tag = bit from name of R or L;
    emit (Key, pair(V,Tag));

Reduce (K': join key, LV: list of V with key K')
create buffers Br and Bl for R and L;
    for t in LV do
        add t.v to  Br or Bl by t.Tag;
    for r in Br do
        for l in Bl do
         emit (null, tuple(r.V,l.V));
```

Listing 1: GRSJ.

```
Map (K:null, V from R or L)
    Tag = bit from name of R or L;
    emit (pair(Key,Tag), pair(V,Tag));

Partitioner(K:key, V:value, P:the number of reducers)
    return hash_f(K.Key) mod P;

Reduce (K': join key, LV: list of V' with key K')
    create buffers B_r for R;
    for t in LV with t.Tag corresponds to R do
        add t.v to B_r;
    for l in LV with l.Tag corresponds to L do
        for r in B_r do
            emit (null, tuple(r.V,l.V));
```

Listing 2: ORSJ.

Optimized reducer-side join enhances previous algorithm by overriding sorting and grouping by the key, as well as tagging data source. Also known as Improved Repartition Join in [6], Default join in [14]. The abbreviation is ORSJ. In Listing 2 pseudo code is shown. In the algorithm all the values of the first tag are followed by the values of the second one. In contrast with the General reducer-side join, the tag is attached to both a key and a value. Due to the fact that the tag is attached to a key, the partitioner must be overridden in order to split the nodes by the key only. This case requires buffering for only one of input sets.

Optimized reducer-side join inherits major disadvantages of General reducer-side join namely the transferring through the network additional information about the source and the algorithm sensitivity to the data skew.

The Hybrid join [4] combines the Map-side and Reduce-side joins. The abbreviation is HYB and Listing 3 describe pseudo code.

```
Job 1: partition the smaller file S
  Map (K:null, V from S)
    emit (Key,V);
  Reduce (K':join key, LV: list of V' with key K')
    for t in LV do
      emit (null, t);

Job 2: join two datasets
  Map (K:null, V from B)
    emit (Key,V);
  init()  //for Reduce phase
    read needed partition of output  om Job 1;
    add it to hashMap(Key, list(V)) H;
  Reduce (K':join key, LV: list of
          V' with key K')
    if(K' in H) then
      for r in LV do
        for l in H.get(K') do
          emit (null, tuple(r,l));
```

Listing 3: HYB

In Map phase, we process only one set and the second set is partitioned in advance. The pre-partitioned set is pulled out of blocks from a distributed system in the Reduce phase, where it is joined with another data set that came from the Map phase. The similarity with the Map-side join is the restriction that one of the sets has to be split in advance with the same partitioner, which will split the second set. Unlike Map-side join, it is necessary to split in advance only one set. The similarity with the Reduce-side join is that algorithm requires two phases, one of them for pre-processing of data and one for direct join. In contrast with the Reduce-side join we do not need additional information about the source of data, as they come to the Reducer at a time.

3.2. Map-Side join

Map-side join is an algorithm without Reduce phase. This kind of join can be divided into two groups. First of them is partition join, when data previously partitioned into the same number of parts with the same partitioner. The relevant parts will be joined during the Map phase. This map-side join is sensitive to the data skew. The second is in memory join, when the smaller dataset send whole to all mappers and bigger dataset is partitioned over the mappers. The problem with this type of join occurs when the smaller of the sets cannot fit in memory.

There are three methods to avoid this problem:

- JDBM-based map join,
- Multi-phase map join,
- Reversed map join.

Map-side partition join algorithm assumes that the two sets of data pre-partitioned into the same number of splits by the same partitioner. Also known as default map join. The abbreviation is MSPJ and Listing 4 describe pseudo code. At the Map phase one of the sets is read and loaded into the hash table, then two sets are joined by the hash table. This algorithm buffers all records with the same keys in memory, as is the case with skew data may fail due to lack of enough memory.

```
Job 1: partition dataset S as in HYB
Job 2: partition dataset B as in HYB
Job 3: join two datasets
  init()   //for Map phase
     read needed partition of output file from Job 1;
     add it to hashMap(Key, list(V)) H;
   Map(K:null, V from B)
     if (K in H) then
          for r in LV do
               for l in H.get(K) do
              emit(null, tuple(r,l));
```

Listing 4: MSPJ

```
Job 1: partition S dataset as in HYB
Job 2: partition B dataset as in HYB
Job 3: join two datasets
   init()  //for Map phase
      find needed partition SP of output file from Job 1;
      read first lines with the same key K2 from SP and add
            to buffer Bu;
   Map(K:null, V from B)
       while (K > K2) do
              read T from SP with key K2;
              while (K == K2) do
                  add T to Bu;
                  read T from SP with key K2;
         if (K == K2) then
              for r in Bu do
                 emit(null, tuple(r,V));
```

Listing 5: MSPMJ.

Map-side partition merge join is an improvement of the previous version of the join. The abbreviation is MSPMJ and pseudo code is presented in Listing 5. If data sets in addition to their partition are sorted by the same ordering, we apply merge join. The advantage of this approach is that the reading of the second set is on-demand, but not completely, thus memory overflow can be avoided. As in the previous cases, for optimization can be used the semi-join filtering and range partitioner.

In-Memory Join does not require to distribute original data in advance unlike the versions of map joins discussed above. The same algorithms are called Map-side replication join in [7], Broadcast Join in [6], Memory-backed joins [4], Fragment-Replicate join in [14]. The abbreviation is IMMJ. Nevertheless, this algorithm has a strong restriction on the size of one of the sets: it must fit completely in memory. The advantage of this approach is its resistance to the data skew because it sequentially reads the same number of tuples at each node. There are two options for transferring the smaller of the sets:

- using a distributed cache,
- reading from a distributed file system.

```
init()  // for Map phase
   read S from HDFS;
   add it to hashMap(Key, list(V)) H;
map (K:null, V from B)
      if (K in H) then
        for l in H.get(K) do
              emit (null, tuple(v,l));
```

Listing 6: IMMJ

```
init()   //for Map phase
   read S from HDFS;
   add it to hashMap(Key, list(V)) H;
map (K:null, V from S)
     add to hashMap(Key, V) H;
close()  //for Map phase
      find B in HDFS
      while (not end B) do
            read line T;
            K = join key from tuple T;
            if (K in H) then
                  for l in H.get(K) do
                      emit(null, tuple(T,l));
```

Listing 7: REV.

The next three algorithms optimize the In-Memory Join for a case, when two sets are large and no of them fits into the memory.

JDBM-based map join is presented in [21]. In this case, JDBM library automatically swaps hash table from memory to disk.

```
The same as IMMJ, but H is implemented by HTree
instead of hashMap .
```

Listing 8: JDBM

```
For part P from S that fit into memory do IMMJ(P,B).
```

Listing 9: Multi-phase map join.

Multi-phase map join [21] is algorithm where the smaller of the sets is partitioned into parts that fit into memory, and for each part runs In-Memory join. The problem

with this approach is that it has a poor performance. If the size of the set, which to be put in the memory is increased twice, the execution time of this join is also doubled. It is important to note that the set, which will not be loaded into memory, will be read many times from the disk.

Idea of Reversed map join [21] approach is that the bigger of the sets, which is partitions during the Map phase, loading in the hash table. Also known as Broadcast Join in [6]. The abbreviation is REV. The second dataset is read from a file line by line and joined using a hash table.

3.3. Semi-Join

Sometimes a large portion of the data set does not take part in the join. Deleting of tuples that will not be used in join significantly reduces the amount of data transferred over the network and the size of the dataset for the join. This preprocessing can be carried out using semi-joins by selection or by a bitwise filter. However, these filtering techniques introduce some cost (an additional MR job), so the semi-join can improve the performance of the system only if the join key has low selectivity. There are three ways to implement the semi-join operation:

- a semi-join using bloom-filter,
- semi-join using selection,
- an adaptive semi-join.

Bloom-filter is a bit array that defines a membership of element in the set. False positive answers are possible, but there are no false-negative responses in the solution of the containment problem. The accuracy of the containment problem solution depends on the size of the bitmap and on the number of elements in the set. These parameters are set by the user. It is known that for a bitmap of fixed size *m* and for the data set of n tuples, the optimal number of hash functions is k=0.6931*m/n. In the context of MapReduce, the semi-join is performed in two jobs. The first job consists of the Map phase, in which keys from one set are selected and added to the Bloom-filter. The Reduce phase combines several Bloom-filters from first phase into one. The second job consists only of the Map phase, which filters the second data set with a Bloom-filter constructed in previous job. The accuracy of this approach can be improved by increasing the size of the bitmap. However in this case, a larger bitmap consumes more amounts of memory. The advantage of this method is its the compactness. The performance of the semi-join using Bloom-filter highly depends on the balance between the Bloom-filter size, which increases the time needed for its reconstruction of the filter in the second job, and the number of false positive responses in the containment solution. The large size of the data set can seriously degrade the performance of the join.

```
Job 1: construct Bloom filter
    Map (K:null, V from L)
          Add Key to BloomFilter Bl
    close()  //for Map phase
          emit(null, Bl);

   Reduce (K': key, LV) //only 1 Reducer
     for l in LV do
          union filters by operation Or
     close()  // for Reduce phase
          write resulting filter into file;

Job 2: filter dataset
    init()   //for Map phase
          read filter from file in Bl
    Map (K:null, V from R)
          if (Key in Bl)  then
             emit (null, V);

Job 3: do join with L dataset and filtered dataset from
Job 2.
```

Listing 10: Semi-join using Bloom-filter

```
Job 1: find unique keys
   Map (K:null, V from L)
        Create HashMap H;
        if (not Key in H) then
             add Key to H;
             emit (Key, null);

   Reduce (K': key, LV) //only one Reducer
       emit (null,key);

Job 2: filter dataset
   init()  //for Map phase
         add to  HashMap H unique keys from job 1;
   Map (K:null, V from R)
         if (Key in H) then
            emit (null,V);

Job 3: do join with L dataset and filtered dataset from
Job 2.
```

Listing 11: Semi-join with selection.

Semi-join with selection extracts unique keys and constructs a hash table. The second set is filtered by the hash table constructed in the previous step. In the

context of MapReduce, the semi-join is performed in two jobs. Unique keys are selected during the Map phase of the first job and then they are combined into one file during the Map phase. The second job consists of only the Map phase, which filters out the second set. The semi-join using selection has some limitations. Hash table in memory, based on records of unique keys, can be very large, and depends on the key size and the number of different keys.

The Adaptive semijoin is performed in one job, but filters the original data on the flight during the join. Similar to the Reduce-side join at the Map phase the keys from two data sets are read and values are set equal to tags which identify the source of the keys. At the Reduce phase keys with different tags are selected. The disadvantage of this approach is that additional information about the source of data is transmitted over the network.

Job 1: find keys which are present in two datasets

```
Map (K:null, V from R or L)
      Tag = bit from name of R or L;
       emit (Key,Tag);

      Reduce (K': join key,
       LV: list of V with key K')
         Val = first value from LV;
            for t in LV do
              if (not Val==Val2) then
                 emit (null, K');
```

Job 2: before joining it is necessary to filter the smaller dataset dataset by keys from the Job 1 that will be loaded into hash map. Then the bigger dataset is joined with filtered one

Listing 12: Adaptive semi-join.

3.4. Range Partitioners

All algorithms, except the In-Memory join and their optimizations are sensitive to the data skew. This section describes two techniques of the default hash partitioner replacement.

A Simple Range-based Partitioner [4] (this kind similar to the *Skew join* in [14]) applies a range vector of dimension *n* constructed from the join keys before starting a MR job. By this vector join keys will be splitted into *n* parts, where *n* is the number of Reduce jobs. Ideally partitioner vector is constructed from the whole original set of keys, in practice a certain number of keys is chosen randomly from the data set. It is known that the optimal number of keys for the vector construction is equal to the square root of the total number of tuples. With a heavy data skew into a single key value, some elements of the vector may be identical. If the key belongs to multiple nodes, a node is selected randomly in the case of data on which to build

a hash table, otherwise the key is sent to all nodes (to save memory as a hash table is contained in the memory).

Virtual Processor Partitioner [4] is an improvement of the previous algorithm based on increasing the number of partition. The number of parts is specified multiple of the tasks number. The approach tends to load the nodes with the same keys uniformly (compared with the previous version). The same keys are scattered on more nodes than in the previous case.

```
//before the MR job starts
// optimal max = sqrt(|R|+|L|)
getSamples (Red:the number of reducers,
max: the max
                number of samples)
    C = max/Splits.length;
    Create buffer B;
   for s in Splits of R and L do

        get C keys from s;
        add it to B
    sort B;
   //in case simple range partitioner P == 1
   //in case virtual  range partitioner P > 1
   for j<(Red*P) do
        T = B.length/(Red*P)*(j+1);
        write into file B[T];
```

```
        Map(K:null, V from L or R)
Tag = bit from name of R or L;
 read file with samples and add samples
 to Buffer B;
 //in case virtual partition it is needed to
 // each index mod |Reducers|
 Ind = {i:  B[i-1] < Key <= B[i]}
 // Ind may be array of indexes in skew
  case
 if (Ind.length >1) then
     if (V in L) then
node = random(Ind);
emit (pair(Key, node), pair(V, Tag));
     else
for i in Ind do
   emit (pair(Key, i), pair(V, Tag));
  else  emit (pair(Key, Ind), pair(V,
  Tag));
        Partitioner (K:key, V:value, P:
        the number of reducers)
  return K.Ind;
         Reducer (K': join key, LV: list of
         V' with key K')
  The same as GRSJ
```

Listing 13: The range partitioners.

3.5. Distributed cache

The advantage of using distributed cache is that data set are copied only once at the node. It is especially effective if several tasks at one node need the same file. In contrast the access to the global file system needs more communication between the nodes. Better performance of the joins without the cache can be achieved by increasing number of the files replication, so there's a good chance to access the file version locally.

3.6. Comparative analysis of algorithms

The features of join algorithms are presented in the Table 1. The approaches with pre-processing is good when data is prepared in advance for example come from another MapReduce job. Algorithms with one phase and without tagging is more preferable due to the fact that no additional transferring data through the network are needed. Approaches that sensitive to the data skew may be improved by optimizations with range partitioner. In case of data low selectivity semi-join algorithms can improve performance and reduce the possibility of memory overflow.

	Pre-proces-sing	The number of phases	Tags	Sensi-tive to data skew	Need distr. cache	Memory overflow	Join algorithm
GRSJ	-	2	To value	yes	-	Number tuples for the same key is large	Nested loop
ORSJ	-	2	To key and value	yes	-	Number tuples for the same key is big	Nested loop
HYB	1 data	2	-	yes	-	Part size is large	Hash
MSPJ	2 data	1	-	yes	-	Part size is large	Hash
MSPMJ	2 data + sort	1	-	yes	-	-	Sort-merge
IMMJ	-	1	-	-	yes	Size of smaller dataset is large	Hash
MUL	1 data	1*part	-	-	yes	-	IMMJ

JDBM	-	1	-	-	yes	-	JDBM hash table
REV	-	1	-	-	yes	Part size is big and number of tuples for the same key is big	Hash

Table 1: Comparative analysis of algorithms.

The multiphase and JDBM map join algorithms is excluded from our experiments because of their poor performance.

4. Experiments

4.1. Dataset

Data are the set of tuples, which attributes are separated by a comma. Tuple is split into a pair of a key and a value, where value is the remaining attributes. Generation of synthetic data was done as in [4]. Join keys are distributed randomly except experiment with the data skew.

4.2. Cluster configuration

Cluster consists of three virtual machines, where one of them is master and slave at the same time, the remaining two are the slaves. Host configuration consists of 1 processor, 512 mb of memory for nodes, 5 gb is the disk size. Hadoop 20.203.0 runs on Ubuntu 10.10.

4.3. The General Case

The base idea of this experiment is to compare executions time of different phases of various algorithms. Some parameters are fixed: the number of Map and Reduce tasks is 3, the input size is 10^4x10^5 and 10^6x10^6 tuples.

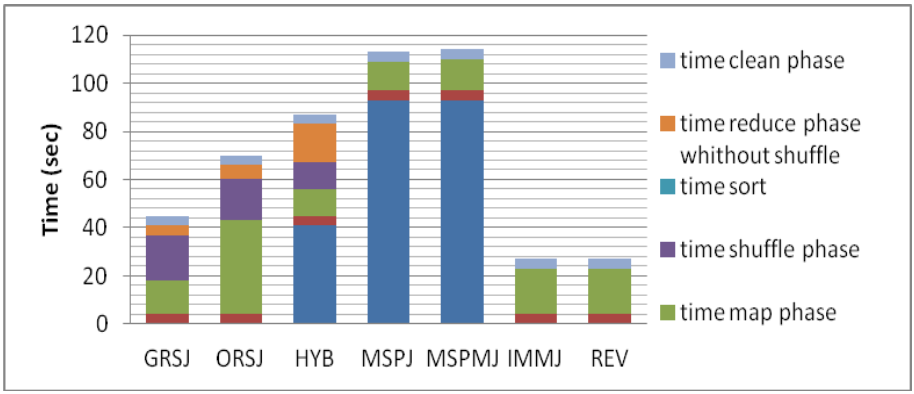


*Figure 1: Executions time of different phases of various algorithms. Size 10^4*10^5.*

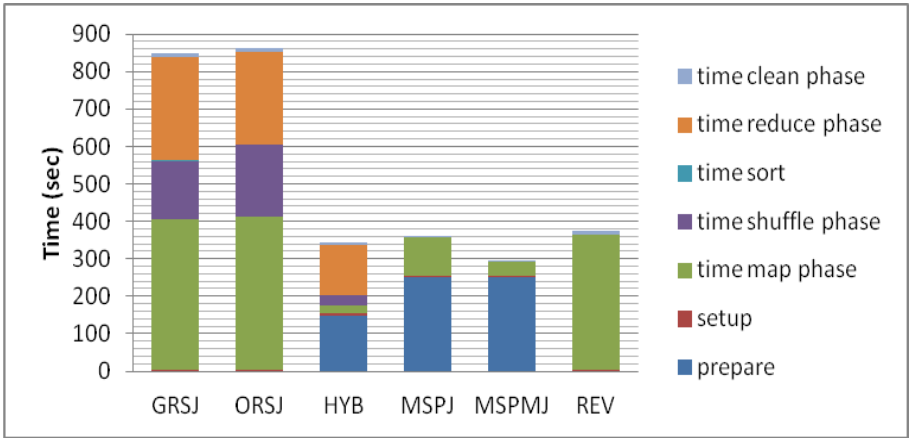


*Figure 2: Executions time of different phases of various algorithms. Size 10^6*10^6.*

For a small amount of data, Map phase, in which all tuples are tagged, and Shuffle phase, in which data are transferred from one phase to another, are more costly in Reduce-Side joins. It should be noted that GRSJ is better than ORSJ on small data, but it is the same on big data. It is because in first case time does not spend on combining tuples. Possible, on the larger data ORSJ outperform GRSJ when the usefulness of grouping by key will be more significant. Also for algorithms with pre-processing more time are spent on partitioning data. The algorithms in memory (IMMJ and REV) are similar in small data. Two algorithms are not shown in the

graph because of their bad times: JDBM-based map join and Multi-phase map join. In large data IMMJ algorithm could not be executed because of memory overflow.

4.4. Semi-Join

The main idea of this experiment is to compare different semi-join algorithms. These parameters are fixed: the number of Map and Reduce tasks is 3, the bitmap size of Bloom-filter is $25*10^5$, the number of hash-functions in Bloom-filter is 173, built-in Jenkins hash algorithm is used in Bloom-filter. Adaptive semi-join (ASGRSJ) does not finish because of memory overflow. The abbreviation of Bloom-filter semi-join for GRSJ is BGRSJ. The abbreviation of semi-join with selection for GRSJ is SGRSJ respectively.

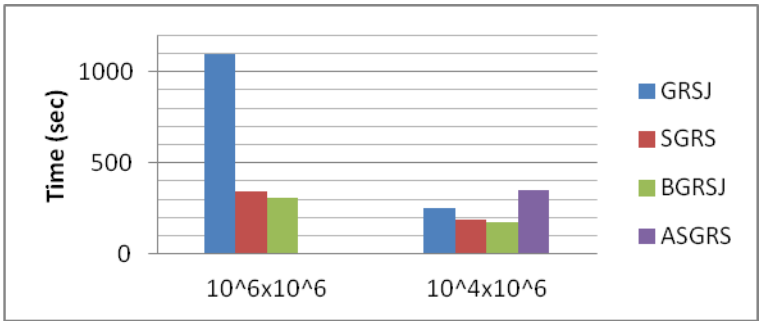


Figure 3: Comparison of different semi-join implementations.

4.5. Speculative execution

Speculative execution reduces negative effects of non-uniform performance of physical nodes. In this experiment two join algorithms GRSJ and IMMJ is chosen because of different numbers of phases and one of them sensitive to the data skew. Two dataset are considered: normal data that consists of 10^5x10^5 tuples and skew data that contain for one data $5*10^4$ same key in tuples and for second data 10 same keys in tuples. In case of IMMJ, which is not sensitive to the data skew, the performance with speculative execution is the similar approach without it. In case of GRSJ algorithm with uniform data approach without speculative execution is better than with it. But GRSJ algorithm with skew data and speculative execution outperforms four times approach without it.

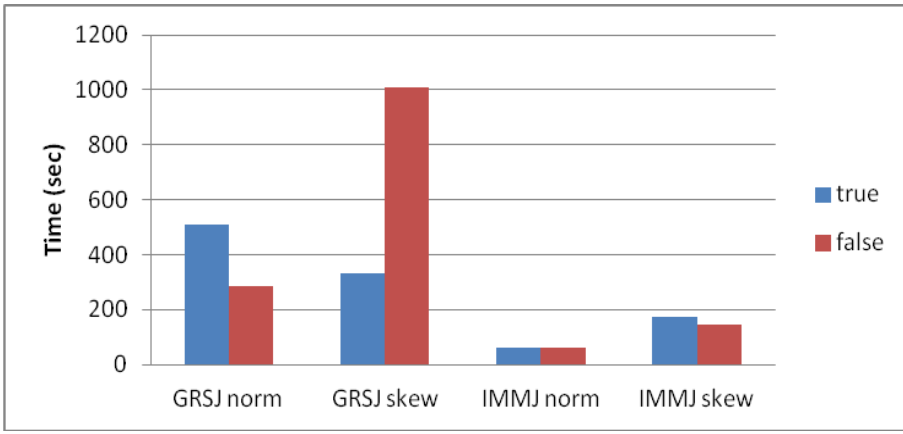


Figure 4: The effect of speculative execution.

4.6. Distributed cache

In [21] was showed that using of distributed cache is not always good strategy. They suggested that the problem can be a high speed network. This experiment was carried out for Reversed Map-Side join, because for which a distributed cache can be important. Replication was varied as 1, 2, 3 and size of data is fixed – 10^6x10^6 tuples. When data is small, the difference is not always visible. In large data algorithms with distributed cache outperform approach of reading from a globally distributed system.

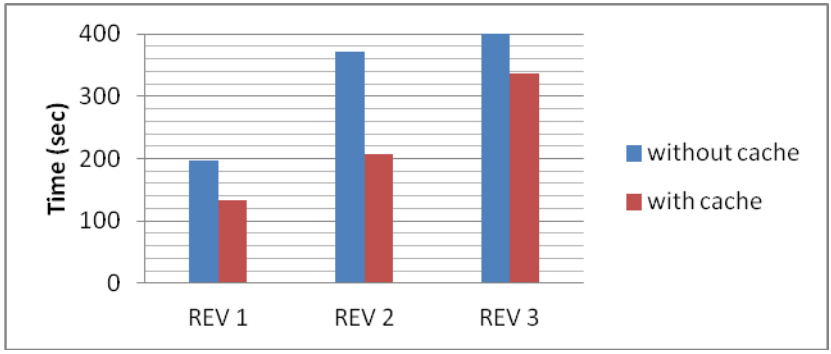


Figure 5: Performance of Reversed Map-Side join with and without using distributed cache.

4.7. Skew data

It is known that many of the presented algorithms are sensitive to the data skew. In this experiment take part such algorithms as Reduce-side join with Simple Range-based Partitioner for GRSJ (GRSJRange) and Virtual Processor Partitionerfor GRSJ (GRSJVirtual), and also for comparing in memory join: IMMJ, REV because of resistant to the skew. Fixed parameters are used: size of two dataset is $2*10^6$, one of the data set has the same key in $5*10^5$ tuples, and another has the same keys in 10 or 1 tuples. In case with IMMJ was memory overflow.

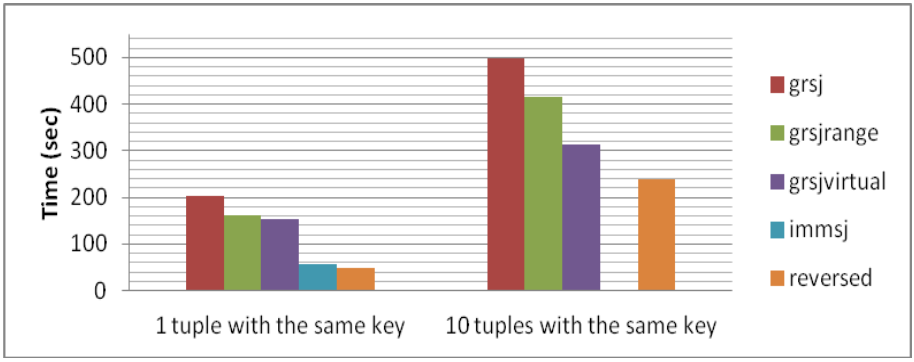


Figure 6: Processing the data skew.

Although these experiments do not completely cover the tunable set of Hadoop parameters, they are shown the advantages and disadvantages of the proposed algorithms. The main problems of these algorithms are time spent on pre-processing, transferring data, the data skew, and memory overflow.

Each of the optimization techniques introduces additional cost to the implementation of the join, so the algorithm based on the tunable settings and specific data should be carefully chosen. Also important are the parameters of the network bandwidth when distributed cache are used or not used and a hardware specification of nodes because of it is importance when speculative executions are on. Speculative execution reduces negative effects of non-uniform performance of physical nodes.

Based on the collected statistics such as data size, how many keys will be taking part in the join, these statistics may be collected as well as the construction of a range partitioner, the query planner can choose an efficient variant of the join. For example, in [5] was proposed what-if analyses and cost-based optimization.

5. Future work

The algorithms discussed in this paper, only two sets are joined. It is interesting to extend from binary operation to multi argument joins. Among the proposed algorithms, there is no effective universal solution. Therefore, it is necessary to evaluate the proposed cost models for join algorithms. And for this problem it is need to use real cluster with more than three nodes in it and more powerful to process bigger data, due to the fact that the execution time on the virtual machine may be different from the real cluster in reading/writing, transferring data over the network and so on.

Also the idea of processing the data skew in MapReduce applications from [19] can be applied to the join algorithms. Another direction to future work is to extend algorithm to support a theta-join and outer join.

An interesting area for future work is to develop, implement and evaluate algorithms or extended algebraic operations suitable for complex similarity queries in an open distributed heterogeneous environment. The reasons to evaluate complex structured queries are: a need to combine search criteria for different types of information; a query refinement e.g. based on user profile or feedback; advanced users may need query structuring. The execution model and algebraic operation to be implemented are outlined in [31]. The main goal is to solve the problems presented in [8] as a problem.

In addition, one of the issues is efficient physical representation of data. Binary formats are known to outperform the text both in speed reading and partitioning key / value pairs, and the transmission of compressed data over the network. Along with the binary data format, column storage has already been proposed for paradigm MapReduce. It is interesting to find the best representation for specific data.

6. Conclusion

In this work we describe the state of the art in the area of massive parallel processing, presented our comparative study of these algorithms with optimizations such as semi-join and range partiotioner. Also our directions of future work is discussed.

References

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. ACM.

[2] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. Proc. VLDB Endow., 2:922–933, August 2009.

[3] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In Proceedings of the 13th International Conference on Extending Database Technology, EDBT '10, pages 99–110, New York, NY, USA, 2010. ACM.

[4] Fariha Atta. Implementation and analysis of join algorithms to handle skew for the hadoop mapreduce framework. Master's thesis, MSc Informatics, School of Informatics, University of Edinburgh, 2010.
[5] Shivnath Babu. Towards automatic optimization of mapreduce programs. In Proceedings of the 1st ACM symposium on Cloud computing, SoCC '10, pages 137–142, New York, NY, USA, 2010. ACM.
[6] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In Proceedings of the 2010 international conference on Management of data, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.
[7] A Chatzistergiou. Designing a parallel query engine over map/reduce. Master's thesis, MSc Informatics, School of Informatics, University of Edinburgh, 2010.
[8] Surajit Chaudhuri, Raghu Ramakrishnan, and Gerhard Weikum. Integrating db and ir technologies: What is the sound of one hand clapping? In CIDR, pages 1–12, 2005.
[9] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. Mad skills: new analysis practices for big data. Proc. VLDB Endow., 2:1481–1492, August 2009.
[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. Commun. ACM, 53:72–77, January 2010.
[11] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. Mapreduce: simplified data processing on large clusters. In In OSDI04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation. USENIX Association, 2004.
[12] Leonidas Fegaras, Chengkai Li, and Upa Gupta. An optimization framework for map-reduce queries. In EDBT 2012, march 2012.
[13] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. Column-oriented storage techniques for mapreduce. Proc. VLDB Endow., 4:419–429, April 2011.
[14] Alan F Gates. Programming Pig. O'Reilly Media, 2011.
[15] Herodotos Herodotou. Hadoop performance models. CoRR, abs/1106.0940, 2011.
[16] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. PVLDB, 4(11):1111– 1122, 2011.
[17] Eaman Jahani, Michael J. Cafarella, and Christopher R´e. Automatic optimization for mapreduce programs. Proc. VLDB Endow., 4:385–396, mar 2011.
[18] Dawei Jiang, Anthony K. H. Tung, and Gang Chen. Map-join-reduce: Toward scalable and efficient data analysis on large clusters. IEEE Transactions on Knowledge and Data Engineering, 23:1299– 1311, 2011.
[19] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. A study of skew in mapreduce applications. Moskow, Russia, june 2011. In the 5th Open Cirrus Summit.
[20] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In Proceedings of the 2011 international conference on Management of data, SIGMOD '11, pages 961–972, New York, NY, USA, 2011. ACM.
[21] Gang Luo and Liang Dong. Adaptive join plan generation in hadoop. Technical report, Duke University, 2010.
[22] Christine Morin and Gilles Muller, editors. European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010. ACM, 2010.

[23] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In Proceedings of the 2011 international conference on Management of data, SIGMOD '11, pages 949–960, New York, NY, USA, 2011. ACM.
[24] Konstantina Palla. A comparative analysis of join algorithms using the hadoop map/reduce framework. Master's thesis, MSc Informatics, School of Informatics, University of Edinburgh, 2009.
[25] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In Proceedings of the 35th SIGMOD international conference on Management of data, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.
[26] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. SIGMOD Rec., 18:110–121, June 1989.
[27] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In Proceedings of the 2010 international conference on Management of data, SIGMOD '10, pages 495–506, New York, NY, USA, 2010. ACM.
[28] Vertica Systems, Inc. Managing Big Data with Hadoop & Vertica, 2009.
[29] Guanying Wang, Ali Raza Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In MASCOTS, pages 1–11. IEEE, 2009.
[30] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.
[31] Anna Yarygina, Boris Novikov, and Natalia Vassilieva. Processing complex similarity queries: A systematic approach. In Maria Bielikova, Johann Eder, and A Min Tjoa, editors, ABDIS 2011 Research Communications: Proceedings II of the 5th East-European Conference on Advances in Databases and Information Systems 20 – 23 September 2011, Vienna, pages 212–221. Austrian Computer Society, September 2011.
[32] Minqi Zhou, Rong Zhang, Dadan Zeng, Weining Qian, and Aoying Zhou. Join optimization in the mapreduce environment for column-wise data store. In Proceedings of the 2010 Sixth International Conference on Semantics, Knowledge and Grids, SKG '10, pages 97–104, Washington, DC.

Periodic event sets detection in temporal databases

© Ekaterina Ivannikova
Saint Petersburg State University
ivannikovae@gmail.com

Abstract. Temporal data regularity is an important data property that can be used in different applications. Such regularity is explored in the field of periodic pattern mining. In this paper, we raise a problem of periodic sets detection and suggest the method for its solution. The existing algorithms for the periodic event mining are considered in detail and a new approach is proposed in the paper. The comparison of the algorithms and their performance are demonstrated through a series of experiments.

Keywords: periodic set; pattern mining; temporal mining; data regularity

1. Introduction

Pattern mining is an important area of data mining, and it has been growing rapidly over the two past decades. The initial stimulus for the development of methods was the problem of market basket analysis, that requires to determine which products are usually purchased together by using a transaction database of supermarket buying. The new knowledge could be used for the correct placement of goods or for the advertising purpose. The first efficient algorithm for finding frequent patterns has been proposed in [1]. Now there are many algorithms for detecting the patterns that are applied in various fields. For example, they are used in biology to identify the sequences of nucleotides, in the analysis of log files for the detection of failures or attacks, in medicine for diagnosis, in marketing for advertising or tips for users, etc.

Many kinds of the data in the real world are time series or temporal databases. Periodic pattern mining is the problem that regards temporal regularity. Periodic patterns are characterized by a certain persistence and predictability. Information about such regularity can be used for many tasks: for the purpose of personal promotion to the users, for the timely reminders about the future events or forecasting.

By the periodic pattern we mean the set of items that frequently occur together at regular time intervals. There are two ways for prediction if the pattern and its period p are known. If the pattern has occurred during some time interval (t_{beg}, t_{end}) then it is likely to repeat during (t_{beg}+ p, t_{end} + p). And if several events of the pattern have

occurred then other events of the pattern are likely to happen soon. Therefore, it is important to identify and describe the periodicity.

There are several types of periodic patterns considered in literature. Most of the studies on their mining apply the Apriori property heuristic and adopt some variations of Apriori-like mining methods [9]. Apriori property is used to reduce the search space and formulated as "All nonempty subsets of a frequent itemset must also be frequent", i.e. an itemset is frequent only if all of its sub-itemsets are frequent. This observation applies to construct k+1-patterns based on the found k-patterns set. But 1-patterns are generally detected by a simple search and are not consider in detail. In addition, in most of the previous works the 1-pattern consists of a single item while the pattern that consists of a number of events may be interested in some tasks. In this work we present an algorithm for such periodic 1-patterns finding and discuss several approaches to periodic event detection. We propose a new algorithm to the periodic event mining as well. The experiments show that our method is the more advantageous in some cases.

The remainder of this paper is organized as follows. In section 2 the previous works on the frequent and periodic pattern mining will be discussed. In section 3 the notation used throughout the paper and the formal definition of periodic patterns are introduced. Section 4 describes the several possible ways of patterns detection and section 5 presents the experimental results. The conclusion and future research directions are contained in section 6.

2. Related works

Our study combines frequent pattern mining and periodic pattern mining in periodic sets finding. We will review the related works in these areas below.

2.1. Frequent pattern mining

There are a huge number of studies in this field in literature. The existing algorithms can be classified into three main groups:

- iterative Apriori-like level-wise mining techniques
- pattern mining methods without candidate generation
- mining techniques with the vertical data format

As mentioned above, the concept of frequent itemset and the first algorithm for finding frequent patterns by using downward closure property, called Apriori, were introduced by Agrawal and Srikant in [1]. This approach requires candidate set generation and scanning the database to check each candidate. Such techniques as hashing [14], partitioning [15], upper bound of the number of candidate patterns that can be generated in the level-wise approach [5] were developed for improvements and extensions of algorithms in that category [7].

In paper [10] a problem of finding long frequent patterns is considered and a new FP-growth approach that mines frequent itemsets without candidate generation was

devised. At the first step items are ordered in frequency-descending order and according to this list, the database is compressed into a special FP-tree and after that the tree is mined in some way. An alternative approach is proposed in [6].

These methods are used to discover patterns from a set of transactions in a horizontal data format (i.e. {Transaction_id :itemset}). An example of alternative class of the algorithms that first transforms the data into a vertical data format (i.e. {item :Transaction_id set}) is Eclat [18]. These methods don't require scanning the database to count the support of (k+1)-candidates.

2.2. Periodic pattern mining

The problem of mining periodic patterns can be viewed from different perspectives [9]: depending on the coverage of the pattern there are full- and partial-periodic patterns. Basing on the precision of the periodicity synchronous and asynchronous pattern can be identified. And a pattern can also be precise or approximate.

Generally, in the works related with our, the following notation of periodic pattern is used. Let S be the sequence $S = S_1, S_2, \ldots, S_n$. A pattern p is the nonempty sequence $p = p_1 \ldots p_k$, where $p_i \subset S_j \bigcup \{*\}$. The additional event {*} – symbol that matches any event and is used to represent the "don't care" position in the pattern. The frequency of a pattern p is the count of j such that the string s is true in $S_{i|p|+1} \ldots S_{i|p|+|p|}$. If the frequency of the p exceeds a minimum support threshold, then this pattern is named periodic. A pattern with the k non-{*} positions is also called an i-pattern.

Cyclic association rules that display regular cyclic variation over time were first introduced in [13]. These rules are based on partial-periodic patterns with perfect periodicity in the sense that each pattern reoccurs in every cycle. In the article several techniques called cycle-pruning, cycle-skipping and cycle-elimination were proposed to optimize mine periodic patterns with 100% support.

General partial-periodic patterns with imperfect periodicity studied in [3, 4, 8] are more common in real world. In work [3] a new structure named abbreviated list table (ALT) that maintains the occurrence counts of all symbols and corresponding periods was proposed. Using this structure only a small number of data sequence passes are required to compute the periods and the patterns of size 1. Han et al [8] explored interesting properties such as the Apriori property and the max-subpattern hit set property related to partial periodicity and proposed several methods for efficient mining of k-patterns. Another original algorithms for symbol and segment periodicity detection based on mapping scheme and convolution may be found in [4].

The above methods were used to discover potential periods from the entire time-series data. In paper [16] the authors presented the dense periodic patterns that may exist only in a limited range of the time-series.

Most of the works are concentrated on the k-patterns constructing and use a base line algorithm for 1-patterns mining. To the best of our knowledge only paper [3]

represents the way to improve 1-patterns mining. In our work we propose a new approach to the 1-patterns mining and perform a comparison analysis with the existing base line and ATL approaches. In addition, most of the existing studies focused on the detection of such patterns where S_i and p_j are single events while we explore the patterns where p_i may be a set of events.

Another group of works [11, 12, 17] is focused on the regular activities that occur with a calendar-based periodicity. In [12] the calendar schema is proposed to construct periodic patterns. Calendar-based approach allows to specify multiple time granularities. For example, schema (2010, *, 1) means that the pattern occurred on the first day of each month in 2010. A fuzzy periodic calendar and an algorithm for mining fuzzy periodicity are developed in [11].

3. Problem definition

Let E be a set of events. Time-series S is a sequence of records (t_i, e_i) with an event $e_i \in E$ and a time stamp t_i when the event occurred.

We assume that the pattern time interval is limited by user-specified window W, i.e. all events in pattern occur within the interval W. Time-series S can be presented as $S = S_1 S_2 \dots S_n$, where S_i are sequential disjoint sets of the events happening during the window. For example, for W equal to an hour, S_1 may contain the events which occurred from 12 am to 13 am, S_2 contains the events that took place from 13 to 14 am, and so on.

Let T be a subset of E. The set T is contained in S_i, if all the events of T are in S_i. The ***binary sequence*** for itemset T ($BinSeq_T$) is constructed as follow: $BinSeq_T[i] = 1$ if T is contained in S_i, and $BinSeq_T[i] = 0$ otherwise. The ***candidate pair*** (p, o) with period p and offset o has a ***support*** Sup of the period (p, o) for the set T:

$$Sup(p,o) = \frac{|i : BinSeq[i*p+o]=1|}{|BinSeq[i*p+i]|} * 100\%$$

$$Sup(p,o) = \frac{|i : BinSeq_T[i*p+o]=1|}{|BinSeq_T[i*p+o]|} * 100\% \text{ , where}$$

$$|BinSeq_T[i*p+o]| = \frac{Size(BinSeq_T)}{p} \qquad , i*p+o \le Size(BinSeq_T)$$

The notation (p, o, sup) will be used if it is known that the candidate pair (p, o) has the support sup. We say that the set T is a ***periodic 1-pattern*** with period p and offset o if Sup(p, o) is not less than a specified minimum support. The length of a 1-pattern is the number of elements in the pattern. The 1-pattern of length 1 we will also be called a ***periodic event*** and the 1-pattern of length k greater than 1 we will be referred to as a ***periodic k-set*** for short. Our goal is to find all possible periodic sets in a given time-series.

4. Algorithm

Our algorithm belongs to a group of methods using an iterative approach known as a level-wise search. This approach uses k-patterns to generate (k+1)-patterns. At the first step of the algorithm the set of periodic events is found by searching the binary sequences. Further, the periodic sets with two items can be constructed and tested for the periodicity (i.e. which of them satisfy a minimum support), and so on, until no more periodic k-sets can be detected.

The following predefined parameters are used in the algorithms: minimum period (P_min), maximum period (P_max), minimum support (Sup_min).

4.1. Periodic items

Three methods for the periodic events detection will be discussed below. The first method is the base line algorithm that is used in most of existing works. In the second approach we adopt the idea of ATL structure described in [3]. The third approach represents a new way to periodic item detection. The periodic 1-sets or periodic events are found using binary sequences of events. So, for each event it is needed to construct the binary sequence as described in section 3.

Base line approach.

This method requires counting the support value for all possible periods-candidates as shown in fig. 1.

```
forP_min ≤ p ≤P_max
for 0 ≤ o < p
       if (Sup(p, o) ≥ Sup_min) then
       Result.AddPeriod(p, o)
```

Fig. 1

Such an approach requires the full scan of the binary sequence for the testing of some period. To check all the candidates *(P_max – P_min + 1)* passes are needed for each event. If n is the length of the binary sequence, $\lfloor n/p \rfloor$ steps are required to check one candidate (p, o). There are p candidate pairs with period p and different offsets. To test them $p * \lfloor n/p \rfloor \sim n$ steps are required. And the number of interesting periods is *(P_max-P_min+1)*. So, the algorithm is performed in time $O((P\_max-P\_min+1) * n)$.

ATL-based approach.

In this case only one full scan of the sequence is required. The number of times each candidate pair occurs is counted by scanning.

Initially, count is equal to 0 for all the candidates. When a position i is considered, if BinSeq[i] = 1 then the count of the occurrences increases by one for all the cycles (p, o = i mod p).

For example, consider a binary sequence: BinSeq = 01001011001, P_min = 2, P_max = 5.

Let the algorithm scan sequence at position i=4: BinSeq [4]=1 => (2, 0).count++, (3, 1).count++, (4, 0).count++, (5, 4).count++, and so on while i less than the sequence length.

After the occurrences numbers of candidate periods have been counted the supports of pairs are calculated as follow:

$$Sup(p, o) = (p, o).counts / \frac{Size(BinSeq)}{p} * 100\%$$

This method is described below in Fig 2.

```
Result; // the set of finding periods
for  0 ≤ I < Size(BinSeq)
if(BinSeq[i]=1) then
       forP_min ≤ p ≤ P_max
        (p, i mod p).count++;
forP_min ≤ p ≤P_max
for 0 ≤ I < p
       if(sup(p, o) ≥ Sup_min) then
       Result.Add(p, o);
```

Fig. 2

Time complexity of this approach is *O(n).*

Divider-based pruning approach.

This method is based on the following observation: if there is a triple (p, o, sup), then a triple (p_1, d_1, sup_1) = (p/d, o, sup/d) with smaller period and support also exists. To be more precise d is a divider of p and sup_1 no less than sup/d. So, in order for a triple (p, o, sup) to exist, the existence of triple (p_1 = p/d, o, $sup_1 \geq$ sup/d) calculated at the previous steps is necessary. Support values that were counted for smaller periods can be used to reduce computing at the next iterations for larger periods mining. If for some candidate (p, o) the support is equal to *sup* and *sup* < *Sup_min*, then the candidate (p*i, o) is not suitable in case *p*i* < *P_max* and *sup*i* < *Sup_min.*

For example, at some iteration of the algorithm a candidate pair (4, 3) is tested on periodicity with the minimum support 80 %. If the support of the candidate is 30% then candidate (8, 3) may be excluded from further consideration.

The second observation we will use is that if the period (p, o) is found, then the multiple periods (p*i, o) are not so interesting.

At the beginning of the algorithm we assume that there are all possible periods, i.e. pairs (p, o). Denote this set as Cand. Pseudocode of the algorithm is given in Figure 3.

```
Result; // the set of finding periods
while (not Cand.Empty())
  (p, o) = Cand.GiveNextPair();
sup = Sup(p, o);
if (sup ≥ Sup_min) then
Result.Add(p, o);
i = 1;
    while (p*i ≤ P_max)
    Cand.Remove(p*i, o);
else
       i = 1;
       while (p*i ≤ P_max and
sup*I < Sup_min)
       Cand.Remove(p*i, o);
       i++;
```

Fig. 3

Algorithm evaluation:

1) In the algorithm a check of all candidates with the prime periods is required. Such pairs couldn't be removed from the set of candidates at the previous steps of the algorithm.

The number of primes less than N is estimated approximately as *N/ln(N)*. Consequently, the number of prime periods in the range of P_min to P_max equal to $K = P\_max/ln(P\_max) - P\_min/ln(P\_min)$. To compute the support of pairs with period p and all possible offsets the scan of the entire binary sequence is required. Thus, to test all prime periods, we need to perform $K*n$ steps.

2)Let's consider what candidates with the composite periods should be treated on the average. There are $(P\_min + P\_max)/2 * (P\_max - P\_min + 1)$ pairs with period p from P_min to P_max and the corresponding offset from 0 to p-1.

The number of prime periods is equal to K. The mean period is *(P_min + P_max)/2* and the number of candidates with this period and different offsets is *(P_min + P_max)/2* also. We estimate the count of candidates with prime periods as *K*(P_min + P_max)/2*. Therefore, the number of candidates with the composite periods may be computed like that:

$$\frac{(P_{\min} + P_{max})}{2} * (P_{max} - P_{min} + 1) - K * \frac{(P_{\min} + P_{max})}{2} =$$

$$\frac{(P_{\min} + P_{max})}{2} * (P_{max} - P_{min} + 1 - K)$$

We assume that a half of these pairs on the average are excluded from consideration, i.e. from the candidate set, at the previous steps of the algorithm. To check a candidate with period p it is required to scan n/p elements of the binary sequence. Consequently, to check a candidate with the mean period it is required *n / (P_min+P_max) / 2 = 2*n / (P_min+P_max)* elements. So, to check the candidates with composite periods it is needed to take

$$\frac{1}{2} * \frac{(P\_\min + P\_max)}{2} * (P_{max} - P_{min} + 1 - K) * \frac{2 * n}{(P_{min} + P_{max})} =$$

$$= *(P\_max - P\_min + 1 - K)*n \text{steps on the average.}$$

3) So, summing the results in 1) and 2) for evaluating prime and composite periods respectively we conclude that the algorithm perform
*K*n + *(P_max – P_min + 1 - K)*n = *(P_max – P_min + 1+K)*n* steps. And the time complexity of this approach is *O(*(P_max – P_min + 1+K)*n)* in the mean.

4.2. Periodic k-sets

The Apriori property can be adapted to periodic sets: All nonempty subsets of a periodic set must also be periodic sets with the same periods and offsets. So, for k+1-sets generation we will use the k-sets.

Let $P_k(p, o)$ be the collection of k-sets having period p with offset o. This set contains the patterns with their binary sequences. The order of elements in the patterns is not significant, and we will keep items in lexicographic order. In this case we apply the join step proposed by Agrawal, etc. in [1] to the candidate k+1-

sets (denote this set as $C_{k+1}(p, o)$) generation. P_k is joined with P_k in the following way:

$$\text{Let } p = p_1 p_2 \ldots p_k,\ q = q_1 q_2 \ldots q_k \in P_k$$

$$\text{If p, q such that } p_1 = q_1, \ldots, p_{k-1} = q_{k-1}, p_k < q_k, \text{ then}$$

$$c = p_1 p_2 \ldots p_k q_k \in C_{k+1}$$

We will store k-sets with their binary sequences. $BinSeq_p$ is the binary sequence for k-pattern P and $BinSeq_{qk}$ is the binary sequence of item q_k. Support of the candidate set $c = p_1 p_2 \ldots p_k q_k$ is calculated as:

$$Sup_c(p, o) = \frac{|\, i : BinSeq_p[i * p + o] = 1 \,\&\, BinSeq_{qk}[i * p + o] = 1 \,|}{|\, BinSeq_p[i * p + o] \,|} * 100\%$$

$$|\, BinSeq_p[i * p + o] \,| = \frac{Size(BinSeq_p)}{p}$$

If the support of the candidate c is more than the given minimum support, then $c \in P_{k+1}(p, o)$. All k+1-candidates are tested and the set of k+1-sets is composed. Similarly, the collection of k+2-sets is obtained from the set of k+1-patterns and so on.

Let's estimate the time required to generate $P_{k+1}(p, o)$ from the set $P_k(p, o)$. Let m be the number of the patterns in $P_k(p, o)$. The count of the candidate k+1-sets deriving at the join step can be evaluated as $(m-1) + (m-2) + \ldots + 1 = m(m-1)/2$ in the worst case. If n is the binary sequence length, then $2*n/p$ steps are required to check the one candidate. So, the algorithm perform $m(m-1)*n/p$ steps during the construction of the $P_{k+1}(p, o)$ set from $P_k(p, o)$.

So, we have shown the method for periodic k-sets mining above. Note that the periodic k-patterns can be obtained from the found periodic sets using known methods for periodic pattern mining.

5. Experiments

5.1. Data generation

For our experiments we used synthetic data. The time-series were generated by tuning the following parameters: the beginning date and the end date of the sequence, the number of different events (|E|), the length of time-series (i.e. total number of entries in a file), the count of periodic sets in series, the minimum and maximum periods, the minimum and maximum window for periodic sets, the minimum and maximum support of periodic sets.

Note that in addition to the known generated periodic sets the time-series may contain a number of others patterns. These periods are formed by noise events, which correspond to the real data. Some of the periods may be obvious, well-known or uninteresting, but the task of the revelation of interesting and useful periodic patterns is not in the scope of this work.

The events in the data have different frequency. |E| is the number of different (noisy) events. We have an ordered list of the events. At some moment of the time an event with a sequential number N occurs. The number N is calculated as *N = random.Next(random.Next(|E|))*.

So, the smaller the event ordinal number, the more frequently it happens. The intervals between successive events in the generated data are the same.

5.2. Experiment performance

The algorithms described for periodic events detection are denoted as BL (Base line Approach), ATL (ATL-based approach) and DBP (Divider-based pruning approach).

Fig. 1 illustrates that the behavior of the algorithms depends on the time-series frequency. Time-series frequency (F_S) is a value that indicates how many events occur in a time unit or a patterns window. The other parameters of the data: data for the time span in a month, 1200 different events, the window size for patterns from 10 minutes to an hour, the periods from 3 hours to a week, the support of the generated patterns ranges from 60 to 100%. The algorithm works with P_min= 3 hour, P_max= 1 week, Sup_min= 70%.

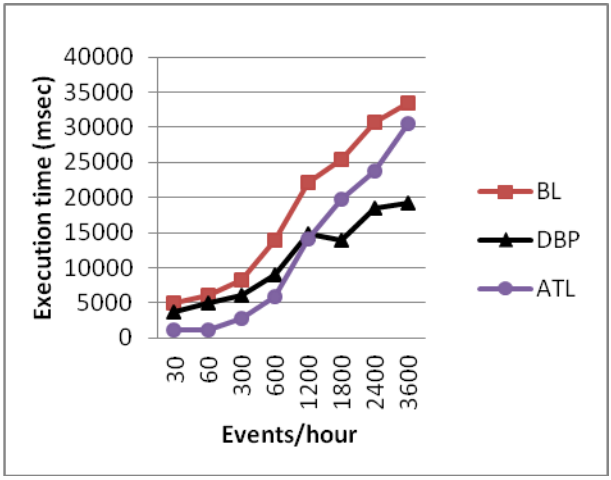


Fig. 1

Figure 1 shows a significant efficiency gain by DBP and ATL over base line approach BL. These algorithms allow to improve the execution time by 1,6-5,4 times. ATL is more efficient for a smaller number of entries an hour while it is opposite with DBP. We assume that the algorithms performance really depends on the ratio of different events number and time-series frequency, and not only on the frequency of series. It is confirmed by the second experiment (Fig. 2), where the size of events set E is changed for a fixed frequency (20 events per minute).

For our data if the rate $|E|/F_S < 0,7$ then the algorithm ATL is more efficient, and it is otherwise if $|E|/F_S > 0,7$ then DBP.

Fig. 3 gives execution time against range of periods. The minimum period is set at three hour and the maximum period value varies from 24 to 168 hours. All algorithms are ecexuted longer with increasing the range. But DBP execution time grows slower by 1,3-1,5 times as compared to the other two.

The scalability of the algotithms with respect to the analyzed data size is displayed in Fig. 4. We compare the performance of each approach for data from 8 to 96 Mb, which corresponds to the data period from 1 to 12 months if the time-series has 5 events per minute on average and 900 different events. The graphs show that three algorithms have scalability close to linear. However, ATL execution time increase grows about 1,6 times more slowly than BL and 2 times more slowly than DBP for such data.

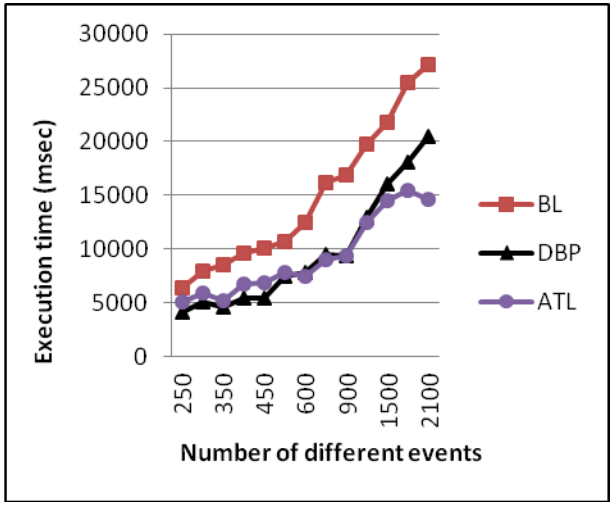


Fig. 2

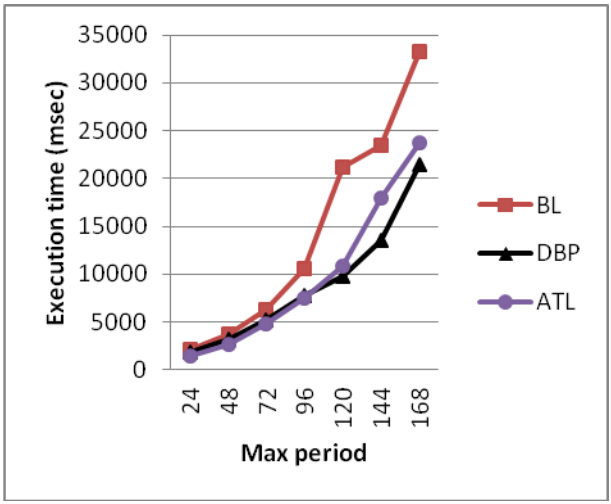


Fig. 3

Obviously, the generation of k+1-sets depends largely on the previous step, i.e. how many k-sets were found (the greater the number of k-sets, the more time is required for k+1-sets detection) while the generation time of the 1-sets relies on the input data size mostly. Therefore the stage of the periodic event extracting may take a significant part of periodic set mining algorithm and the improvement of this step can accelerate the algorithm performance as a whole. Fig. 5 shows the performance of the algorithm against the number of 1-sets. For step of 1-sets detection we use the BL algorithm.

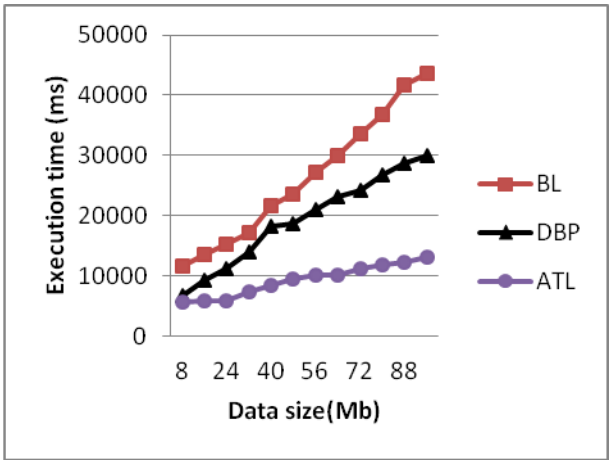


Fig. 4

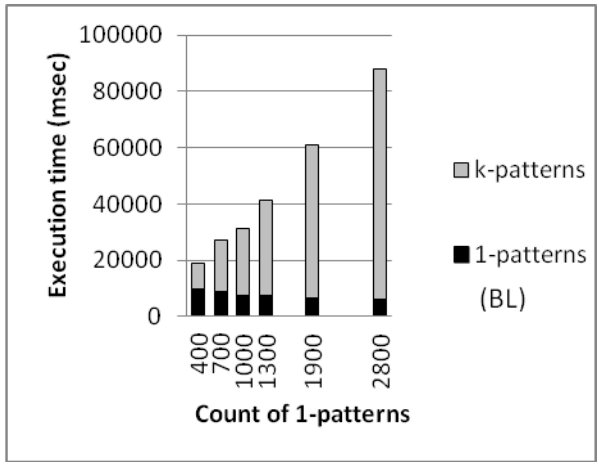


Fig. 5

Unlike the BL and ATL methods, using the introduced DBP approach in order to mine the periodic events allows to reduce the time of the k-patterns discovery as well. It is achieved due to the removal of multiple periods. For the same data in previous experiment the k-sets generation after the first step with DBP is 2-12% faster than after BL.

6. Conclusion

In this work the problem of periodic 1-patterns finding was considered. We represent the approach to periodic sets generation relying on the methods of frequent pattern mining. The new algotithm DPA for the periodic item mining was introduced as well as its evaluation was determined. We considered in detail the other existing approaches and compared them with the proposed one. The series of experiments shows that the proposed algorithms DPA can give up to a hundreds percent increase in performance of 1-patterns mining over the base line approach used in most of the previous studies. The our algorithm is the more advantageous then ATL in some cases also. The experimental comparison of the existing methods with different data and parameters is described in the paper.

In the future work it is interesting to analyze the memory management and explore the algorithms performance on the real and big data. Although the BL requires more time, it is no need to store the additional structures as DBP or ATL. Other directions for the future work are the solution of useful periodic patterns detection problem and developing the parallel extensions of the algorithms.

References

[1] RakeshAgrawal and RamakrishnanSrikant. Fast algorithms for mining association rules in large databases. In VLDB, pages 487-499, 1994.

[2] Juan M. Ale and Gustavo Rossi. Discovering association rules in temporal databases. In Encyclopedia of Database Technologies and Applications, pages 195-200. 2005.

[3] Huiping Cao, David W. Cheung, and Nikos Mamoulis. Discovering partial periodic patterns in discrete data sequences. In PAKDD, pages 653-658, 2004.

[4] Mohamed G. Elfeky, Walid G. Aref, and Ahmed K. Elmagarmid. Periodicity detection in time series databases. IEEE Trans. Knowl. Data Eng., 17(7):875-887, 2005.

[5] FlorisGeerts, Bart Goethals, and Jan Van den Bussche. A tight upper bound on the number of candidate patterns. In ICDM, pages 155-162, 2001.

[6] GöstaGrahne and Jianfei Zhu. E-ciently using prex-trees in mining frequent itemsets. In FIMI, 2003.

[7] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. Data Min. Knowl. Discov.,15(1):55-86, 2007.

[8] Jiawei Han, Guozhu Dong, and Yiwen Yin. E-cient mining of partial periodic patterns in time series database. In ICDE, pages 106-115, 1999.

[9] Jiawei Han and MichelineKamber. Data Mining: Concepts and Techniques, second edition. Morgan Kaufmann, 2000.

[10] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In SIGMOD Conference, pages 1-12, 2000.

[11] Wan-Jui Lee, Jung-Yi Jiang, and Shie-Jue Lee. Mining fuzzy periodic association rules. Data Knowl. Eng., 65(3):442_462, 2008.

[12] Yingjiu Li, PengNing, Xiaoyang Sean Wang, and SushilJajodia. Discovering calendar-based temporal association rules. In TIME, pages 111-118, 2001.

[13] BanuÖzden, Sridhar Ramaswamy, and Abraham Silberschatz. Cyclic association rules. In ICDE, pages 412-421, 1998.

[14] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. In Proceedings of the 1995 ACMSIGMOD international conference on Management of data, SIGMOD '95, pages 175-186, New York, NY, USA, 1995. ACM.

[15] Ashok Savasere, Edward Omiecinski, and Shamkant B. Navathe. An effcient algorithm for mining association rules in large databases. In VLDB, pages 432-444, 1995.

[16] Chang Sheng, Wynne Hsu, and Mong-Li Lee. Mining dense periodic patterns in time series data. In ICDE, page 115, 2006.

[17] Keshri Verma and Om Prakash Vyas. E-cient calendar based temporal association rule. SIGMOD Record, 34(3):63-70, 2005.

[18] Mohammed Javeed Zaki. Scalable algorithms for association mining. IEEETrans. Knowl. DataEng., 12(3):372-390, 2000.

Зависимости между ошибками на классах тестируемых реализаций

Игорь Бурдонов, Александр Косачев
<igor@ispras.ru>, kos@ispras.ru

Аннотация. Статья посвящена проблеме зависимости между ошибками, определяемыми спецификацией, и связанной с ней проблеме оптимизации тестов. Между ошибками имеется зависимость, если существует такое строгое подмножество ошибок, что любая неконформная реализация (то есть реализация, в которой есть какая-нибудь ошибка) содержит ошибку из этого подмножества. Соответственно, достаточно, чтобы тесты обнаруживали ошибки только из этого подмножества. Предлагается формальная модель тестового взаимодействия самого общего вида и конформность типа редукции, для которых зависимость между ошибками практически отсутствует. Показывается, что многие известные конформности в различных семантиках взаимодействия являются частными случаями этой общей модели. В этой общей модели зависимость между ошибками может возникать, когда в качестве класса тестируемых реализаций выбирается то или иное строгое подмножество класса всех реализаций. Частные семантики взаимодействия и/или различные гипотезы о реализации (в частности, гипотезы о безопасности) как раз и предполагают, что тестируемая реализация не любая, а относится к некоторому подклассу (безопасных) реализаций.

Ключевые слова: Семантика взаимодействия, трассы, LTS, конформность, зависимость между ошибками, генерация тестов, дивергенция, разрушение, безопасное тестирование.

1. Введение

Правильность исследуемой системы в самом широком смысле понимается как её соответствие заданным требованиям. Для верификации (проверки) этого соответствия с помощью формальных методов, объекты и отношения реального мира отображаются в модельные, математические объекты и отношения. Модель исследуемой системы называется реализацией, модель требований – спецификацией, а соответствие требованиям отображается в модельную конформность. Последняя понимается как обычное математическое соответствие, то есть подмножество декартового произведения множеств реализаций и спецификаций.

Спецификация и конформность считаются заданными. Что касается реализации как модели исследуемой системы, то *тестовая гипотеза*

предполагает, что такая модель существует для каждой исследуемой системы [[8]].

Если реализация, как модель исследуемой системы, известна, то возможна статическая (аналитическая) верификация, которая сводится к проверке того, что пара формальных моделей <реализация, спецификация> принадлежит допустимому множеству таких пар, определяемому отношением конформности.

Что делать, если реализация неизвестна (или её слишком сложно построить по исследуемой системе)? В этом случае возникает необходимость в тестировании, которое понимается как динамическая верификация конформности, то есть её проверка в процессе экспериментов. Конечно, для того, чтобы тестирование было возможным, сама конформность должна быть выражена в терминах взаимодействия реализации с окружающей средой. Тест взаимодействует с системой, подменяя собой окружение.

В данной работе мы будем рассматривать не любое тестирование, а только такое, которое основано на трёх предположениях.

Первое предположение. Мы будем рассматривать только *дискретное* тестовое взаимодействие, которое сводится к последовательности дискретных событий двух видов: тестовых воздействий на реализацию и наблюдений над поведением реализации. Эту последовательность мы будем называть *трассой*. Отметим, что в общем случае не всякое поведение реализации может наблюдаться в тестовом эксперименте, то есть в реализации могут происходить события, которые не наблюдаемы и, тем самым, не различимы между собой и не входят в трассу. Такое ненаблюдаемое поведение реализации традиционно обозначается символом τ и называется τ-активностью.

Дискретное взаимодействие моделируется с помощью, так называемой, машины тестирования, внутри которой находится реализация. Тестовому воздействию соответствует нажатие той или иной кнопки на клавиатуре машины, а наблюдению – появление его символа на экране дисплея. Таким образом, трасса – это последовательность кнопок и наблюдений. Тест понимается как инструкция оператору машины, в которой указывается, что оператор должен делать после той или иной трассы: нажимать кнопки (и какие кнопки) и/или ждать наблюдений.

Всё, что мы можем узнать о реализации с помощью тестирования, – это множество её трасс. Наблюдение в эксперименте некоторой трассы предполагает, что все её префиксы наблюдались в этом же эксперименте в более ранние моменты времени. Поэтому множество трасс реализации префикс-замкнуто. Тестовый эксперимент может быть пустым: ни одна кнопка не нажимается, и наблюдения не ожидаются. В этом случае естественно считать, что наблюдается пустая трасса. Тем самым, в каждой реализации есть пустая трасса, то есть множество трасс реализации не пусто.

Поскольку тестовое воздействие (нажатие кнопки) не зависит от самой реализации (оператор в любой момент времени может нажать любую кнопку), продолжение трассы реализации кнопкой также даёт трассу реализации. Это означает, что множество трасс реализации вместе с каждой трассой σ содержит и все трассы вида $\sigma\rho$, где ρ – последовательность кнопок.

Спецификацию теперь можно понимать как описание того, какие множества трасс реализаций правильные (конформные), а какие нет. В общем случае, если реализация имеет множество трасс $\mathbf{I}$, то конформна или неконформна не каждая отдельная трасса $\sigma \in \mathbf{I}$, а всё множество трасс $\mathbf{I}$ целиком.

Тест (как инструкция оператору) также задаётся непустым префикс-замкнутым множеством трасс. Тестовый эксперимент – это прогон теста, который заканчивается, когда получается максимальная в тесте трасса или «ответвление в сторону», то есть после некоторой трассы теста получается наблюдение, которым эта трасса не продолжается в тесте. Результатом прогона теста является трасса реализации $\sigma \in \mathbf{I}$. Различные прогоны одного и того же теста могут давать разные результаты, если реализация и/или тест недетерминированы.

Тест недетерминирован, если после какой-то трассы теста недетерминировано поведение оператора машины тестирования: он может как ждать наблюдения, так и нажимать кнопки, или он может только нажимать кнопки, но таких кнопок несколько. Иными словами, тест недетерминирован, если некоторая его немаксимальная трасса продолжается в тесте и наблюдениями и кнопками, или несколькими кнопками. Недетерминированный тест эквивалентен набору детерминированных тестов в том смысле, что они дают возможность наблюдения одного и того же множества трасс реализации. Поэтому, как правило, рассматривают только детерминированные тесты.

Что касается реализации, то предполагается, что её недетерминизм – это результат абстрагирования от некоторых неучитываемых внешних факторов – погодных условий, которые определяют выбор того или иного поведения детерминировано. *Гипотеза о глобальном тестировании* предполагает, что любые погодные условия могут быть воспроизведены в тестовом эксперименте. Для этого даже детерминированный тест должен прогоняться несколько раз, чтобы наблюдать все возможные для этого теста трассы реализации.

При тестировании выполняется прогон некоторых тестов из некоторого набора тестов при некоторых погодных условиях. Результатом тестирования является множество $\mathbf{X}$ трасс, наблюдаемых во всех этих тестовых экспериментах. Выносится вердикт ***pass*** (проходит) или ***fail*** (ошибка). Набор тестов *значимый*, если каждая конформная реализация его проходит, *исчерпывающий*, если каждая неконформная реализация его не проходит (обнаруживается ошибка), и *полный*, если он значимый и исчерпывающий. Заметим, что $\mathbf{X}$ – это не обязательно множество всех трасс реализации. Если

спецификация утверждает, что любая реализация с большим множеством трасс $\mathbf{I} \supseteq \mathbf{X}$ неконформна, то значимый набор тестов может (хотя и не обязан), а полный набор тестов должен выносить вердикт ***fail*** при наблюдении множества трасс $\mathbf{X}$ (или любого его надмножества). Если спецификация утверждает, что любая реализация с большим множеством трасс $\mathbf{I} \supseteq \mathbf{X}$, наоборот, конформна, то исчерпывающий (и полный) набор тестов может (хотя и не обязан), а полный набор тестов должен выносить вердикт ***pass*** при наблюдении множества трасс $\mathbf{X}$ (или любого его надмножества).

Второе предположение. В настоящей работе мы ограничимся только теми конформностями, которые отвечают *принципу независимости трасс*: любая трасса реализации конформна или неконформна независимо от других её трасс. Конформности такого типа называются *редукциями*. Не является редукцией, например, конформность, которая разрешает реализации иметь как трассу σ_1, так и трассу σ_2, но не обе одновременно. Принцип независимости исключает из рассмотрения конформности типа симуляций, которые основаны на соответствии состояний реализации и спецификации, а также конформности, которые требуют обязательного наличия в реализации тех или иных наблюдений после тех или иных трасс.

Для редукции можно считать, что спецификация s (прямо или косвенно) определяет множество *разрешаемых* трасс $\mathbf{\Sigma}$. Если реализация имеет множество трасс $\mathbf{I}$, то конформность означает вложенность $\mathbf{I} \subseteq \mathbf{\Sigma}$ и является частичным (нестрогим) порядком (рефлексивное, симметричное и транзитивное отношение). Трасса $\sigma \notin \mathbf{\Sigma}$ называется *ошибкой*.

При тестировании редукции обнаружение любой ошибки $\sigma \in \mathbf{I} \setminus \mathbf{\Sigma}$ означает, что реализация неконформна. Поэтому значимый набор тестов (тест) может (хотя и не обязан) выносить вердикт ***fail*** сразу, как только наблюдается такая трасса σ. Набор тестов исчерпывающий, если для каждой неконформной реализации хотя бы одна имеющаяся в ней ошибка $\sigma \in \mathbf{I} \setminus \mathbf{\Sigma}$ может быть обнаружена некоторым тестом из набора, то есть является трассой этого теста. Это означает, что неконформность реализации обнаруживается всегда за конечное время, тогда как вывод о конформности реализации может быть сделан, вообще говоря, только после всех прогонов при всех возможных погодных условиях всех тестов полного набора (число таких прогонов может быть бесконечно).

Третье предположение. На практике, естественно, используются только конечные тесты, точнее, тесты, которые заканчиваются за конечное время. При дискретном взаимодействии это означает, что при задании спецификации и генерации тестов используются только конечные трассы. Заметим, что для спецификации, основанной на конечных трассах, бесконечные тестовые эксперименты ничего не добавляют. Однако в общем случае это не так. Например, рассмотрим две реализации, в которых возможны только два наблюдения: x и y. Кнопки не используются. В одной реализации есть

бесконечная цепочка x. В другой реализации такой бесконечной цепочки нет, но есть бесконечный «веер» конечных цепочек x. В обеих реализациях нет перехода по y. При конечных тестовых экспериментах эти две реализации неразличимы. Для спецификации, в которой наблюдение y считается ошибкой после любого числа x, эти реализации обе конформны. В то же время бесконечный тестовый эксперимент позволяет эти реализации различить: в первой реализации есть бесконечная трасса x, а во второй нет. Если допускаются бесконечные трассы, то спецификация может трактовать бесконечную цепочку x как ошибку. Разумеется, такая ошибка не может быть найдена за конечное время.

По определению любая реализация, содержащая ошибку, неконформна. В то же время, кроме ошибок, определяемых спецификацией, то есть трасс, не принадлежащих множеству разрешаемых трасс $\mathbf{\Sigma}$, могут быть другие трассы (принадлежащие $\mathbf{\Sigma}$), которые, тем не менее, не встречаются в конформных реализациях. Такие трассы будем называть неконформными. После этого под ошибкой мы будем понимать любую неконформную трассу, а ошибки, определяемые спецификацией (трассы не из $\mathbf{\Sigma}$), будем называть ошибками 1-го рода. Ошибка 2-го рода – это неконформная трасса, не являющаяся ошибкой 1-го рода, то есть принадлежащая $\mathbf{\Sigma}$.

В данной статье рассматривается проблема зависимости между ошибками и тесно связанная с ней проблема оптимизации полного набора тестов. Будем говорить, что из множества ошибок A следует множество ошибок B и обозначать $A \rightarrow B$, если любая реализация, в которой есть ошибка из A, содержит ошибку из B. Если $A \rightarrow B$, то вместо тестов, которые ловят ошибки из A можно использовать тесты, которые ловят ошибки из B. Если A – это множество всех ошибок, то $B \subset A$ и, очевидно, $B \rightarrow A$. Если также $A \rightarrow B$, то множества A и B эквивалентны (обозначается $A \sim B$). Одним из таких подмножеств ошибок, эквивалентных множеству всех ошибок, является, конечно, множество ошибок 1-го рода. Однако могут существовать и другие множества ошибок, эквивалентные множеству ошибок 1-го рода, в том числе его строгие подмножества. Бывает и так, что множество ошибок 1-го рода бесконечно, но существует эквивалентное ему конечное множество ошибок. Это даёт возможность существенной оптимизации тестов.

Один вид такой зависимости между ошибками присущ любой конформности типа редукции для любого дискретного взаимодействия. Во-первых, любая реализация, содержащая трассу σ, содержит и трассу $\sigma\rho$, где ρ – последовательность кнопок. Поэтому, если $\sigma\rho$ ошибка, то σ тоже ошибка. Поэтому, если $\sigma\rho \in A$, то $A \rightarrow A \cup \{\sigma\}$. Во-вторых, множество трасс реализации префикс-замкнуто. Поэтому если ошибка μ является префиксом трассы σ (будем обозначать это $\mu \leq \sigma$), то σ тоже ошибка. Поэтому, если $\mu \in A$, то $A \rightarrow A \setminus \{\sigma\}$. Это даёт возможность следующей оптимизации тестов: для полноты тестирования достаточно обнаруживать только такие ошибки,

которые минимальны по префиксности во множестве всех ошибок (а не только ошибок 1-го рода), такие ошибки не заканчиваются кнопками. Множество таких ошибок эквивалентно множеству ошибок 1-го рода и, тем самым, множеству всех ошибок.

В то же время существует много различных конформностей типа редукции, для которых между ошибками имеются и другие зависимости. Нахождение таких зависимостей и связанная с этим оптимизация тестов иногда представляют собой трудную задачу [например, [6],[7]].

Цель данной статьи – определить общую природу зависимостей между ошибками. Для этого мы формально определим общую модель дискретного взаимодействия и общую конформность типа редукции. Мы покажем следующее. Во-первых, для такой общей редукции не существует зависимости между ошибками, кроме указанной выше. Во-вторых, другие конформности типа редукции являются частным случаем общей редукции, то есть сводятся к ней. При этом сужается класс рассматриваемых спецификаций и тестируемых реализаций. В-третьих, сужение класса спецификаций не влияет на зависимость между ошибками, тогда как сужение класса тестируемых реализаций влечет появление дополнительных зависимостей между ошибками [например, [6],[7]]. Важно отметить, что каждая такая частная редукция определяет некий естественный для неё класс тестируемых реализаций. Однако на практике часто используются дополнительные ограничения на тестируемые реализации, что, в свою очередь, также приводит к появлению дополнительных зависимостей между ошибками [например, [14],[15]]. Иными словами, мы сведём проблему зависимостей между ошибками на классе тестируемых реализаций, естественном для той или иной частной редукции, к общей проблеме зависимостей между ошибками, возникающей как результат сужения класса тестируемых реализаций.

2. Общая модель

В этом разделе мы формально определим общую модель дискретного взаимодействия и общую конформность типа редукции.

2.1. Семантика взаимодействия

Для многих семантик взаимодействия частного вида между кнопками и наблюдениями существует та или иная предустановленная связь. Некоторые из таких семантик мы рассмотрим ниже. В общем же случае никакой предустановленной связи между кнопками и наблюдениями мы предполагать не будем. Другое дело, что в конкретной реализации такая связь может быть.

Будем считать, что заданы два непересекающихся универсума символов: **B** – тестовых воздействий (кнопок – **b**uttons) и **O** – наблюдений (**o**bservations). Такую семантику будем называть ***B*/*O*-*семантикой***. Трасса – это

последовательность в алфавите **B**∪**O**. Семантику будем называть *конечной*, если суммарное число кнопок и наблюдений конечны.

2.2. Машина тестирования

B/**O**-семантика моделируется машиной тестирования, представляющей собой «чёрный ящик», внутри которого находится реализация. Машина снабжена клавиатурой для управления и дисплеем для наблюдения.

Клавиатура представляет собой множество кнопок **B**. Тестовое воздействие осуществляется нажатием той или иной кнопки на клавиатуре. Когда нажимается кнопка, машина тестирования передаёт в реализацию однократный сигнал о соответствующем тестовом воздействии и дожидается ответного сигнала о том, что реализация «приняла к сведению» это тестовое воздействие, после чего машина может передавать в реализацию сигнал о следующем тестовом воздействии. Кнопка не фиксируется (автоматически отжимается), что даёт возможность оператору машины нажимать следующую (другую или ту же самую) кнопку. Одновременно можно нажимать только одну кнопку, соответствующую ровно одному тестовому воздействию.

На дисплее машины последовательно высвечиваются символы кнопок и наблюдений, то есть символы из **B**∪**O**. Для того чтобы оператор машины мог различать идущие подряд одинаковые символы, между ними экран кратковременно гаснет. Последовательность символов, появляющихся на экране, как раз и является трассой, наблюдаемой в процессе тестового эксперимента. Символ кнопки появляется на экране в тот момент, когда оператор нажимает соответствующую кнопку. Наблюдение появляется на экране дисплея тогда, когда в реализации происходит соответствующее наблюдаемое событие. Ненаблюдаемая τ-активность реализации никак не отражается на экране дисплея.

Для того чтобы можно было выполнять несколько тестовых экспериментов, машина может быть снабжена кнопкой *рестарта*. Эта кнопка сбрасывает реализацию в начальное состояние и гасит экран. Каждый новый рестарт машины может вызывать изменение погодных условий, от которых зависит поведение реализации. Гипотеза о глобальном тестировании предполагает, что в последовательности рестартов воспроизводятся все возможные погодные условия.

В то же время рестарт позволяет выполнить не более чем счётное число тестовых экспериментов, тем самым, для не более чем счётного числа погодных условий. Для того чтобы обойти это ограничение, машина тестирования может быть снабжена не кнопкой рестарта, а кнопкой *репликации*. Однократное нажатие такой кнопки создаёт множество копий машины тестирования произвольной мощности. Тестирование происходит с каждой копией машины независимым образом, то есть на каждой копии выполняется свой тестовый эксперимент. Для каждой копии фиксируется свой вариант погодных условий. Гипотеза о глобальном тестировании

предполагает, что для каждого варианта погодных условий при репликации создается, по крайней мере, одна копия машины.

Важно отметить, что для конформностей типа редукции репликацию достаточно делать один раз перед началом тестирования, а не много раз после получения тех или иных трасс. Многократная репликация (после каждого шага тестирования, то есть после каждого наблюдения и после нажатия каждой кнопки) требуется для конформностей типа симуляции.

2.3. Реализация

Для конформности типа редукции реализация, фактически, сводится к множеству её трасс. Такая *трассовая модель реализации* формально определяется как множество $I \subseteq (\mathbf{B} \cup \mathbf{O})^*$, которое: 1) не пусто, 2) префикс-замкнуто, 3) вместе с каждой трассой σ содержит и все трассы вида $\sigma\rho$, где ρ – последовательность кнопок.

Для компактного задания множества трасс, в частности, для задания бесконечного множества трасс конечным образом, используется модель LTS (Labelled Transition System). Она представляет собой ориентированный граф, вершины которого называются состояниями, одно состояние выделено в качестве начального, дуги помечены символами из $\mathbf{B} \cup \mathbf{O}$ и называются переходами. Ненаблюдаемая τ-активность понимается как цепочка элементарных τ-событий, каждое из которых изображается переходом, помеченным символом τ. LTS-реализацию будем называть *конечной*, если конечно число её состояний, достижимых из начального состояния.

Поскольку тестовое воздействие (нажатие кнопки машины тестирования) на реализацию выполняется извне её и не зависит от неё, переход по кнопке означает лишь тот факт, что реализация «узнала» о выполненном тестовом воздействии. В результате такого перехода реализация меняет своё состояние, что впоследствии может привести к изменению её поведения, то есть к появлению других наблюдений. Если в некотором состоянии реализация игнорирует тестовое воздействие, то это эквивалентно тому, что в этом состоянии есть переход-петля по этой кнопке. Поэтому отсутствие перехода по кнопке в состоянии реализации трактуется как наличие перехода-петли по этой кнопке в этом состоянии.

Маршрутом называется последовательность смежных переходов, когда начало любого перехода, кроме первого, совпадает с концом предыдущего перехода. Трасса реализации – это последовательность пометок переходов маршрута, начинающегося в начальном состоянии, с пропуском символа τ.

Множество трасс LTS-реализации является трассовой моделью реализации и, наоборот, для любой трассовой модели реализации существует LTS с таким же множеством трасс.

Очевидно, что для каждой трассы σ существует наименьшая по множеству трасс реализация, содержащая эту трассу σ, – это множество трасс $\{\mu\rho \mid \mu \le \sigma \ \& \ \rho \in B^*\}$. Соответствующая LTS-реализация изображена на 0.

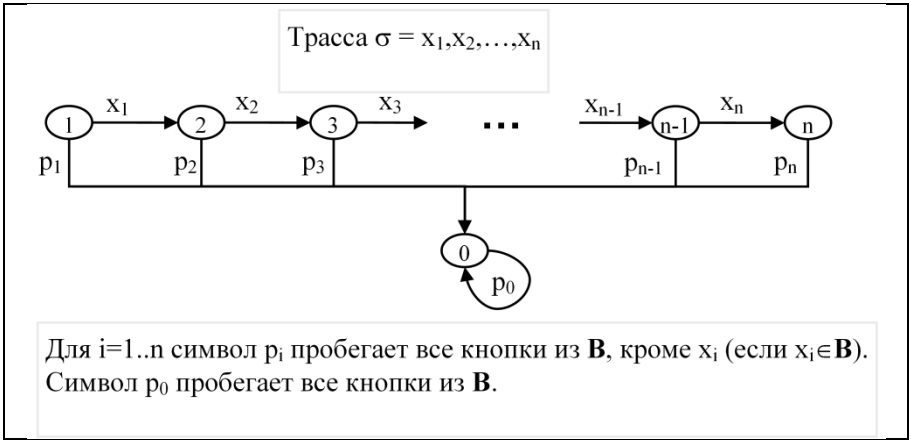


Рис. 1.

2.4. Взаимодействие с реализацией

При тестировании в любой наблюдаемой трассе подпоследовательность кнопок содержит ровно те кнопки, которые оператор машины нажимал и ровно в том порядке, в котором он их нажимал. С внешней точки зрения тестовое воздействие влияет лишь на те наблюдения, которые появляются в трассе. Иными словами, нажатие кнопки лишь регулирует поток наблюдений над поведением реализации. Это достигается с помощью переходов по кнопкам, которые меняют состояние реализации при нажатии кнопки и, тем самым, дальнейший поток наблюдений.

Что касается ненаблюдаемого поведения реализации, то мы исходим из *основного допущения о τ-активности*: между любыми двумя наблюдениями (и до первого в трассе наблюдения) в реализации может быть любая конечная τ-активность [[10]]. В терминах LTS это означает, что между двумя переходами по наблюдению (и перед первым таким переходом) реализация может выполнить любое конечное число τ-переходов. Мы распространяем это допущение также на тестовые воздействия: реализация может выполнить любое конечное число τ-переходов между любыми двумя переходами по наблюдениям или кнопкам (и перед первым таким переходом).

Особенностью нашей модели взаимодействия является *приоритет тестового воздействия над поведением реализации*, как наблюдаемым, так и ненаблюдаемым. Если после получения в тестовом эксперименте трассы σ

оператор не нажимает кнопку, а ждёт наблюдений, то может быть получено любое наблюдение, которое в реализации есть после трассы σ, то есть может быть получена любая имеющаяся в реализации трасса σu, где u – наблюдение. Кроме того, если после трассы σ в реализации есть бесконечная τ-активность (*дивергенция* как бесконечная цепочка τ-переходов), то никаких наблюдений может и не быть. Если же сразу после наблюдения трассы σ оператор нажимает кнопку p, то реализация обязана выполнить переход по кнопке p, будет получена трасса σp. Однако по основному допущению о τ-активности между переходом по последнему символу (наблюдению или кнопке) трассы σ и переходом по следующему наблюдению u в трассе σu или следующей кнопке p в трассе σp реализация может выполнить любое конечное число τ-переходов. Итак, реализация обязана «принять к сведению» тестовое воздействие через конечное время после нажатия соответствующей кнопки. В то же время мы никак не оговариваем, что означает это «принятие к сведению», допускается и простое игнорирование тестового воздействия, что в LTS моделируется переходом-петлёй по этой кнопке (отсутствие перехода по кнопке интерпретируется как наличие такой петли).

В результате мы получаем следующий протокол взаимодействия. Если нет тестового воздействия, то есть никакая кнопка не нажата, реализация может выполнить, в зависимости от погодных условий, любую цепочку переходов по наблюдениям и τ-переходов, начинающуюся в её текущем состоянии. Если осуществляется тестовое воздействие, то есть оператор нажал кнопку p, то реализация может выполнить, в зависимости от погодных условий, любую конечную цепочку τ-переходов, после чего должна выполнить любой переход по кнопке p. Выполняя p-переход, реализация как бы «сообщает» машине тестирования о том, что тестовое воздействие ею воспринято. После этого реализация готова к восприятию следующего тестового воздействия (нажатию той или иной кнопки).

Заметим, что при таком протоколе взаимодействия появление на экране дисплея символа кнопки в момент нажатия кнопки, фактически, эквивалентно его появлению в момент совершения реализацией перехода по этой кнопке. Именно поэтому наблюдаемая на экране дисплея трасса совпадает с трассой маршрута, который реализация за это время проходит.

При наличии в состоянии нескольких переходов, которые реализация может выполнять, выбирается один из них недетерминированным образом. Также при нажатой кнопке выбор числа τ-переходов, которые выполняются до перехода по кнопке, происходит недетерминировано. Оба этих выбора понимаются как выборы в зависимости от погодных условий. Гипотеза о глобальном тестировании гарантирует возможность перебора всех возможных погодных условий.

В любом сеансе взаимодействия с реализацией через машину тестирования на экране дисплея наблюдается некоторая трасса реализации, а также (в более ранние моменты времени) все ее префиксы.

2.5. Оператор машины тестирования

Оператор машины тестирования моделирует работу тестовой системы. Мы предполагаем, что при тестировании оператор выполняет тест, понимаемый как инструкция оператору. В этой инструкции указывается, что может делать оператор после получения той или иной трассы: ждать наблюдений и/или нажимать кнопки и какие именно кнопки. Естественно, что если тест определяет поведение оператора после трассы μ, то для того, чтобы можно было получить эту трассу μ, тест должен определять поведение оператора и после любого префикса трассы μ.

Если тест разрешает оператору нажимать кнопку p после трассы μ, то предполагается, что оператор может нажать эту кнопку через любое время после получения трассы μ. Тем самым, оператору *не запрещено* выдерживать любые паузы после получения тех или иных трасс, в том числе перед нажатием следующей кнопки: в любой момент времени он может устроить себе «перерыв на чай». Это означает, что когда оператор нажимает кнопку p спустя какое-то время после трассы μ, а после нажатия кнопки ждёт ещё какое-то время, то реально будет получена не трасса μp, а трасса $\mu\pi_1 p\pi_2$, где π_1 и π_2 – последовательности наблюдений.

В то же время для полноты тестирования необходимо, чтобы любая интересующая нас трасса реализации могла наблюдаться при взаимодействии с реализацией через машину тестирования. А для этого оператор должен иметь *возможность* достаточно быстро нажимать кнопки после полученных трасс. Тогда, нажимая кнопку p сразу после трассы μ, оператор будет наблюдать именно трассу μp. Разумеется, если после этой трассы оператор какое-то время не выключает машину и не нажимает кнопок, то может быть получено продолжение этой трассы последовательностью наблюдений, то есть трасса $\mu p\pi_2$.

2.6. Спецификация и общая редукция

Как было сказано во введении, спецификация явно или неявно определяет множество разрешённых трасс и одновременно его дополнение – множество ошибок 1-го рода. В данной работе под спецификацией будет пониматься произвольное множество конечных трасс, понимаемое просто как множество ошибок 1-го рода. Реализация конформна, если в ней нет ошибок 1-го рода, то есть трасс спецификации. Такую конформность будем называть далее *общей редукцией*.

Спецификацию как множество ошибок 1-го рода можно задавать с помощью порождающего графа, то есть LTS с выделенными конечными вершинами.

LTS-спецификацию будем называть *конечной*, если конечно число её состояний, достижимых из начального состояния. Для некоторых бесконечных множеств ошибок 1-го рода LTS-спецификация может быть конечной. Как известно, для любого порождающего графа существует процедура детерминизации, строящая детерминированный граф, порождающий то же множество последовательностей. Поэтому для любой спецификации существует детерминированная LTS-спецификация, определяющая то же множество трасс. Детерминированность здесь означает, что в каждом достижимом состоянии нет τ-переходов и для каждого символа $x \in \mathbf{B} \cup \mathbf{O}$ определено не более одного перехода по x из этого состояния. Если LTS-спецификация конечная, то после детерминизации она тоже остаётся конечной.

Спецификация s задаёт класс конформных реализаций, который будем обозначать как C_s. Если спецификация содержит пустую трассу, то есть пустая трасса считается ошибкой 1-го рода, то все реализации неконформны, поскольку любая реализация содержит пустую трассу. Если спецификация – это пустое множество, то все реализации конформны.

2.7. Тест

Тестом будем называть множество t конечных трасс. Получение при тестировании любой из этих трасс приводит к вынесению вердикта ***fail***, получение любой другой трассы – к получению вердикта ***pass***. Тест понимается как инструкция оператору машины тестирования. Задача теста – проверить, имеется ли в реализации хотя бы одна из трасс теста: если это так, то тестирование заканчивается и выносится общий вердикт ***fail***.

Нажатие кнопки. В процессе тестирования после получения трассы μ оператор может нажать кнопку p, если трасса μp является префиксом некоторой трассы $\sigma \in t$. Предполагается, что если трасса μp, где p кнопка, является префиксом некоторой трассы $\sigma \in t$, то хотя бы в одном сеансе тестирования после получения трассы μ оператор нажимает кнопку p, причем достаточно быстро после получения трассы μ. Также предполагается, что если трасса μu, где u наблюдение, является префиксом некоторой трассы $\sigma \in t$, то хотя бы в одном сеансе тестирования после получения трассы μ оператор ожидает наблюдения. Если эти предположения выполнены, то гипотеза о глобальном тестировании гарантирует, что если некоторая трасса $\sigma \in t$ встречается в реализации, то она будет получена хотя бы в одном сеансе тестирования.

Выключение машины. Обозначим префикс-замыкание множества t последовательностей: $\boldsymbol{pre}(t) = \{\mu \mid \exists \sigma \in t\ \mu \le \sigma\}$. В процессе тестирования после получения трассы μ возможны три случая:

1) $\mu \in \boldsymbol{pre}(t) \setminus t$, то есть μ является строгим префиксом какой-либо трассы из t;
2) $\mu \in t$;

3) $\mu \notin \boldsymbol{pre}(t)$, то есть μ не является префиксом какой-либо трассы из t.

В случае 1 оператор должен продолжать сеанс тестирования, то есть не должен выключать машину тестирования. В случае 2 и 3 оператор должен закончить сеанс тестирования, то есть должен выключить машину тестирования. При этом выносится вердикт: в случае 2 – ***fail***, в случае 3 – ***pass***.

Трасса, которая может быть получена в некотором сеансе тестирования с данным тестом (не только в конце сеанса), имеет вид либо $\mu\pi$, где μ является префиксом некоторой трассы $\sigma \in t$, а π последовательность наблюдений, либо $\mu\pi_1 p\pi_2$, где μp является префиксом некоторой трассы $\sigma \in t$, p – кнопка, а π_1 и π_2 последовательности наблюдений. Множество таких трасс будем называть расширением теста и обозначать:

$$\boldsymbol{exp}(t) = \{\mu\pi \mid \mu \in \boldsymbol{pre}(t) \,\&\, \pi \in \mathbf{O}^*\} \cup \{\mu\pi_1 p\pi_2 \mid \mu p \in \boldsymbol{pre}(t) \,\&\, p \in \mathbf{B} \,\&\, \pi_1 \in \mathbf{O}^* \,\&\, \pi_1 \in \mathbf{O}^*\}.$$

Множество трасс, которые могут быть получены в конце сеанса тестирования, равно $(\boldsymbol{exp}(t) \setminus \boldsymbol{pre}(t)) \cup t = \boldsymbol{exp}(t) \setminus (\boldsymbol{pre}(t) \setminus t)$.

Реализация *проходит* тест, если при любом сеансе тестирования (при любых погодных условиях) выносится вердикт ***pass***. Реализация *проходит* набор тестов, если она проходит каждый тест из набора. Для заданного класса реализаций (в частности, для класса всех реализаций) набор тестов (тест) значимый, если каждая конформная реализация из этого класса его проходит, исчерпывающий, если каждая неконформная реализация из этого класса его не проходит, и полный, если он значимый и исчерпывающий.

Тест *детерминированный*, если он однозначно определяет поведение оператора. Это значит, что любая трасса из префикс-замыкания теста в этом префикс-замыкании либо продолжается одной кнопкой и не продолжается наблюдениями, либо не продолжается кнопками:

$$\forall \mu \in \boldsymbol{pre}(t)\ (\ |\{p \in \mathbf{B} \mid \mu p \in \boldsymbol{pre}(t)\}| = 1 \,\&\, \{u \in \mathbf{O} \mid \mu u \in \boldsymbol{pre}(t)\} = \varnothing\ ) \vee \{p \in \mathbf{B} \mid \mu p \in \boldsymbol{pre}(t)\} = \varnothing.$$

Тест *примитивный*, если он содержит только одну трассу. Очевидно, что примитивный тест детерминированный. Любой тест t эквивалентен объединению множества примитивных тестов в том смысле, что они выносят вердикт ***fail*** для одних и тех же реализаций: $t = \cup\ \{\ \{\sigma\} \mid \sigma \in t\ \}$. Также очевидно, что спецификация (как множество ошибок 1-го рода) является полным тестом на классе всех реализаций. Отсюда следует, что набор примитивных тестов, построенных по всем ошибкам 1-го рода, то есть по всем трассам спецификации, является полным на классе всех реализаций.

2.8. Нормализация спецификации и оптимизация тестов

Как уже было отмечено во введении, кроме ошибок 1-го рода, то есть трасс спецификации, могут быть и другие ошибки – неконформные трассы, то есть трассы, не встречающиеся в конформных реализациях. Ошибки, не являющиеся ошибками 1-го рода, называются ошибками 2-го рода. Для полноты тестирования достаточно обнаруживать только такие ошибки, которые минимальны по префиксности во множестве всех ошибок (а не только ошибок 1-го рода). Такие ошибки будем называть *первичными* ошибками, *вторичная* ошибка – это ошибка, у которой есть строгий префикс, являющийся ошибкой. Первичные ошибки не заканчиваются кнопками. Множество первичных ошибок эквивалентно множеству ошибок 1-го рода и, тем самым, множеству всех ошибок. Оно, очевидно, является наименьшим по вложенности подмножеством ошибок, эквивалентным множеству всех ошибок. Его можно рассматривать как спецификацию, которую будем называть *нормализованной* спецификацией.

Для каждой спецификации s множество всех ошибок строится с помощью систематического применения следующих операций:

1) Если p – кнопка и $\sigma p \in s$, то добавляем в s трассу σ.
2) Если $\mu \in s$ и $\mu < \sigma$, то добавляем в s трассу σ.

Если s* – множество всех ошибок для спецификации s, то процедура нормализации сводится к удалению неминимальных по префиксности ошибок: если $\mu \in s$, $\sigma \in s$ и $\mu < \sigma$, то удаляем из s трассу σ.

Нормализацию можно выполнить и непосредственно по исходной спецификации s как систематическое применение следующих операций:

3) Если p – кнопка и $\sigma p \in s$, то добавляем в s трассу σ.
4) Если $\mu \in s$, $\sigma \in s$ и $\mu < \sigma$, то удаляем из s трассу σ.

Нормализованные спецификации взаимно-однозначно соответствуют своим классам конформных реализаций: $a = b \Leftrightarrow C_a = C_b$.

Пусть спецификация s нормализованная. Как было отмечено выше, для каждой трассы σ существует наименьшая по множеству трасс реализация, содержащая эту трассу σ, – это множество трасс $\{\mu\rho \mid \mu \le \sigma \,\&\, \rho \in B^*\}$. Отсюда следует, что набор T тестов значимый тогда и только тогда, когда каждая трасса каждого теста из набора имеет в качестве префикса ошибку из s, то есть s коинициально $\cup$T. Набор T тестов исчерпывающий тогда и только тогда, когда каждая трасса из s имеет префикс, являющийся трассой некоторого теста из набора, то есть $\cup$T коинициально s.

Набор T тестов полный тогда и только тогда, когда s и $\cup$T взаимно коинициальны. Поскольку s нормализована, условие $\cup$T коинициально s можно заменить на условие вложенности $s \subseteq \cup T$. Действительно, в противном

случае найдётся трасса $\mu \in s \setminus \cup T$, а тогда, поскольку $\cup T$ коинициально s, найдётся трасса $\mu_1 < \mu$ такая, что $\mu_1 \in \cup T$, а тогда, поскольку s коинициально $\cup T$, найдётся трасса $\mu_2 \le \mu_1$ такая, что $\mu_2 \in s$, следовательно, в s имеются две ошибки $\mu_2 < \mu$, что противоречит нормализованности s.

Иными словами, набор T тестов полный тогда и только тогда, когда все трассы всех его тестов – это все первичные ошибки (трассы из нормализованной спецификации s) и некоторые их продолжения. Очевидная оптимизация – это удаление таких продолжений, что даёт в итоге набор T` тестов, множество всех трасс всех тестов которого – это нормализованная спецификация: $s = \cup\{\{\sigma\} \mid \sigma \in s\} = \cup T`$. Тем самым, оптимизированный полный набор T` тестов – это покрытие s, а набор примитивных тестов $\{\{\sigma\} | \sigma \in s\}$ является одним из возможных разбиений s.

3. Класс реализаций

По разным причинам в качестве тестируемых реализаций рассматриваются не любые реализации, а принадлежащие тому или иному классу реализаций I. Это приводит к появлению дополнительных зависимостей между ошибками (в том числе, первичными) и, соответственно, даёт возможность дополнительной оптимизации тестов.

Первое определение эквивалентности спецификаций: Будем говорить, что две спецификации a и b эквивалентны на классе реализаций I, если на этом классе спецификации определяют одну и ту же конформность реализаций: $I \cap C_a = I \cap C_b$. Если I – класс всех реализаций, то $C_a \subseteq I$ и $C_b \subseteq I$, поэтому эквивалентность нормализованных спецификаций ($C_a = C_b$) совпадает с равенством (a=b). На других подклассах реализаций это, вообще говоря, не верно.

Трассу, встречающуюся в реализациях класса I, будем называть *актуальной* на классе I. Если I – множество трассовых реализаций, то множество актуальных трасс равно $\cup I$. На классе всех реализаций все трассы актуальны. На других классах реализаций могут быть как актуальные, так и неактуальные трассы. Трасса, которая встречается в конформных реализациях из класса I, называется конформной на классе I. Это конформная трасса, которая актуальна на классе I. Ошибки (в том числе ошибки 1-го рода и 2-го рода) делятся на актуальные и неактуальные. При тестировании реализаций из класса I, очевидно, достаточно обнаруживать только актуальные на этом классе ошибки. Множество трасс, конформных на классе I, для спецификации s равно $\cup(I \cap C_s)$. Соответственно, множество ошибок, актуальных на классе I, равно $\cup I \setminus \cup(I \cap C_s)$.

Это дает нам второе определение эквивалентности спецификаций: Будем говорить, что две спецификации a и b эквивалентны на классе реализаций I, если они определяют одно и то же множество актуальных ошибок: $\cup I \setminus \cup(I \cap C_a) = \cup I \setminus \cup(I \cap C_b)$.

На самом деле, два определения эквивалентности спецификаций равносильны:

$$I \cap C_a = I \cap C_b \Leftrightarrow \cup I \setminus \cup(I \cap C_a) = \cup I \setminus \cup(I \cap C_b).$$

Докажем это. Сначала покажем, что $\cup(I \cap C_a) = \cup(I \cap C_b) \Leftrightarrow \cup I \setminus \cup(I \cap C_a) = \cup I \setminus \cup(I \cap C_b)$.

Действительно, если $\cup(I \cap C_a) = \cup(I \cap C_b)$, то, очевидно, $\cup I \setminus \cup(I \cap C_a) = \cup I \setminus \cup(I \cap C_b)$.

Покажем, что если $\cup I \setminus \cup(I \cap C_a) = \cup I \setminus \cup(I \cap C_b)$, то $\cup(I \cap C_a) = \cup(I \cap C_b)$. Пусть это не верно, например, трасса $\sigma \in \cup(I \cap C_a) \setminus \cup(I \cap C_b)$. Тогда эта трасса принадлежит некоторой реализации из I, которая конформна для a. Но тогда эта реализация не принадлежит C_b, то есть в ней есть ошибка из b. Эта ошибка принадлежит $\cup(I \cap C_a)$, следовательно, не принадлежит $\cup I \setminus \cup(I \cap C_a)$. Также эта ошибка принадлежит $\cup I$, но не принадлежит $\cup(I \cap C_b)$, следовательно, принадлежит $\cup I \setminus \cup(I \cap C_b)$, что противоречит равенству $\cup I \setminus \cup(I \cap C_a) = \cup I \setminus \cup(I \cap C_b)$. Теперь покажем, что $I \cap C_a = I \cap C_b \Leftrightarrow \cup(I \cap C_a) = \cup(I \cap C_b)$. Если $I \cap C_a = I \cap C_b$, то, очевидно, $\cup(I \cap C_a) = \cup(I \cap C_b)$. Покажем, что если $\cup(I \cap C_a) = \cup(I \cap C_b)$, то $I \cap C_a = I \cap C_b$. Пусть это не верно, тогда есть некоторая реализация, принадлежащая, например, $I \cap C_a \setminus I \cap C_b$. Тогда эта реализация принадлежит I и не принадлежит C_b, следовательно, в ней есть некоторая ошибка из b. Тем самым, эта ошибка принадлежит $\cup(I \cap C_a)$. Но эта ошибка не может принадлежать $\cup(I \cap C_b)$, что противоречит равенству $\cup(I \cap C_a) = \cup(I \cap C_b)$.

Теперь пусть I – это отображение, задающее для каждой спецификации s класс тестируемых реализаций I_s. Будем говорить, что для отображения I спецификация b *может быть использована вместо спецификации* a, если 1) $I_a \subseteq I_b$, 2) $I_a \cap C_a = I_a \cap C_b$. Первое условие говорит о том, что любая реализация, которую мы могли тестировать для проверки конформности спецификации a, можно тестировать для проверки конформности спецификации b. Второе условие (эквивалентность спецификаций на классе I_a) говорит о том, что спецификации a и b определяют одинаковую конформность реализаций на классе тестируемых реализаций для спецификации a.

Ошибка обнаруживается набором тестов, если она является трассой одного из тестов набора. Любой набор тестов, который полон на классе тестируемых реализаций I, очевидно, задаёт множество обнаруживаемых ошибок (множество всех трасс всех его тестов), эквивалентное на классе I множеству всех ошибок.

4. Гипотеза о безопасности

В наших работах [[1],[2],[5],[6]] мы ввели понятие безопасного тестирования. Это такое тестирование, при котором не проходятся трассы реализации, которые считаются опасными. Гипотеза о безопасности определяет класс

реализаций, которые можно безопасно тестировать для проверки конформности данной спецификации.

4.1. Общий вид гипотезы о безопасности

Будем говорить, что задана *гипотеза о безопасности*, если для каждой реализации i определено префикс-замкнутое подмножество ***SafeTraces***(i) ее трасс, которые называются безопасными трассами. Тестирование данной реализации называется безопасным, если в процессе него могут получаться только безопасные трассы этой реализации. Реализация i безопасна для теста t, если тестирование с помощью этого теста безопасно для этой реализации: ***exp***(t) $\subseteq$ ***SafeTraces***(i). Каждый тест t определяет класс безопасных реализаций ***SafeImpl***(t) = { i | ***exp***(t) $\subseteq$ ***SafeTraces***(i) }. Набор тестов T определяет класс реализаций, безопасных для каждого теста из набора: ***SafeImpl***(T) = { i | $\forall$t$\in$T ***exp***(t) $\subseteq$ ***SafeTraces***(i) } = $\cap$ { ***SafeImpl***(t) | t$\in$T }. Спецификация s определяет класс безопасных реализаций как класс реализаций, безопасных для полного теста s или, что то же самое, для набора примитивных тестов, построенных по ошибкам спецификации: ***SafeImpl***(s) = ***SafeImpl***({{σ}|$\sigma\in$s}). В общем случае, если предполагается безопасное тестирование реализаций из заданного класса I, тестироваться будут безопасные реализации из класса I, то есть реализации из класса I $\cap$ ***SafeImpl***(s).

4.2. Гипотеза о конечном времени ожидания наблюдения

Для того чтобы каждый сеанс тестирования был конечным по времени, нужно, чтобы были конечными времена ожидания кнопок и наблюдений на экране дисплея: 1) кнопка должна появляться на экране дисплея через конечное время после ее нажатия, 2) если оператор ждет наблюдений, то какое-нибудь наблюдение должно появиться на экране дисплея через конечное время.

Первое условие гарантированно выполнено в данной модели взаимодействия с реализацией. А второе условие может и не выполняться. Сформулируем требование к реализации, чтобы выполнялось это второе условие для данного теста.

Гипотеза о наблюдениях – λ-гипотеза[1]: если трасса реализации является префиксом трассы теста и продолжается в префикс-замыкании теста наблюдением, то в реализации она также должна продолжаться каким-нибудь (не обязательно тем же самым) наблюдением при любом поведении реализации. Это означает: 1) в реализации в каждом стабильном состоянии (состоянии, в котором не начинаются τ-переходы) после этой трассы имеется

[1] Символом λ принято обозначать ситуацию, когда возникает deadlock или дивергенция [[10]]

переход по какому-нибудь наблюдению, 2) после трассы нет дивергенции. λ-гипотеза является частным случаем гипотезы о безопасности.

Определим формально множество $SafeTraces_{\lambda}(i)$ безопасных трасс реализации i для λ-гипотезы. λ-трассой реализации будем называть трассу реализации, которая заканчивается в стабильном состоянии, где нет переходов по наблюдениям, или в дивергентном состоянии. Трассу реализации будем называть безопасной, если любой ее строгий префикс, за которым в трассе следует наблюдение, не является λ-трассой.

λ-гипотеза не меняет актуальность трасс: все трассы актуальны. λ-гипотеза меняет конформность трасс: на классе безопасных реализаций, определяемом этой гипотезой, трасса μ неконформна, если для каждого наблюдения u трасса μu является ошибкой. Для определения первичных ошибок спецификации в случае λ-гипотезы применяется следующая **процедура λ-нормализации спецификации**: систематически применяем три действия:

1) Если для каждого наблюдения u трасса $\sigma u \in s$, то добавляем в s трассу σ.
2) Если p – кнопка и $\sigma p \in s$, то добавляем в s трассу σ.
3) Если $\mu \in s$, $\sigma \in s$ и $\mu < \sigma$, то удаляем из s трассу σ.

Полученное множество трасс – это множество первичных ошибок в случае λ-гипотезы.

4.3. Гипотеза о разрушении

Другой разновидностью гипотезы о безопасности является, так называемая, гипотеза о разрушении. Под разрушением понимается любое поведение реализации, которое нежелательно во время тестирования [[1],[2],[5]]. Причины нежелательности того или иного поведения могут быть самыми разными, мы не налагаем здесь никаких ограничений. Для изображения разрушения в LTS-модели реализации некоторые её переходы по наблюдениям или τ-переходы (ненаблюдаемое поведение) заменяются γ-переходами, то есть переходами, помеченными специальным символом разрушения – γ.

Теперь под моделью реализации мы будем понимать LTS в алфавите с добавленным символом γ. Поскольку нас не интересует поведение реализации после разрушения, под трассой будем понимать последовательность кнопок и наблюдений, быть может, заканчивающуюся разрушением.

Гипотеза о разрушении – γ-гипотеза: для спецификации s любая трасса из ***exp***(s) не продолжается в реализации разрушением. γ-гипотеза является частным случаем гипотезы о безопасности.

Определим формально множество $\boldsymbol{SafeTraces_{\gamma}}(i)$ безопасных трасс реализации i для γ-гипотезы: трассу реализации будем называть безопасной, если любой ее префикс не продолжается в реализации разрушением.

γ-гипотеза меняет актуальность трасс: трасса актуальная, если она не имеет вида μγ, где $\mu \in \boldsymbol{exp}(s)$. γ-гипотеза не меняет конформность актуальных трасс: на классе безопасных реализаций, определяемом этой гипотезой, неконформна та и только та актуальная трасса, префикс которой является ошибкой. Поскольку γ-гипотеза не меняет конформность актуальных трасс, процедура нормализации не требуется: все ошибки спецификации являются первичными.

λ- и γ-гипотезы в совокупности определяют класс безопасных реализаций $\boldsymbol{SafeImpl_{\lambda\gamma}}(s) = \boldsymbol{SafeImpl_{\lambda}}(s) \cap \boldsymbol{SafeImpl_{\gamma}}(s)$.

5. Моделирование других семантик

В этом разделе мы покажем, каким образом некоторые известные конформности типа редукции сводятся к общей редукции в **B/O**-семантике, описанной выше.

5.1. R/Q-семантика

В наших работах [[1],[5],[6]] рассматривалась **R/Q**-семантика как обобщение многих семантик взаимодействия, в частности, семантики популярного отношение ***ioco*** [[12],[13]]. **R/Q**-семантика задаётся двумя непересекающимися множествами кнопок: **R** и **Q**. Фиксируется жёсткая связь между кнопками и наблюдениями таким образом, что каждой кнопке p однозначно соответствует подмножество $\boldsymbol{obs}(p)$ наблюдений, «разрешаемых» этой кнопкой. Наблюдения делятся на действия из фиксированного алфавита **L** и отказы – подмножества **L**, причём отказ $r \subseteq \mathbf{L}$ понимается как отсутствие действий из множества r. **R**-кнопке соответствует как множество $r \subseteq \mathbf{L}$ разрешаемых ею действий, так и отказ r, то есть такая кнопка задаётся множеством разрешаемых ею наблюдений $r \cup \{r\}$[2]. **Q**-кнопке соответствует только множество $q \subseteq \mathbf{L}$ разрешаемых ею действий (говорят, что соответствующий отказ q ненаблюдаем). После нажатия кнопки допускается только одно наблюдение, разрешаемое этой кнопкой; чтобы получить следующее наблюдение, нужно нажать ту же или другую кнопку. Это правило не распространяется на τ-активность и разрушение. В LTS-модели реализации изображаются только переходы по действиям, τ- и γ-переходы. Переходы по отказам – виртуальные: отказ r, где $r \cup \{r\} \in \mathbf{R}$, наблюдается в стабильном

[2] В наших предыдущих работах кнопка задавалась только множеством r разрешаемых ею действий с дополнительным указанием, что $r \in \mathbf{R}$.

состоянии, в котором нет переходов по действиям из r; можно считать, что в этом состоянии имеется виртуальный переход-петля по отказу r.

В **R/Q**-семантике реализация задавалась LTS в алфавите $\mathbf{L}\cup\{\gamma\}$. Определялось отношение ***safe in*** безопасности кнопок после трасс реализации. Кнопка безопасна после трассы, если она *неразрушающая*: в реализации трасса не заканчивается в дивергентном состоянии или в состоянии, где есть переход по действию, разрешаемому этой кнопкой, ведущий в состояние, где есть γ-переход. Для **R**-кнопки этого достаточно для её безопасности по ***safe in***, а для **Q**-кнопки дополнительно требуется, чтобы трасса не заканчивалась в стабильном состоянии, где нет переходов по действиям из этой кнопки.

Спецификация задавалась LTS в алфавите $\mathbf{L}\cup\{\gamma\}$ и, кроме того, отношением ***safe by***, которое определяло кнопки, безопасные после трасс спецификации. Это отношение должно соблюдать три правила. 1) Безопасные по ***safe by*** кнопки неразрушающие. 2) Если после трассы некоторое действие разрешается неразрушающей кнопкой, то оно разрешается и некоторой (не обязательно той же самой) безопасной по ***safe by*** кнопкой. 3) Если **Q**-кнопка безопасна по ***safe by*** после трассы, то трасса продолжается в спецификации каким-нибудь действием, разрешаемым этой кнопкой. Безопасность кнопок определяет безопасные трассы спецификации как трассы, в которых каждое наблюдение разрешается некоторой кнопкой, безопасной после предшествующего этому наблюдению префиксу трассы.

Гипотеза о безопасности в **R/Q**-семантике требовала: 1) разрушение возможно в реализации с самого начала (до нажатия кнопок), если это имеет место в спецификации; 2) после общей безопасной трассы спецификации и реализации кнопка, безопасная по ***safe by*** в спецификации, должна быть безопасна по ***safe in*** в реализации. Класс безопасных реализаций, определяемый этой гипотезой о безопасности, обозначим $\boldsymbol{SafeImpl}_{\mathbf{R/Q}}(s)$.

Отношение конформности ***saco*** определялось для реализаций, удовлетворяющих такой гипотезе о безопасности, и требовало: после общей безопасной трассы спецификации и реализации любое наблюдение в реализации, которое разрешается кнопкой, безопасной после этой трассы по ***safe by*** в спецификации, должно быть после этой трассы и в спецификации. Класс безопасных и конформных реализаций, определяемый отношением ***saco***, обозначим $\boldsymbol{ConfImpl}_{\mathbf{R/Q}}(s)$.

Для генерации стандартного полного набора тестов используются, так называемые, тестовые трассы спецификации. Тестовая трасса – это безопасная трасса спецификации, продолженная наблюдением, которое разрешается какой-нибудь кнопкой, безопасной в спецификации после этой трассы. Эти кнопки как раз и вставляются в трассу для получения теста: перед каждым **R**-отказом r вставляется кнопка $r\cup\{r\}$, а перед каждым действием z – какая-нибудь безопасная кнопка p, разрешающая z. Если тестовая трасса отсутствует в спецификации, назначается вердикт ***fail*** (в спецификации нет последнего наблюдения трассы).

Чтобы представить **R/Q**-семантику как частный случай **B/O**-семантики и конформность ***saco*** как частный случай общей редукции, прежде всего, будем предполагать, что все её кнопки и наблюдения являются кнопками и наблюдениями общей машины тестирования: $\mathbf{R} \cup \mathbf{Q} \subseteq \mathbf{B}$ и $\mathbf{L} \cup \mathbf{R} \subseteq \mathbf{O}$.

Сначала выполним следующее преобразование LTS-реализаций:

1) Добавляем виртуальные петли по отказам (в стабильных состояниях).
2) Для каждой кнопки $p \in \mathbf{R} \cup \mathbf{Q}$ и каждого состояния a добавляем новое состояние a_p и новый переход $a \text{—} p \rightarrow a_p$.
3) В каждом новом состоянии a_p проводим переход $a_p \text{—} x \rightarrow b$ тогда и только тогда, когда $x \in \boldsymbol{obs}(p)$ и имеется переход $a \text{—} x \rightarrow b$.
4) В каждом новом состоянии a_p проводим переход $a_p \text{—} \tau \rightarrow b_p$ тогда и только тогда, когда имеется переход $a \text{—} \tau \rightarrow b$, и проводим переход $a_p \text{—} \gamma \rightarrow b_p$ тогда и только тогда, когда имеется переход $a \text{—} \gamma \rightarrow b$.
5) После этого удаляем все переходы по наблюдениям из старых состояний, оставляя τ- и γ-переходы.

У нас получится LTS, состояния которой делятся на «старые» и «новые», переходы по кнопкам ведут из старых состояний в новые (из a в a_p), переходы по наблюдениям – из новых состояний в старые (из a_p в b), причем только по тем наблюдениям, которые разрешаются кнопкой, которой помечен переход в это новое состояние. Переход по отказу r ведёт из состояния a_r в состояние a. τ- и γ-переходы ведут как из старых состояний в старые (из a в b), так и из соответствующих новых в соответствующие новые (из a_p в b_p).

Преобразованная **R/Q**-спецификация – это множество ошибок, состоящее из всех трасс тестов стандартного полного набора тестов, которым назначен вердикт ***fail***.

После этих преобразований конформность рассматривается не на классе всех реализаций, а на классе преобразованных **R/Q**-реализаций. Из-за этого возникают множественное следование ошибок и эквивалентные множества ошибок, не совпадающие со спецификацией, то есть возникают различные эквивалентные спецификации, которые остаются различными даже после нормализации. Все такие спецификации получаются в результате аналогичного преобразования из любого полного набора **R/Q**-тестов, если взять все ***fail***-трассы тестов набора как ошибки. Или, иными словами, существуют полные наборы тестов, различие между которыми не устраняется тривиальной оптимизацией, аналогичной нормализации спецификаций. В этом и состоит проблема оптимизации тестов для **R/Q**-семантики.

Тем не менее, эта проблема сводится к проблеме эквивалентности спецификаций на том или ином классе реализаций. Любую новую спецификацию с **λγ**-гипотезой можно использовать вместо старой

спецификации на классе реализаций, определяемом **R/Q**-гипотезой о безопасности. Если результаты преобразований реализации i и спецификации s обозначить i^* и s^*, соответственно, то:

$$(\boldsymbol{SafeImpl}_{\mathbf{R/Q}}(s))^* \subseteq \boldsymbol{SafeImpl}_{\lambda\gamma}(s^*) \text{ и}$$
$$(\boldsymbol{ConfImpl}_{\mathbf{R/Q}}(s))^* = (\boldsymbol{SafeImpl}_{\mathbf{R/Q}}(s))^* \cap C_{s^*}.$$

Любую безопасную для данной спецификации трассу, которую мы можем получить на **R/Q**-машине с данной реализацией i, мы можем получить на **B/O**-машине с преобразованной реализацией i^*. Нажатие кнопки точно такое же, потом ждём наблюдения, а потом снова нажимаем кнопку и т.д. Заметим, что нажатие «лишней» кнопки из $\mathbf{B} \setminus (\mathbf{R} \cup \mathbf{Q})$ преобразованной реализацией просто игнорируется, поскольку в каждом её состоянии нет перехода по такой кнопке, что трактуется как наличие перехода-петли по кнопке. «Лишние» наблюдения из $\mathbf{O} \setminus (\mathbf{L} \cup \mathbf{R})$ отсутствуют в преобразованной реализации и поэтому не будут появляться на экране дисплея.

5.2. R/Q-семантика с приоритетами

В **R/Q**-семантике τ- и γ-действия всегда разрешены, а нажатие кнопки p разрешает реализации, кроме того, выполнять любое действие из множества действий p. Если реализация может выполнить любое действие, разрешаемое нажатой кнопкой и определённое в её текущем состоянии, то говорят, что система не имеет приоритетов. В этом случае для того, чтобы реализация могла выполнить действие z, определённое в её текущем состоянии, можно нажимать любую кнопку p такую, что $z \in p \cup \{\tau,\gamma\}$ (если $z=\tau$ или $z=\gamma$, то такое действие может выполняться и в том случае, когда никакая кнопка не нажата). Система с приоритетами – это такая система, в которой выполнимость действия, определённого в текущем состоянии реализации, зависит от множества разрешённых внешних действий (если никакая кнопка не нажата, это множество пусто). При этом внешнее действие должно принадлежать «кнопочному» множеству p, а τ- и γ-действия разрешены при нажатии любой кнопки, а также когда никакая кнопка не нажата.

Системы с приоритетами для **R/Q**-семантики были введены в наших работах [[3],[4]]. В LTS-модели такой системы переход помечается не только действием z из $\mathbf{L} \cup \{\tau,\gamma\}$, но и предикатом π от множества разрешённых внешних действий, то есть парой (z,π). Если разрешено множество внешних действий p (нажата **R**-кнопка $p \cup \{p\}$, **Q**-кнопка p или $p=\varnothing$ и никакая кнопка не нажата), то такой переход может выполняться только в том случае, когда $z \in p \cup \{\tau,\gamma\}$ и $\pi(p)=\boldsymbol{true}$. Такой предикат можно понимать как булевскую функцию от булевских переменных $z_1,z_2,\ldots$, взаимно-однозначно соответствующих внешним действиям из алфавита **L**: переменная z_i принимает значение ***true***, если $z_i \in p$. Эта булевская функция может быть представлена в виде совершенной дизъюнктивной нормальной формы, дизъюнкты которой соответствуют множествам разрешённых внешних

действий $p_1,p_2,\ldots$ Поэтому переход по паре (z,π) эквивалентен множеству кратных переходов вида (z,p_j); переход (z,p_j) выполняется, если $z\in p_j\cup\{\tau,\gamma\}$ и разрешено множество внешних действий p_j.

При наличии приоритетов меняются понятия отказа и дивергенции. Отказ p возникает при нажатии кнопки p в том случае, когда в текущем состоянии нет переходов, помеченных парой вида (z,p), где $z\in p\cup\{\tau,\gamma\}$. Дивергенция 1) при нажатой кнопке p, или 2) когда никакая кнопка не нажата, возникает, когда в текущем состоянии начинается бесконечная цепочка переходов, помеченных, в первом случае, парой (τ,p) или, во втором случае, парой $(\tau,\varnothing)$. Соответственно, будем говорить о p-дивергенции и о p-дивергентных и p-конвергентных состояниях.

Рассмотрим несколько характерных примеров использования приоритетов.

Выход из дивергенции. Запрос, поступающий извне, может бесконечно долго игнорироваться системой, если он имеет тот же приоритет, что бесконечная внутренняя активность, то есть дивергенция. Заметим, что внутренняя активность может быть инициирована предыдущим запросом. Если речь идёт о составной системе, собранной из нескольких компонентов, то дивергенция может быть естественным результатом взаимодействия компонентов между собой. И в этом случае для обработки запроса, поступающего в систему (в один из её компонентов) извне, он должен иметь больший приоритет, чем внутреннее взаимодействие. Моделирование в **R/Q**-семантике. Переход по внешнему действию имеет тождественно истинный предикат, а τ-переход имеет предикат π, истинный только на пустом подмножестве алфавита внешних действий: $\pi(U)=(U=\varnothing)$.

Выход из осцилляции (приоритет приёма над выдачей). Под осцилляцией понимается бесконечная цепочка выдачи сообщений системой. Для того чтобы такую цепочку можно было прервать, заставив систему обрабатывать поступающий извне запрос, последний должен иметь больший приоритет, чем выдача сообщений. Моделирование в **R/Q**-семантике. Переход по запросу имеет тождественно истинный предикат, а переход по выдаче сообщения имеет предикат π, истинный на любом подмножестве действий, не содержащем запросов: $\pi(U)=(\forall ?x\ ?x\notin U)$, где префиксный знак «?» означает запрос. Обычно также подразумевается, что внутренняя активность менее приоритетна, чем приём запроса, то есть τ-переход имеет такой же предикат, как переход по выдаче сообщения.

Приоритет выдачи над приёмом в неограниченных очередях. Этот обратный пример характерен для неограниченной очереди, используемой в качестве буфера между взаимодействующими системами, в частности, при тестировании в контексте [[11]]. Здесь нужно, чтобы выборка из очереди была приоритетней постановки в очередь. В противном случае очередь имеет право только принимать сообщения и никогда их не выдавать. При тестировании в контексте для входной очереди это означает, что все входные сообщения,

посылаемые тестом, не доходят до реализации, бесконечно накапливаясь в очереди. Соответственно, для выходной очереди это означает, что тест может не получать никаких ответных сообщений от реализации, хотя она их выдаёт, поскольку они «оседают» в очереди. Моделирование в **R/Q**-семантике. Переход по выдаче имеет тождественно истинный предикат, а переход по приёму имеет предикат π, истинный на любом подмножестве действий, не содержащем выдачу: $\pi(U)=(\forall !y\, !y \notin U)$, где префиксный знак «!» означает выдачу. Обычно также подразумевается, что внутренняя активность менее приоритетна, чем выдача, то есть τ-переход имеет такой же предикат, как переход по приёму.

Прерывание цепочки действий. Команда «отменить» (cancel) должна останавливать выполнение последовательности действий, инициированной предыдущим запросом, и вызывать цепочку завершающих действий. При отсутствии приоритетов такая команда, даже если она выдана сразу после выдачи запроса, имеет право быть выполнена только после того, как вся обработка закончится, то есть, фактически, ничего «не отменяет». Моделирование в **R/Q**-семантике. Переход по команде «отменить» (cancel) имеет тождественно истинный предикат, а все остальные переходы имеют предикат π, истинный на любом подмножестве действий, не содержащем "cancel": $\pi(U)=(\text{cancel} \notin U)$.

Приоритетная обработка запросов. Если в систему поступает одновременно несколько запросов, то часто требуется их обработка в соответствии с некоторыми приоритетами между ними. Это реализуется в виде очереди запросов с приоритетами или в виде нескольких очередей запросов с приоритетами между очередями. К этому типу приоритетов относится и обработка аппаратных прерываний в операционной системе. Моделирование в **R/Q**-семантике. Множество запросов разбивается на непересекающиеся подмножества $X_1, X_2, \ldots$ так, что запросы из подмножества с большим индексом имеют больший приоритет. Предикат π_i на переходе по запросу из X_i истинен на любом подмножестве действий, не содержащем запросы из подмножества с большим номером: $\pi_i(U)=(\forall j>i\ U \cap X_j=\varnothing)$.

Как было сказано выше, в **B/O**-семантике имеется приоритет тестового воздействия над поведением реализации, как наблюдаемым, так и ненаблюдаемым. Это даёт возможность представить **R/Q**-семантику с приоритетами как частный случай **B/O**-семантики и конформность ***saco*** с приоритетами как частный случай общей редукции. Как и для **R/Q**-семантики без приоритетов будем предполагать, что все её кнопки и наблюдения являются кнопками и наблюдениями общей машины тестирования: $\mathbf{R} \cup \mathbf{Q} \subseteq \mathbf{B}$ и $\mathbf{L} \cup \mathbf{R} \subseteq \mathbf{O}$.

Сначала выполним следующее преобразование LTS-реализаций:

1) Добавляем виртуальные петли по отказам: если в состоянии a нет переходов вида a—(z,p)→b, где z∈p∪{τ,γ}, добавляем переход a—(p,p)→a.
2) Для каждого подмножества действий p⊆**L** и каждого состояния a добавляем новое состояние a_p и новый переход a—p→a_p.
3) В каждом новом состоянии a_p проводим переход a_p—x→b тогда и только тогда, когда x∈***obs***(p) и имеется переход a—(x,p)→b.
4) В каждом новом состоянии a_p проводим переход a_p—τ→b_p тогда и только тогда, когда имеется переход a—(τ,p)→b, и проводим переход a_p—γ→b_p тогда и только тогда, когда имеется переход a—(γ,p)→b.
5) В каждом новом состоянии a_p для каждого подмножества q⊆**L** проводим переход a_p—q→a_q.
6) В каждом старом состоянии a переход a—(τ,∅)→b заменяется переходом a—τ→b, а переход a—(γ,∅)→b заменяется переходом a—γ→b. Все остальные переходы из состояния a удаляются.

У нас получится LTS, состояния которой делятся на «старые» и «новые». Переход по кнопке p ведёт из старого состояния в новое состояние: a—p→a_p, а также из нового состояния в новое состояние: a_p—q→a_q. Переход по x, где x наблюдение, ведёт из нового состояния в старое: a_p—x→b, причем только в том случае, когда был переход a—(x,p)→b. Если x – это отказ, то p=x и b=a. τ- и γ-переход ведёт либо из нового состояния в новое состояние: a_p—τ/γ→b_p, если был переход a—(τ/γ,p)→b, либо из старого состояния в старое состояние: a—τ/γ→b, если был переход a—(τ/γ,∅)→b.

Преобразование спецификации, конформность, класс тестируемых реализаций и тестирование через машину тестирования определяются аналогично случаю отсутствия приоритетов.

Отметим одну особенность такого моделирования **R/Q**-семантики с приоритетами средствами **B/O**-семантики. В **R/Q**-семантике, содержащей пустую кнопку (∅∈**R**∪**Q**), не различаются τ- и γ-переходы при нажатии пустой кнопки и при отсутствии нажатой кнопки: в обоих случаях множество разрешённых внешних действий одно и то же – пустое множество. Из-за этого невозможно потребовать, чтобы такой переход срабатывал только в том случае, когда никакая кнопка не нажата, или, наоборот, чтобы он срабатывал при нажатии пустой кнопки, но не мог выполняться, если никакая кнопка не

нажата. При моделировании в **B/O**-семантике τ-переходы $a_{\varnothing}—\tau\rightarrow b_{\varnothing}$ и $a—\tau\rightarrow b$ возникают всегда одновременно (если был переход $a—(\tau,\varnothing)\rightarrow b$). Но после такого моделирования в **B/O**-семантике мы можем разрешить эту проблему, оставив только один из этих двух τ-переходов.

5.3. Ready-trace семантика

Ready-trace семантика [[9]] отличается от **R/Q**-семантики тем, что при возникновении **R**-отказа наблюдается не **R**-отказ, а множество действий, которые реализация может выполнить в данном стабильном состоянии. Это множество называется множеством готовности (***ready set***). Понятно, что если после нажатия **R**-кнопки p наблюдается множество готовности r, то $p\cap r=\varnothing$. Тем самым, наблюдаться может, вообще говоря, не любое множество готовности, а только такое, которое имеют непустое пересечение хотя бы с одной **R**-кнопкой. После наблюдения множества готовности r нажатие любой кнопки, имеющей с r пустое пересечение, вызовет отказ в том же состоянии. Поэтому имеет смысл нажимать только такие кнопки, которые имеют с r непустое пересечение. После наблюдения действия нажатие **R**-кнопки может вызвать наблюдение действия или множества готовности, а нажатие **Q**-кнопки – наблюдение действия или ненаблюдаемый deadlock.

Таким образом, множества готовности в ready-trace семантике занимают место **R**-отказов в **R/Q**-семантике. Трасса готовности – это последовательность действий и множеств готовности, причем после множества готовности может быть только действие, принадлежащее этому множеству. Для получения трасс готовности в каждом стабильном состоянии добавляем переход-петлю по множеству готовности, если это множество имеет пустое пересечение с какой-нибудь **R**-кнопкой. Далее, как обычно, рассматриваются маршруты и в качестве трасс берутся пометки на их переходах с пропуском символа τ.

Безопасность кнопок в реализации (***safe in***) и спецификации (***safe by***) определяется так же, как в **R/Q**-семантике, но только после трасс готовности. Аналогично определяются безопасные трассы реализации и спецификации, а также гипотеза о безопасности. Спецификация указывает, какие множества готовности и какие действия могут наблюдаться после тех или иных трасс готовности. Отношение конформности ***resaco*** требует: после общей безопасной трассы готовности спецификации и реализации любое наблюдение (действие или множество готовности), которое возможно в реализации после нажатия кнопки, безопасной после этой трассы по ***safe by*** в спецификации, должно быть после этой трассы и в спецификации. Аналогично определяется и генерация полного набора тестов: в трассы готовности вставляются кнопки: перед действием вставляется безопасная **R**- или **Q**-кнопка, разрешающая это действие, а перед множеством готовности – безопасная **R**-кнопка, имеющая пустое пересечение с этим множеством готовности.

Чтобы представить ready-trace семантику как частный случай **B/O**-семантики и конформность ***resaco*** как частный случай общей редукции, прежде всего,

будем предполагать, что все её кнопки и наблюдения (действия и наблюдаемые множества готовности) являются кнопками и наблюдениями общей машины тестирования: $\mathbf{R} \cup \mathbf{Q} \subseteq \mathbf{B}$ и $\mathbf{L} \cup \{r \subseteq \mathbf{L} | \exists p \in \mathbf{R}\ r \cap p = \varnothing\} \subseteq \mathbf{O}$.

Преобразование LTS-реализаций для ready-trace семантики аналогично преобразованию для **R/Q**-семантики, но только вместо виртуальных петель по **R**-отказам проводятся виртуальные петли для наблюдаемых множеств готовности:

1) Для каждой кнопки $p \in \mathbf{R} \cup \mathbf{Q}$ и каждого состояния a добавляем новое состояние a_p и новый переход $a \text{—} p \rightarrow a_p$.
2) В каждом новом состоянии a_p проводим переход $a_p \text{—} x \rightarrow b$ тогда и только тогда, когда $x \in p$ и имеется переход $a \text{—} x \rightarrow b$.
3) В каждом новом состоянии a_p проводим переход $a_p \text{—} r \rightarrow a$ тогда и только тогда, когда состояние a стабильно, соответствующее ему множество готовности равно r и $r \cap p = \varnothing$.
4) В каждом новом состоянии a_p проводим переход $a_p \text{—} \tau \rightarrow b_p$ тогда и только тогда, когда имеется переход $a \text{—} \tau \rightarrow b$, и проводим переход $a_p \text{—} \gamma \rightarrow b_p$ тогда и только тогда, когда имеется переход $a \text{—} \gamma \rightarrow b$.
5) После этого удаляем все переходы по внешним действиям из старых состояний, оставляя τ- и γ-переходы.

У нас получится LTS, состояния которой делятся на «старые» и «новые», переходы по кнопкам ведут из старых состояний в новые (из a в a_p), переходы по действиям – из новых состояний в старые (из a_p в b), причем только по тем действиям x, которые разрешаются кнопкой p ($x \in p$), которой помечен переход в это новое состояние a_p. Переход по множеству готовности r ведёт из состояния a_r в состояние a. А τ- и γ-переходы ведут как из старых состояний в старые (из a в b), так и из соответствующих новых в соответствующие новые (из a_p в b_p).

Преобразование спецификации, конформность, класс тестируемых реализаций и тестирование через машину тестирования определяются аналогично **R/Q**-семантике.

6. Оптимизация тестов для различных классов реализаций

В предыдущих разделах мы показали, что для **B/O**-семантики и общей редукции на классе всех возможных реализаций имеются только тривиальные зависимости между ошибками, которые легко устраняются процедурой нормализации спецификации. Также мы рассмотрели две гипотезы о безопасности: λ- и γ-гипотезу, которые сужают класс тестируемых реализаций. При этом λ-гипотеза создаёт дополнительную зависимость между ошибками, которая, однако, легко устраняется дополнительной λ-нормализацией, а γ-гипотеза не создаёт дополнительной зависимости между ошибками и дополнительной нормализации не требуется. Далее мы рассмотрели примеры семантик и конформностей, которые сводятся к **B/O**-семантике и общей редукции, но рассматриваются на суженных классах

реализаций. Из-за такого сужения возникают уже нетривиальные зависимости между ошибками, что требует нетривиальной оптимизации тестов [[6],[7]].

Всё это можно рассматривать как частный случай общей проблемы сужения класса тестируемых реализаций, которое создаёт зависимости между ошибками и даёт возможность оптимизации тестов. Рассмотрим несколько примеров такого сужения класса тестируемых реализаций, не связанных напрямую с выбором той или иной семантики или гипотезы о безопасности. В этих примерах мы покажем, что такое сужение позволяет использовать конечные полные наборы тестов. Во всех этих примерах предполагается конечность **B**/**O**-семантики и LTS-спецификации s. Будем считать, что суммарное число кнопок и наблюдений не превосходит числа m, а число состояний детерминированной LTS-спецификации не превосходит числа k.

Первый пример – класс LTS-реализаций с ограниченным числом состояний. Если число состояний реализации не превосходит n, то для полноты тестирования достаточно ограничиться тестами длиной не более nk. Тогда этот набор тестов содержит не более $O(m^{nk})$ тестов.

Для доказательства этого утверждения достаточно построить композицию LTS реализации и спецификации по следующим правилам. Состояниями композиционной LTS являются пары состояний реализации и спецификации, начальное состояние – пара начальных состояний. Переход (a,b)—x→(a`,b`) определяется тогда и только тогда, когда в реализации есть переход a—x→a`, а в спецификации есть переход b—x→b`. Реализация неконформна тогда и только тогда, когда в ней есть ошибка 1-го рода, то есть трасса спецификации. Такая трасса в детерминированной спецификации заканчивается в одном состоянии, которое объявлено конечным. В реализации есть такая трасса тогда и только тогда, когда в композиционной LTS из начального состояния достижимо состояние вида (a,b), где b – конечное состояние спецификации. Такое состояние достижимо по простому маршруту (проходящему через каждое состояние не более одного раза), длина которого не превосходит числа достижимых состояний композиционной LTS, которое, в свою очередь, не превосходит общего числа состояний, равного nk. Таким образом, реализация неконформна тогда и только тогда, когда в ней есть ошибочная трасса длиной не более nk. Иными словами, набор всех примитивных тестов длиной не более nk, является полным. Число таких последовательностей в алфавите с m символами, очевидно, равно $O(m^{nk})$.

Второй пример – конечный (с точностью до изоморфизма) класс тестируемых реализаций. Для конечной семантики класс LTS-реализаций с ограниченным числом состояний, очевидно, конечен с точностью до изоморфизма. Поэтому первый пример является частным случаем второго примера. Если семантика и спецификация конечны, то для любого конечного класса реализаций существует конечный полный набор тестов. Для доказательства достаточно заметить, что любой конечный класс реализаций I является подклассом класса

реализаций, число состояний которых ограничено числом n, где n – максимальное число состояний реализаций из класса I.

Третий пример – конечный подкласс неконформных реализаций класса тестируемых реализаций. В работах [[14],[15]] такой подкласс называется *классом неисправностей*. Для конечности полноты тестового набора достаточно не конечности класса I тестируемых реализаций, а его подкласса $I\backslash C_s$ неисправностей. Действительно, в каждой неконформной реализации из $i\in I\backslash C_s$ имеется некоторая ошибка 1-го рода, выберем одну такую ошибку σ_i. Набор ошибок $s_I=\{\sigma_i|i\in I\backslash C_s\}$ конечен и, очевидно, является полным тестом, а набор $\{\{\sigma_i\}|i\in I\backslash C_s\}$ примитивных тестов, построенный по этим ошибкам, является полным набором тестов для класса I.

Таким образом, фактически, при тестировании мы пытаемся найти не все ошибки 1-го рода, определяемые спецификацией s, а их конечное подмножество $s_I\subseteq s$. Это эквивалентно тому, что вместо спецификации s мы используем спецификацию s_I. Иными словами, на классе реализаций I спецификации s и s_I эквивалентны. Правда, выполняя тестирование по спецификации s, мы можем быстрее найти ошибку, чем при тестировании по спецификации s_I. Это объясняется тем, что неконформная реализация $i\in I\backslash C_s$ может содержать не только ошибку σ_i, но и какие-то ошибки, не вошедшие в конечный набор s_I. Например, спецификация s может определять как ошибочные некоторые наблюдения с самого начала (до нажатия кнопок): $a,b_1,b_2,b_3,\ldots$, а s_I содержит только одну такую ошибку a. При тестировании мы можем с самого начала ждать наблюдений и, опираясь на спецификацию s, выносим вердикт ***fail***, если получаем любую ошибку $a,b_1,b_2,b_3,\ldots$, но, опираясь на спецификацию s_I, вердикт ***fail*** выносится только для ошибки a.

Эти рассуждения дают четвёртый пример – конечное подмножество ошибок $s_I\subseteq s$ такое, что каждая неконформная (то есть содержащая хотя бы одну ошибку из s) реализация из класса I содержит хотя бы одну ошибку из s_I. Вместо конечного класса неисправностей достаточно использовать просто конечный поднабор набора ошибок, определяемого спецификацией.

Далее напомним, что класс реализаций I определяет ошибки 2-го рода: трассы, которые не встречаются в конформных реализациях класса I, но встречаются в некоторых его неконформных реализациях. Такая ошибка 2-го рода σ может не быть ошибкой 1-го рода, то есть $\sigma\notin s$. Поэтому четвёртый пример является частным случаем последнего, пятого примера, когда для класса тестируемых реализаций I задан конечный набор ошибок (1-го и 2-го родов) s_I такой, что каждая неконформная (то есть содержащая хотя бы одну ошибку из s) реализация из класса I содержит хотя бы одну ошибку из s_I. Этот пример последний, поскольку его условие просто эквивалентно условию существования конечного полного набора тестов. Если такой набор тестов существует, то множество трасс всех тестов набора – это и есть множество ошибок s_I.

7. Обоснование выбранной модели взаимодействия

В этом заключительном разделе статьи мы дадим обоснование выбранной нами модели взаимодействия. Мы рассмотрим шесть вопросов, которые возникают в связи с этой моделью: 1) когда кнопка попадает в трассу: при её нажатии и/или при выполнении LTS-реализацией перехода по кнопке; 2) почему нажатие кнопки блокирует наблюдения; 3) почему оператор должен уметь нажать кнопку достаточно быстро после получения трассы; 4) почему нажатие кнопки не блокирует τ-активность; 5) почему нажатие кнопки блокирует дивергенцию, то есть разрешает только конечную τ-активность; 6) почему переход по каждой кнопке определён в каждом состоянии реализации?

7.1. Когда кнопка попадает в трассу?

Нажатие кнопки выполняется оператором машины тестирования, и в этот момент времени он знает, какая трасса уже получена. Поэтому ничто не мешает оператору отметить тот факт, что он нажал данную кнопку после наблюдения данной трассы. С другой стороны, поведение реализации, вообще говоря, зависит от той трассы, после которой оператор нажимает кнопку. Поэтому в любом случае при нажатии кнопки она попадает в трассу.

Если при выполнении LTS-реализацией перехода по кнопке p на экране дисплея также появляется кнопка p, то это аналогично тому, как при выполнении реализацией перехода по наблюдению на экране дисплея появляется это наблюдение. Это означает, что выполнение перехода по кнопке p есть, фактически, наблюдение, появление которого в трассе обозначим p`, чтобы отличить от p, означающего нажатие кнопки p.

Такое наблюдение, в принципе, ничем не отличается от других наблюдений, поэтому режим работы с наблюдаемым переходом по кнопке можно считать частным случаем общей модели взаимодействия. Чтобы в этой модели изобразить такой режим работы, достаточно в LTS-реализации каждый переход по кнопке a—p→b заменить на два перехода с введением дополнительного промежуточного состояния: a—p→a`—p`→b.

7.2. Почему нажатие кнопки блокирует наблюдения?

Если нажатие кнопки не блокирует наблюдения, то появляется дополнительная зависимость между трассами реализации (и, следовательно, между ошибками). Поясним это на примере. Пусть при взаимодействии с реализацией может наблюдаться трасса up, где u наблюдение, а p кнопка. Тогда, поскольку наблюдается трасса up, то наблюдается и её префикс – трасса u. Если оператор нажимает кнопку p до наблюдения u, но наблюдения не блокируются этим нажатием, то реализация всё равно может выполнить переход по u. Тем самым, будет наблюдаться трасса pu. Следовательно, если при взаимодействии с реализацией может наблюдаться трасса up, то может наблюдаться и трасса pu.

В выбранной нами модели взаимодействия такой дополнительной зависимости между трассами нет. В то же время режим работы с отсутствием блокировки наблюдений при нажатии кнопки легко моделируется в нашей модели. Для этого достаточно систематически выполнить следующее преобразование реализации, пока оно возможно: если в реализации имеются переходы $a—p→a_p$, $a—u→b$ и $b—p→b_p$, то добавим переход $a_p—u→b_p$. Тем самым, если в состоянии a начиналась трасса up, заканчивающаяся в состоянии b_p, то теперь будет и трасса pu, заканчивающаяся в том же состоянии.

Таким образом, модель взаимодействия с блокировкой наблюдений при нажатии кнопки является более общей. Классу всех реализаций для модели без блокировки соответствует подкласс реализаций, получаемых описанной выше процедурой, для модели с блокировкой. Как и в общем случае, такое сужение класса реализаций приводит к появлению зависимостей между трассами реализации (в частности, между ошибками).

Кроме этого, блокировка наблюдений является следствием приоритета тестового воздействия над наблюдениями. Такой приоритет необходим для того, чтобы можно было моделировать поведение систем с приоритетами. В частности, для прерывания цепочки внешних действий командой «отменить» (cancel).

7.3. Зачем оператору нужно быстро нажимать кнопки?

Прежде всего, отметим, что мы исходим из основного допущения о τ-активности: в реализации τ-активность может быть перед и после любого наблюдения, а также перед и после любого перехода по кнопке. Понятно, что любые ограничения на τ-активность только сузили бы класс рассматриваемых реализаций, что могло бы привести к появлению дополнительных зависимостей между ошибками. Наличие τ-активности ещё не означает, что она обязательно будет выполняться, но, конечно, предполагается, что хотя бы при некотором взаимодействии она выполняется. Естественно, что τ-активность может выполняться, когда никакая кнопка не нажата. Блокирует ли нажатие кнопки τ-активность или нет, мы рассмотрим в следующем пункте.

Также мы хотим, чтобы любой достижимый переход в LTS-реализации мог быть выполнен при том или ином взаимодействии с ней (в зависимости от поведения оператора и погодных условий, моделирующих недетерминированное поведение реализации). Если это не так, и какой-то переход не выполняется ни при каком взаимодействии, то это эквивалентно отсутствию этого перехода в реализации. А это, в свою очередь, приводит к сужению класса рассматриваемых реализаций, что также чревато появлением дополнительных зависимостей между ошибками. Только гипотезы о безопасности запрещают выполнение тех или иных «опасных» переходов в реализации, но, как мы рассмотрели выше, это также приводит к сужению

класса реализаций и, возможно, появлению дополнительных зависимостей между ошибками.

Почему мы требуем, чтобы оператор мог нажимать кнопку достаточно быстро после получения трассы, хотя и не обязан это делать всегда? Если оператор не может нажать кнопку достаточно быстро после трассы, то реализация может успеть выполнить после этой трассы один или несколько τ-переходов. Тем самым, переход по кнопке, начинающийся в состоянии до этих τ-переходов, никогда не будет выполнен.

7.4. Почему нажатие кнопки не блокирует τ-активность?

Здесь мы снова опираемся на требование выполнимости каждого достижимого перехода. Если нажатие кнопки блокирует τ-активность, то для того, чтобы реализация могла выполнить некоторую цепочку τ-переходов (а после неё переход по кнопке), оператор не должен нажимать кнопку до тех пор, пока эта цепочка не будет выполнена, и должен нажать кнопку сразу же после выполнения этой цепочки. Поскольку τ-активность ненаблюдаема, оператор должен просто выждать некоторый интервал времени, прежде чем нажать кнопку. Тем самым, к оператору предъявляются весьма нетривиальные требования по скорости его работы: после получения трассы он должен выдерживать паузу, прежде чем нажимать кнопку, причём длительность этой паузы должна быть, вообще говоря, произвольной в разных сеансах тестирования.

Вместо этого мы выбрали вариант, когда нажатие кнопки не блокирует τ-активность. Тогда к оператору предъявляется только одно требование, рассмотренное в предыдущем пункте: он должен уметь нажимать кнопку достаточно быстро после получения трассы, хотя и не обязан это делать всегда. У реализации появляется выбор: выполнять τ-переход или переход по нажатой кнопке. Как обычно, этот выбор недетерминирован и определяется погодными условиями.

7.5. Почему нажатие кнопки блокирует дивергенцию?

Хотя нажатие кнопки не блокирует τ-активность, но разрешает только конечную τ-активность, то есть разрешает выполнять только конечное число τ-переходов. Тем самым, нажатие кнопки блокирует дивергенцию. Это необходимо для того, чтобы реализовать «выход из дивергенции», то есть приоритет тестового воздействия над дивергенцией.

7.6. Почему переход по каждой кнопке определён в каждом состоянии реализации?

До сих пор мы предполагали, что переход по кнопке определён в каждом состоянии LTS-реализации (по умолчанию отсутствие такого перехода в состоянии трактуется как наличие перехода-петли). На самом деле, это

требование не столь принципиально, если его опустить, то это влияет лишь на условие выполнения τ-активности при нажатой кнопке. Новое условие такое: реализация может не выполнить никакого перехода по нажатой кнопке p только в том случае, если она после конечного числа τ-переходов будет бесконечно двигаться по бесконечному τ-маршруту, который проходит только через такие состояния, где нет переходов по кнопке p. В противном случае реализация выполняет конечное число τ-переходов и затем переход по кнопке p.

LTS-реализацию, в которой переходы по кнопкам определены не во всех состояниях, можно промоделировать с помощью LTS, в которой такие переходы есть во всех состояниях. Для этого в исходную LTS-реализацию вносятся следующие изменения.

1. Если переход по кнопке p отсутствует в стабильном состоянии a, то возникает deadlock: реализация не может выполнить переход по p или τ-переход, поскольку их нет, и не может выполнить переход по наблюдению, поскольку такие переходы блокируются нажатой кнопкой p, а разблокированы могут быть только после перехода по p. Внешне (для оператора машины тестирования) такой deadlock выглядит как отсутствие наблюдений. Из такого deadlock`а можно выйти, нажав другую кнопку, по которой в стабильном состоянии a есть переход. В нашей модели это реализуется добавлением перехода a—p→a`, ведущего в новое состояние a`, в котором определяются переходы-петли a`—q→a` по всем кнопкам q, по которым из состояния a нет переходов, а также переходы по кнопкам, по которым есть переходы из состояния a, ведущие туда же, куда они ведут из состояния a: переход a_p—r→b проводится, когда есть переход a—r→b.
2. Если переход по кнопке p отсутствует в нестабильном состоянии a, в котором не начинается бесконечный τ-маршрут, проходящий только конечное число раз через состояния, где есть переход по кнопке p, то через конечное число τ-переходов будет выполнен какой-нибудь переход по p. Нам достаточно добавить любой переход a—p→b`, если из состояния a по τ-переходам достижимо состояние b и имеется (или добавлен в п.1) переход b—p→b`.
3. Если переход по кнопке p отсутствует в дивергентном, состоянии a, в котором начинается бесконечный τ-маршрут, проходящий только конечное число раз через состояния, где есть переход по кнопке p, то возможно, что реализация будет проходить именно этот бесконечный τ-маршрут. В этом случае может не выполниться никакой переход по p, и переходы по наблюдениям останутся заблокированы. Внешне (для оператора машины тестирования) такая дивергенция выглядит как

отсутствие наблюдений. Из неё можно выйти, нажав другую кнопку, для которой условие этого пункта не будет выполняться. Это полностью аналогично п.1, при моделировании в нашей модели выполняются те же изменения в реализации, которые описаны в п.1.

Литература

[1] Бурдонов И.Б., Косачев А.С., Кулямин В.В. Формализация тестового эксперимента. «Программирование», 2007, №5.

[2] Бурдонов И.Б., Косачев А.С., Кулямин В.В. Теория соответствия для систем с блокировками и разрушением. «Наука», 2008.

[3] Бурдонов И.Б., Косачев А.С. Системы с приоритетами: конформность, тестирование, композиция. Труды Института системного программирования РАН, № 14.1, 2008, стр.23-54

[4] Бурдонов И.Б., Косачев А.С. Системы с приоритетами: конформность, тестирование, композиция. "Программирование", 2009, №4.

[5] Игорь Бурдонов. Теория конформности (функциональное тестирование программных систем на основе формальных моделей). LAP LAMBERT Academic Publishing, Saarbrucken, Germany, 2011, ISBN 978-3-8454-1747-9.

[6] Бурдонов И.Б., Косачев А.С. Удаление из спецификации неконформных трасс. Препринт Института Системного Программирования РАН, 2011, №23.

[7] Бурдонов И.Б., Косачев А.С. Пополнение спецификации для ***ioco***. "Программирование", 2011, №1.

[8] Bernot G. Testing against formal specifications: A theoretical view. In S. Abramsky and T.S.E. Maibaum, editors, TAPSOFT'91, Volume 2, pp. 99-119. Lecture Notes in Computer Science 494, Springer-Verlag, 1991.

[9] van Glabbeek R.J. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, CONCUR'90, Lecture Notes in Computer Science 458, Springer-Verlag, 1990, pp 278–297.

[10] van Glabbeek R.J. The linear time - branching time spectrum II; the semantics of sequential processes with silent moves. Proceedings CONCUR '93, Hildesheim, Germany, August 1993 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66-81.

[11] Revised Working Draft on "Framework: Formal Methods in Conformance Testing". JTC1/SC21/WG1/Project 54/1 // ISO Interim Meeting / ITU-T on, Paris, 1995.

[12] Tretmans J. Conformance testing with labelled transition systems: implementation relations and test generation. Computer Networks and ISDN Systems, v.29 n.1, p.49-79, Dec. 1996.

[13] Tretmans J. Test Generation with Inputs, Outputs and Repetitive Quiescence. In: Software-Concepts and Tools, Vol. 17, Issue 3, 1996.

[14] Adenilso da Silva Simão, Alexandre Petrenko, Nina Yevtushenko: Generating Reduced Tests for FSMs with Extra States. TestCom/FATES 2009: 129-145.

[15] Alexandre Petrenko, Nina Yevtushenko: Testing from Partial Deterministic FSM Specifications. IEEE Trans. Computers 54(9): 1154-1165 (2005).

Error dependencies on classes of implementations under testing.

Igor Bourdonov <igor@ispras.ru>, Alexander Kossatchev kos@ispras.ru
ISP RAS, Moscow, Russia

Abstract .The paper discusses the problem of dependency between errors defined by specification and the related problem of test optimization. There is a dependency between errors if a strict subset of errors exists such that any nonconforming implementation (i.e. an implementation containing an error) contains an error from this subset. Accordingly, it is sufficient for the tests to detect errors only from this subset. The most general formal model of test interaction and the reduction type of conformance are suggested, for which dependency between errors is almost absent. Most of the known conformances in various interaction semantics are demonstrated to be special cases of this general model. In this general model, the dependency between errors may occur when any strict subset of the class of all implementations is chosen as a class of implementations under testing. Particular interaction semantics and/or various hypotheses on implementations (specifically, the safety hypothesis), in fact, assume that the implementation under testing should belong to some subclass of (safe) implementations.

Keywords: interaction semantics, traces, LTS, conformance, error dependency, test generation, divergence, destruction, safe testing

References

[1]. Bourdonov I., Kossatchev A., Kuliamin V. Formalization of Test Experiments. Programming and Computer Software, Vol. 33, No. 5, 2007, pp. 239-260.

[2]. Bourdonov I., Kossatchev A., Kuliamin V. Teoriya sootvetstviya dlya system s blokirovkami i razrusheniem [Conformance theory of the systems with Refused Inputs and Forbidden Actions]. Moscow, «Nauka», 2008. (in Russian)

[3]. Bourdonov I., Kossatchev A. Sistemy s prioritetami: konformnost', testirovanie, kompozitsiya [Systems with priority: conformance, testing, composition]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 14.1, 2008. (in Russian)

[4]. Bourdonov I., Kossatchev A. Sistemy s prioritetami: konformnost', testirovanie, kompozitsiya [Systems with priority: conformance, testing, composition]. Programming and Computer Software, Vol. 35, No. 6, 2009,pp. 301-313.

[5]. Bourdonov I. Teoriya konformnosti (funkcional'noe testirovanie prorammny'kh system na osnove formal'ny'kh modelej [Conformance theory (functional testing on formal model base)]. LAP LAMBERT Academic Publishing, Saarbrucken, Germany, 2011, ISBN 978-3-8454-1747-9, 428 pp. http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf) (in Russian

[6]. Bourdonov I., Kossatchev A. Udalenie iz spetsifikatsii nekonformnykh trass [Nonconforming traces elimination from specification]. Preprint № 23, ISP RAN [Preprints of the Institute for System Programming of RAS, Preprint 23], 2011. (in Russian).

[7]. Bourdonov I., Kossatchev A. Specification Completion for IOCO. Programming and Computer Software, Vol. 37, No. 1, 2011, pp. 1-14.

[8]. Bernot G. Testing against formal specifications: A theoretical view. In S. Abramsky and T.S.E. Maibaum, editors, TAPSOFT'91, Volume 2, pp. 99-119. Lecture Notes in Computer Science 494, Springer-Verlag, 1991.
[9]. van Glabbeek R.J. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, CONCUR'90, Lecture Notes in Computer Science 458, Springer-Verlag, 1990, pp 278–297.
[10]. van Glabbeek R.J. The linear time - branching time spectrum II; the semantics of sequential processes with silent moves. Proceedings CONCUR '93, Hildesheim, Germany, August 1993 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66-81.
[11]. Revised Working Draft on "Framework: Formal Methods in Conformance Testing". JTC1/SC21/WG1/Project 54/1 // ISO Interim Meeting / ITU-T on, Paris, 1995.
[12]. Tretmans J. Conformance testing with labelled transition systems: implementation relations and test generation. Computer Networks and ISDN Systems, v.29 n.1, p.49-79, Dec. 1996.
[13]. Tretmans J. Test Generation with Inputs, Outputs and Repetitive Quiescence. In: Software-Concepts and Tools, Vol. 17, Issue 3, 1996.
[14]. Adenilso da Silva Simão, Alexandre Petrenko, Nina Yevtushenko: Generating Reduced Tests for FSMs with Extra States. TestCom/FATES 2009: 129-145.
[15]. Alexandre Petrenko, Nina Yevtushenko: Testing from Partial Deterministic FSM Specifications. IEEE Trans. Computers 54(9): 1154-1165 (2005).

Комбинаторная генерация программных конфигураций ОС[1]

Кулямин В. В.
kuliamin@ispras.ru

Аннотация. В статье представлен метод генерации тестов для конфигурационного тестирования на основе покрывающих наборов, т.е., обеспечивающая покрытие всех возможных комбинаций пар, троек и т.д., значений параметров конфигурации. Новым элементом в предлагаемом методе является учет условий использования отдельных параметров, который вносит коррективы как в учет покрываемых комбинаций, так и в построение отдельных тестов. Данный метод использован на практике для генерации тестовых программных конфигураций операционной системы реального времени, приведены результаты этого применения.

Ключевые слова: конфигурационное тестирование; покрывающий набор; генерация тестов

1. Введение

Одной из важных задач контроля качества системного программного обеспечения (ПО) является проверка корректности его работы в разнообразных конфигурациях. Часть вариативности этих конфигураций может быть связана с используемым аппаратным обеспечением, другая часть — чисто программная и представляет собой набор возможных изменений в составе модулей ПО, в настройках используемых алгоритмов, в значениях констант, определяющих, например, емкость внутренних буферов или количество внешних каналов связи определенного типа. Тестирование корректности работы ПО в разных его конфигурациях называется обычно конфигурационным тестированием.

Основная проблема конфигурационного тестирования — необходимость выбора сравнительно малого числа конфигураций для тестирования, которое нужно провести в рамках определенных временных и бюджетных ограничений, из огромного набора всех возможных конфигураций, полная проверка которого просто физически невозможна. Количество параметров

[1]Работа поддержана ФЦП “Исследования и разработ-,ки по приоритетным направлениям развития научно-технологического комплекса России на 2007–2013 годы” (контракт № 11.519.11.4024).

конфигурации практически используемых систем достигает сотен и тысяч, многие из этих параметров могут иметь больше двух значений. Даже для сотни параметров с двумя значениями общее число возможных конфигураций равно $2^{100} = 1.26765\ldots\cdot10^{30}$, для реальных систем это число гораздо больше.

Дополнительные сложности в процесс выбора небольшого набора тестовых конфигураций из огромного множества возможных привносят разнообразные ограничения на корректные наборы значений параметров конфигурации: некоторые параметры могут принимать лишь подмножество возможных значений, если другие зафиксированы (подобные ограничения можно назвать *общими*), часть параметров вообще не используется и не влияет на работу системы, если определенный параметр имеет неподходящее значение (такого рода ограничения будем называть далее *условиями использования*). Например, неважно, куда именно выводится трасса, если вывод трассы отключен, а если он включен, трасса может быть направлена либо на терминал, либо в файл. Ограничения не позволяют использовать в тестовых конфигурациях произвольные значения параметров — они должны соответствовать всем налагаемым ограничениям, при этом выбор тестовых конфигураций усложняется, а сопутствующее уменьшение множества всех возможных конфигураций не упрощает задачи, поскольку его мощность остается слишком большой для любых переборных методов.

Основная цель конфигурационного тестирования в случае программных конфигураций — добиться выполнения различных частей кода системы, которые включаются в нее или исключаются в зависимости от значений конфигурационных параметров (очень часто эти значения задаются через макроопределения #define/#undef, а части кода включаются или исключаются за счет использования #ifdef). Каждый такой участок кода, условно включаемый в код системы, закрыт лишь небольшим количеством условий, обычно одним-двумя, реже используются 3-4. Для его включения достаточно использовать любую конфигурацию, где это небольшое число параметров, от которых он зависит, имеет заданные значения. Поэтому решение проблемы выбора тестовых конфигураций может быть получено за счет применения покрывающих наборов [1], обеспечивающих использование сочетаний всех возможных пар, троек, и т.д. значений конфигурационных параметров в рамках тестов. Тестовые наборы при этом достаточно компактны — размер покрывающего набора растет логарифмически от числа параметров, в то время как общий размер множества конфигураций растет экспоненциально. Кроме того, есть и эмпирические подтверждения того, что покрывающие наборы достаточно эффективны для выявления дефектов при использовании небольших тестовых наборов [1].

Данная работа представляет один из возможных подходов к построению покрывающих наборов, используемых далее для конфигурационного тестирования. Описываемая техника построения покрывающих наборов является эвристической комбинацией широко известных методов [2],

эффективность которой обуславливается особенностями области применения, — в общей ситуации использование этой техники, скорее всего, не будет настолько успешным. Новым элементом в представленной технике является учет условий использования параметров конфигураций (т.е., ограничений на значения других параметров, при нарушении которых значение рассматриваемого параметра вообще не влияет на работу системы), — в литературе и инструментах генерации покрывающих наборов, известных автору, иногда учитываются только общие ограничения на значения параметров, но не условия использования. В последних разделах статьи приводятся результаты практического применения описываемой техники для генерации тестовых программных (не включающих используемого оборудования) конфигураций операционной системы (ОС) реального времени, разрабатываемой в России.

Структура статьи такова: следующий раздел посвящен точной постановке задачи построения тестовых конфигураций, в четвертом разделе описывается применяемый метод решения этой задачи, пятый раздел представляет практические результаты ее использования, статья завершается заключением.

2. Построение тестовых конфигураций на базе покрывающих наборов

Входные данные для построения тестовых конфигураций задаются *классом конфигураций*, который включает следующие элементы.

- Набор *параметров конфигурации*.
  Для каждого параметра указываются
  - Имя параметра.
  - Конечное множество *допустимых значений* этого параметра.
  - *Условие использования* этого параметра. Это логическое ограничение на значения других параметров, при нарушении которого значение данного параметра не используется тестируемой системой. Т.е., такое ограничение может нарушаться в каких-то из тестовых конфигураций, но при этом значение данного параметра не учитывается в качестве значимого.
- *Общие ограничения* на значения параметров. Это логические ограничения на значения параметров, которые должны быть выполнены во всех тестовых конфигурациях.

Условия использования и общие ограничения можно рассматривать как выражения, построенные при помощи логических операций (отрицание, конъюнкция, дизъюнкция) из элементарных выражений вида p = v, где p — имя параметра данного класса, v — допустимое значение данного параметра.

(*Допустимой*) *конфигурацией* данного класса считается любой набор значений множества параметров этого класса (отображение имен параметров в их значения), присваивающий каждому параметру значение из множества его допустимых значений и удовлетворяющий общим ограничениям.

Сочетанием или *комбинацией значений* $v_{p_1}, \ldots, v_{p_k}$ параметров $p_1, \ldots, p_k$ будем называть отображение, для каждого i сопоставляющее p_i значение v_{p_i}.

Некоторое сочетание значений $v_{p_1}, \ldots, v_{p_k}$ называется *достижимым* в данном классе конфигураций, если существует допустимая конфигурация этого класса, в которой указанные параметры имеют заданные значения ($p_i = v_{p_i}$) и для каждого параметра p_i выполнено его условие использования. Не каждое сочетание значений является достижимым в произвольном классе конфигураций — некоторые могут нарушать общие ограничения класса, другие могут входить в противоречие с условиями использования участвующих в них параметров.

В данной статье рассматривается задача построения набора тестовых конфигураций на основе покрывающих наборов. Ее можно сформулировать так: для заданного класса конфигураций и натурального числа t построить такой набор допустимых конфигураций этого класса, что для любого достижимого в данном классе сочетания значений любых t различных параметров в построенном наборе есть конфигурация, в которой это сочетание значений реализуется (т.е., данные t параметров имеют заданные значения) и условия использования этих t параметров выполнены.

Связь этой задачи с покрывающими наборами видна из определения последнего понятия. *Покрывающим набором* глубины t для k факторов, могущих принимать, соответственно, $n_1, \ldots, n_k$ значений называется матрица A размера N×k, такая, что в i-ом ее столбце используются числа от 0 до n_i-1 и любая комбинация возможных значений любых t факторов встречается в хотя бы в одной из ее строк, т.е., $\forall j_1, \ldots, j_t \in [1..k]\ \forall v_1 \in [0..n_{j_1}-1], \ldots, v_t \in [0..n_{j_t}-1]\ \exists i \in [1..N]\ \forall q \in [1..t]\ A_{ij_q} = v_q$.

Если в заданном классе конфигураций k параметров, могущих принимать $n_1, \ldots, n_k$ значений, то всякое решение поставленной задачи будет соответствовать покрывающему набору — достаточно, как-то пронумеровав значения каждого параметра от 0 до n_i-1, заменить их на числа. Однако не всякий покрывающий набор дает нужное решение — все строки должны удовлетворять общим ограничениям, а учет комбинаций значений параметров ведется только по тем строкам, которые удовлетворяют условиям использования этих параметров.

Если предположить, что условия использования и общие ограничения не снижают существенно размерность пространства допустимых конфигураций, то можно воспользоваться известными методами [2] для быстрого построения подходящего покрывающего набора, а затем за счет небольших его

трансформаций добиться выполнения всех нужных ограничений. В статье описан именно такой подход к решению данной задачи при отсутствии общих ограничений (при решении ее на практике — построении тестовых конфигураций для операционной системы — таких ограничений не оказалось).

Если же имеющиеся ограничения существенно сокращают набор допустимых конфигураций (так, что, например, значения значительной части параметров можно вычислить, исходя из ограничений и значений оставшейся части параметров), указанная выше стратегия, скорее всего, не будет эффективной. В такой ситуации, по-видимому, проще сначала сократить число параметров, выбросив большинство вычислимых параметров из класса конфигураций.

3. Предлагаемый метод генерации тестовых конфигураций

В данном разделе описывается метод построения набора тестовых конфигураций для заданного класса конфигураций, решающий сформулированную выше задачу при отсутствии общих ограничений.

В качестве исходных данных используются класс конфигураций и натуральное число t — размер комбинаций значений параметров, которые нужно покрыть (для пар значений t = 2? lzk троек t = 3, и т.д.). Построение искомого набора конфигураций выполняется в три этапа.

1. Построение базового набора. Строится покрывающий набор глубины t для k факторов, могущих принимать n_1, …, n_k значений. Здесь k — число параметров в заданном классе конфигураций, n_i — число допустимых значений для параметра p_i.
2. Выполняется модификация базового набора, нацеленная на выполнение условий использования параметров.
3. Набор, являющийся числовой матрицей, преобразуется в набор конфигураций при помощи замены номеров значений параметров на сами эти значения. Строка набора при этом соответствует одной конфигурации.

Последний этап не содержит заметных сложностей, поэтому далее разобраны первые два.

3.1. Построение базового набора

На практике используются небольшие значения t = 2-4. Параметры конфигурации упорядочиваются по убыванию количества возможных значений.

Поскольку большинство параметров конфигурации операционной системы часто являются булевскими (они являются флагами, включаемыми/выключаемыми с помощью макроопределений #define/#undef),

для значения t = 2 имеет смысл выделить группу параметров с двумя возможными значениями и использовать для построения части набора, состоящей только из значений этих параметров, наиболее эффективный «булевский» алгоритм [2-4] — он позволяет быстро построить минимальный покрывающий набор глубины 2 для произвольного количества параметров, принимающих только два значения. Для t > 2 такого эффективного алгоритма не известно, используется «жадный» алгоритм [2,5].

Обычно в практических примерах есть небольшое число (2-5) параметров конфигурации с относительно большим количеством значений (10-30). Для построения части набора, связанной с этими параметрами, используется простая эвристика, основанная на частном случае «аффинного» алгоритма [2,4]. Этот алгоритм позволяет быстро построить покрывающий набор произвольной глубины для p+1 параметра, каждый из которых принимает p значений (где p — степень простого числа). Используемая эвристика построена следующим образом: для любого параметра, кроме первого и имеющих два значения, пусть число значений данного параметра равно n, пока n, деленное на наибольший общий делитель n и (максимального) числа значений первого параметра, не превосходит номера строки в наборе, для выбора значения данного параметра в данной строке используется та же формула, что и в «аффинном» алгоритме.

Для заполнения значений в строках, не подходящих под условие эвристики, а также для дополнения части набора для параметров с двумя значениями (или полного построения этой части для t > 2), используется «жадный» алгоритм [2,5] (поскольку параметры конфигурации упорядочиваются по убыванию количества значений, это вариант алгоритма IPO) в простейшей реализации, без вероятностного подбора кандидатов, — т.е., из множества значений-кандидатов в данную позицию набора выбирается такое значение, которое покрывает наибольшее число еще не покрытых комбинаций значений размера t (если таких значений несколько, берется первое найденное из них). Если же все комбинации t значений с участием данного параметра уже покрыты, оставляется неопределенное значение.

Позиции с неопределенными значениями используются далее для более компактного пополнения набора при учете условий использования.

Как видно, данная техника построена с учетом специфики практических примеров классов конфигураций ОС (большинство параметров имеет два значения, максимальное число значений параметра не превосходит 30, количество параметров с достаточно большим числом значений обычно меньше этого числа) и вряд ли будет давать хорошие результаты (достаточно компактные покрывающие наборы) в более общей ситуации. За счет отсутствия вероятностного перебора вариантов в реализации «жадного» алгоритма, данный метод работает несколько быстрее обычных таких реализаций, хотя получаемые с его помощью наборы могут иметь б***о***льший размер.

3.2. Выполнение условий использования параметров

На втором шаге нужно обеспечить выполнение условий использования параметров. Значение параметра в строке, где не выполняется условие его использования, не играет никакой роли, и поэтому не должно учитываться ни в каких комбинациях.

Для каждой пары (t = 2), тройки (t = 3) или четверки (t = 4) параметров выполняется четыре следующих шага.

- Определение противоречивости условий использования для данного набора параметров. Если эти условия противоречивы, добиваться покрытия различных комбинаций значений этих параметров не нужно — эти комбинации недостижимы.
- Определение ограничений на возможные комбинации в допустимых конфигурациях. Даже если условия использования параметров не противоречат друг другу, они могут ограничить достижимые комбинации их значений, например, если один параметр имеет 5 возможных значений, но в условие его использования входит требование того, что второй параметр имеет фиксированное значение, то достижимых пар их значений всего 5, пытаться покрыть другие пары не имеет смысла.
- Определение того, какие комбинации значений уже покрыты в наборе, полученном на предыдущем этапе, с выполненными условиями использования.
- Пополнение набора отсутствующими в нем достижимыми комбинациями с выполненными условиями использования.

Определение противоречивости условий использования само по себе может оказаться довольно трудной задачей — в общем случае она NP-полна [6], известные алгоритмы ее решения экспоненциальны (см, например, [7-9]).

Однако в примерах классов конфигураций, используемых на практике, условия использования часто оказываются не слишком сложны, в частности, будучи приведенными в конъюнктивную нормальную форму, включают конъюнкты с не более чем одной дизъюнкцией (именно так и оказывается во всех практических примерах). В этом случае можно использовать более эффективные алгоритмы ее решения [10-12], лучшие из которых линейны по размеру анализируемого выражения [12].

Алгоритм [12] очень нагляден:

- каждая дизъюнкция с двумя членами $x \vee y$ преобразуется в две импликации $\neg x \rightarrow y$ и $\neg y \rightarrow x$;
- строится граф с вершинами, соответствующими литералам (булевским выражениям, не сконструированным при помощи логических операций) или их отрицаниям, и с ребрами,

соответствующими импликациям;

- с помощью поиска в глубину находятся компоненты сильной связности в построенном графе (количество операций при этом линейно от числа ребер в графе);
- исходное выражение противоречиво тогда и только тогда, когда есть компонент сильной связности, содержащий одновременно литерал и его отрицание.

В данной работе условия использования параметров приводятся в конъюнктивную нормальную форму (КНФ), набор условий использования группы параметров при этом естественным образом оказывается в такой же форме, после чего для определения его противоречивости применяется алгоритм [12], слегка модифицированный, чтобы учесть конъюнкты, являющиеся литералами или их отрицаниями. При этом структура данных, представляющая анализируемые условия, является объединением графа дизъюнкций и обычного множества литералов и их отрицаний, индексированным по параметрам, задействованным в литералах. После работы алгоритма, если условия оказываются непротиворечивыми, из этой же структуры легко извлекаются ограничения, налагаемые на возможные комбинации значений параметров, входящих в группу.

Дальнейшие шаги вполне тривиальны. При пополнении набора отсутствующими в нем достижимыми комбинациями с выполненными условиями использования активно используются оставленные на предыдущих этапах неопределенные значения — их можно заменить на любое допустимое значение параметра, не сокращая множества покрытых комбинаций. Если добавить новую комбинацию (вместе с нужными для выполнения ее условий использования значениями других параметров) только за счет исправления неопределенных значений не получается, в набор добавляется новая строка, в которой определены значения лишь параметров, входящих в добавляемую комбинацию и условия ее использования, все остальные параметры остаются с неопределенными значениями.

4. Практические результаты

Описанный выше метод построения тестового набора конфигураций был реализован в программном инструменте на языке Java и использован для генерации наборов тестовых конфигураций для операционной системы реального времени на разных аппаратных платформах.

Входными данными для него служат Tcl-скрипты, представляющие собой приложение с графическим интерфейсом для настройки значений параметров. Код этого приложения содержит информацию о всех параметрах конфигурации ОС, их возможных значениях и условиях их использования.

Результатом работы является набор директорий, каждая из которых соответствует одной конфигурации и содержит заголовочные файлы языка C с задаваемым этой конфигурацией набором макроопределений.

В таблице 1 показаны результаты использования этого инструмента для генерации тестовых конфигураций, покрывающих все возможные пары значений параметров (t = 2), в разных классах, соответствующих различным аппаратным платформам.

Столбцы в таблице 1 представляют следующую информацию.

- Столбец Platform — идентификатор платформы.
- Столбец NP — число параметров конфигурации.
- Столбец Params — характеристика параметров конфигурации: по убыванию располагаются количества значений параметров, верхний индекс показывает число параметров с данным количеством значений.
- Столбец C — общее количество условий использования. Видно, что большинство параметров имеет нетривиальные условия использования. Во всех случаях подавляющее большинство условий в КНФ не использует дизъюнкций (для платформ 1 и 2 общее число дизъюнкций во всех условиях — 12, для остальных — 13). На всех платформах есть один параметр, условие использования которого включает 7 простых равенств и 4 дизъюнкции из двух равенств, самые сложные условия без дизъюнкций включают 7 равенств для платформ 1-11, и 6 равенств — для остальных.
- Столбец N1 — число конфигураций в получаемом наборе без учета условий использования.
- Столбец N2 — число конфигураций в получаемом наборе с учетом условий использования.
- Столбец T1 — время генерации набора без учета условий использования в секундах (показано среднее значение за 50 запусков).
- Столбец T2 — время трансформации набора, после которой условия использования выполнены, в секундах (показано среднее значение за 50 запусков).

№	Platform	NP	Params	C	N1	N2	T1	T2
1	i386/i386	228	$4^2 3^6 2^{220}$	172	25	92	0.01	0.31
2	i386/x86	215	$4^2 3^6 2^{207}$	170	23	72	0.01	0.26
3	mips/bt205	373	$17^2 15^1 8^8 7^2 6^1 5^3 4^{19} 3^{21} 2^{316}$	279	322	670	1.57	5.07
4	mips/ bt23-202	331	$17^2 15^1 13^4 9^4 7^2 5^3 4^7 3^{10} 2^{298}$	267	319	653	1.07	4.58
5	mips64/ bt128	548	$17^2 15^1 5^3 4^1 3^{15} 2^{526}$	500	289	556	0.57	7.88
6	mips64/ bt211	334	$17^2 15^1 13^4 9^4 7^2 5^3 4^5 3^{10} 2^{302}$	277	319	726	0.78	4.08
7	mips64/ bt206	334	$17^2 15^1 7^2 6^1 5^3 4^5 3^{18} 2^{302}$	278	289	651	0.58	2.99
8	mips64 /mpon	301	$17^2 15^1 5^3 4^1 3^{15} 2^{279}$	251	289	574	0.33	2.15
9	mips64/ vmips	241	$5^1 4^1 3^6 2^{233}$	203	27	89	0.01	0.42
10	mips64/ cprio64	563	$17^2 15^1 5^3 4^1 3^{15} 2^{541}$	511	289	564	0.58	6.70
11	komdiv64/ bt128	548	$17^2 15^1 5^3 4^1 3^{15} 2^{526}$	500	289	556	0.56	7.80
12	R4000/bt128	419	$17^2 15^1 5^3 4^2 3^{14} 2^{397}$	377	289	541	0.44	7.00
13	R4000/bt206	220	$17^2 15^1 14^{14} 7^2 6^1 5^3 4^6 3^{17} 2^{174}$	155	823	826	1.28	1.65
14	R4000/mpon	183	$17^2 15^1 5^3 4^2 3^{14} 2^{161}$	138	289	543	0.23	0.56
15	R4000/ vmips	113	$5^1 4^2 3^5 2^{105}$	80	28	62	0.01	0.07
16	R4000/ cprio64	434	$17^2 15^1 5^3 4^2 3^{14} 2^{412}$	388	289	549	0.46	5.59

Таблица 1. Результаты применения описанного метода — характеристики использованных классов конфигураций, полученных тестовых наборов и время генерации.

Таблица 1 демонстрирует, что предложенный метод генерации конфигурационных тестов работает достаточно эффективно на реальных примерах. Например, (строка 10) тестовый набор для класса конфигураций, имеющего 563 параметра, 511 из которых имеют нетривиальные условия использования, генерируется за 7 секунд и содержит всего лишь 564 тестовых конфигурации.

5. Заключение

В работе представлен метод автоматической генерации тестов для конфигурационного тестирования на основе покрывающих наборов. Новым элементом предложенного метода является учет условий использования отдельных параметров. Генерация осуществляется в два этапа: создание

покрывающего набора с нужными характеристиками без учета условий использования, основанное на широко известных «булевском», «аффинном» и «жадном» алгоритмах построения покрывающих наборов [2] и оставляющее часть значений неопределенными, и обеспечение выполнения условий использования, использующее доопределение неопределенных значений и пополнение набора.

Использованные при построении покрывающих наборов эвристики хорошо работают при выполнении тех ограничений, которые характерны для области применения метода: подавляющее большинство параметров имеют только два значения, максимальное количество значений параметра не превосходит нескольких десятков, число параметров, для которых возможно более двух значений, не больше этого максимального количества значений. В более общей ситуации эти эвристики не так эффективны.

Для выявления противоречивости условий использования применен алгоритм Аспвала-Пласса-Тарьяна [12], работающий для КНФ с двухчленными дизъюнкциями и имеющий линейную от размера условий сложность.

Предложенный подход не применим для классов конфигураций, имеющих общие ограничения на возможные наборы значений параметров. Расширение его на этот случай потребует, по-видимому, серьезных модификаций. В частности, при большом количестве таких ограничений становится неэффективно строить на первом этапе покрывающий набор с исходными характеристиками, в итоге его потребуется слишком сильно изменить. В таком случае, возможно, будет полезна предварительная трансформация набора параметров, в ходе которой должны быть удалены параметры, полностью вычисляемые (или «почти» вычисляемые, в некотором смысле) на основе других параметров.

Литература

[1] Cohen D. M., Dalal S. R., Parelius J., Patton G. C. The Combinatorial Design Approach to Automatic Test Generation. IEEE Software, 13(5):83-87, 1996.

[2] Кулямин В. В., Петухов А. А. Обзор методов построения покрывающих наборов. Программирование, 37(3):3-41, 2011.

[3] Greene C. Sperner families and partitions of a partially ordered set. In Combinatorics, Hall M. Jr. van Lint J., eds. Dordrecht, Holland, 1975, pp. 277-290.

[4] Hartman A. Software and Hardware Testing Using Combinatorial Covering Suites. Proc. of Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications, 2005, pp. 266-327.

[5] Bryce R. C., Colbourn C. J., Cohen M. B. A Framework of Greedy Methods for Constructing Interaction Test Suites. Proc. of Intl. Conf. on Software Engineering (ICSE'05), 2005, pp. 146-155.

[6] Cook S. The complexity of theorem proving procedures. Proc of 3-rd Annual ACM Symposium on Theory of Computing, 1971. pp. 151-158.

[7] Davis M., Logemann G., Loveland D. A machine program for theorem-proving. Communications of the ACM 5(7):394-397, 1962.

[8] Schoning T. A probabilistic algorithm for k-SAT and constraint satisfaction problems. Proc. of 40-th Annual Symposium on Foundations of Computer Science, 1999, pp. 410-414.
[9] Paturi R., Pudlak P., Saks M. E., Zani F. An improved exponential-time algorithm for k-SAT. Journal of the ACM, 52(3):337-364, 2005.
[10] Krom M. R. The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik, 13(1-2):15-20, 1967.
[11] Even S., Itai A., Shamir A. On the complexity of time table and multi-commodity flow problems. SIAM Journal on Computing, 5(4):691-703, 1976.
[12] Aspvall B., Plass M. F., Tarjan R. E. A linear-time algorithm for testing the truth of certain quantified boolean formulas. Information Processing Letters, 8(3):121-123, 1979.

Combinatorial generation of operation system software configurations

V.V. Kuliamin
ISP RAS, Moscow, Russia

Abstract. The paper presents an operating system configuration test generation method based on construction of covering arrays, that is ensuring coverage of all pairs, triple, etc. of configuration parameters values. The method combines known optimal algorithm for binary pairwise coverage array generation with further greedy construction of non-binary part of the array. Novelty of the method proposed is taking into account parameters usage conditions, that is constraints on parameters values, which should be obeyed to force the operating system under test to use the value of certain parameter. Usage conditions require to change both accounting of tuples covered and test construction process, which includes now the parametres assignments making all the necessary constraints to hold. Construction of such an assignment uses satisfiability check based on well-known Aspvall-Plass-Tarjan algorithm. The method proposed is applied in configuration test generation for real-time operating system with several hundreds configuration parameters, the results of this application demonstrate effectiveness of the method — usually several seconds is enough for generation of up to thousand different configurations (taking in account only analysis and generation, without input and output phases) where dozens of usage conditions are satisfied.

Keywords: configuration testing; covering array; test generation

References

[1]. Cohen D. M., Dalal S. R., Parelius J., Patton G. C. The Combinatorial Design Approach to Automatic Test Generation. IEEE Software, 13(5):83-87, 1996.
[2]. Kuliamin V., Petukhov A. A survey of methods for constructing covering arrays. Programming and Computer Software 37(3): 121-146, 2011.
[3]. Greene C. Sperner families and partitions of a partially ordered set. In Combinatorics, Hall M. Jr. van Lint J., eds. Dordrecht, Holland, 1975, pp. 277-290.
[4]. Hartman A. Software and Hardware Testing Using Combinatorial Covering Suites. Proc. of Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications, 2005, pp. 266-327.
[5]. Bryce R. C., Colbourn C. J., Cohen M. B. A Framework of Greedy Methods for Constructing Interaction Test Suites. Proc. of Intl. Conf. on Software Engineering (ICSE'05), 2005, pp. 146-155.
[6]. Cook S. The complexity of theorem proving procedures. Proc of 3-rd Annual ACM Symposium on Theory of Computing, 1971. pp. 151-158.
[7]. Davis M., Logemann G., Loveland D. A machine program for theorem-proving. Communications of the ACM 5(7):394-397, 1962.

[8]. Schoning T. A probabilistic algorithm for k-SAT and constraint satisfaction problems. Proc. of 40-th Annual Symposium on Foundations of Computer Science, 1999, pp. 410-414.
[9]. Paturi R., Pudlak P., Saks M. E., Zani F. An improved exponential-time algorithm for k-SAT. Journal of the ACM, 52(3):337-364, 2005.
[10]. Krom M. R. The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik, 13(1-2):15-20, 1967.
[11]. Even S., Itai A., Shamir A. On the complexity of time table and multi-commodity flow problems. SIAM Journal on Computing, 5(4):691-703, 1976.
[12]. Aspvall B., Plass M. F., Tarjan R. E. A linear-time algorithm for testing the truth of certain quantified boolean formulas. Information Processing Letters, 8(3):121-123, 1979.

Использование аспектно-ориентированного программирования для выполнения запросов по исходному коду программ[1]

Новиков Е.М., Хорошилов А.В.
joker@ispras.ru, khoroshilov@ispras.ru

Аннотация. Запросы по исходному коду программ помогают разработчикам обнаруживать искомые фрагменты кода и определять их взаимоотношения друг с другом. Для выполнения запросов по исходному коду автоматизированным образом существуют различные подходы от достаточно простых, основывающихся на текстовом поиске по шаблонам, до более интеллектуальных, осуществляющих поиск на основе формального представления программ и позволяющих использовать для запросов естественные языки. В статье предлагается подход к выполнению запросов по исходному коду программ на основе аспектно-ориентированного программирования, рассматриваются достоинства и недостатки такого подхода.

Ключевые слова: разработка программы; поддержка программы; запрос по исходному коду программы; формальное представление программы; аспектно-ориентированное программирование.

1. Введение

При разработке и поддержке программ важными задачами являются обнаружение искомых фрагментов кода и определение их взаимоотношений друг с другом. Данные задачи возникают, например, когда разработчики пытаются понять на уровне исходного кода, как в программе реализована та или иная функциональность; при рефакторинге программ; при обратном проектировании; во время отладки программ и при вычислении различных метрик исходного кода. С целью обнаружения искомых фрагментов кода и определения их взаимоотношений друг с другом используются запросы по исходному коду программ.

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 11.519.11.4006)

Все возможные запросы по исходному коду программ могут быть условно отнесены к одному из трех типов [1]:

- Запросы по общей структуре программ. Например, о составе и связи файлов, функций, переменных и типов, составляющих программу.
- Запросы по структуре выражений. Например, найти все вызовы функции, использования поля структуры, циклы и т.д.
- Запросы по потоку управления и потоку данных. Например, найти последовательности выражений, в которых участвует некоторая переменная.

Делать запросы по исходному коду вручную для больших программ достаточно трудоемко, поскольку объем исходного кода очень велик, а взаимоотношения между различными его частями бывают очень сложными. С целью автоматизации процесса были предложены различные подходы от достаточно простых, основывающихся на текстовом поиске по шаблонам, до более интеллектуальных, осуществляющих поиск на основе формального представления программ и позволяющих использовать для запросов естественные языки. В статье предложен подход к написанию и выполнению запросов по исходному коду программ на основе аспектно-ориентированного программирования (АОП).

В разделе 2 данной статьи приведено описание основных понятий АОП. В разделе 3 кратко рассматриваются возможности C Instrumentation Framework, одной из реализаций АОП для языка программирования Си. В разделе 4 обсуждается возможность использования АОП для выполнения запросов по исходному коду программ. Показывается, как с этой целью был доработан C Instrumentation Framework. В разделе 5 предложенный подход сравнивается с существующими подходами. В заключении подводятся итоги работы и рассматриваются направления дальнейшего развития.

2. Основные понятия аспектно-ориентированного программирования

Аспектно-ориентированное программирование (АОП) предлагает специальные средства для поддержки модульности программ в тех случаях, когда существующие средства языков программирования не позволяют выделить определенную функциональность в отдельные модули. Подобная функциональность получила название *сквозной функциональности*. Примерами сквозной функциональности являются ведение журнала, трассировка и обработка ошибок.

В АОП сквозная функциональность программы выделяется в отдельные модули, так называемые *аспекты*, с помощью срезов и рекомендаций. В данной статье под *срезом* будет пониматься описание множества фрагментов исходного кода (*точек соединения*) программы, удовлетворяющих

некоторому логическому условию. Посредством среза можно задать, например, соответствие всем вызовам функций выделения памяти (таким как *malloc*, *calloc* и т.д.).

Рекомендация указывает, каким образом необходимо преобразовать точки соединения программы, соответствие которым задается некоторым срезом. Как правило, эти преобразования заключаются в добавлении набора инструкций до, после или вместо соответствующих точек соединения. Инструкции записываются с помощью того же языка программирования, на котором пишется целевая программа. Помимо этого АОП предоставляет возможность использовать информацию о соответствующей точке соединения, например, имя вызываемой функции, типы ее аргументов и т.п.

```
// Аспект Logging состоит из среза и рекомендации.
aspect Logging {
  // Срез move задает соответствие точкам соединения программы,
  // вызовам методов.
  pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.setX(int))              ||
    call(void Point.setY(int));
  // Рекомендация говорит, что перед выполнением точек соединения
  // среза на экран должно быть напечатано сообщение.
  before(): move() {
    System.out.println("about to move");
  }
```

Рис. 1. Пример аспекта AspectJ, выделяющего сквозную функциональность ведения журнала для графической системы.

На Рис.1 показан пример аспекта AspectJ, посредством которого для графической системы выделена сквозная функциональность, ведение журнала [7]. Для этого в аспекте задается срез *move*, задающий соответствие точкам соединения, вызовам методов *setXY* класса *FigureElement* и вызовам методов *setX* и *setY* класса *Point*. Кроме того, в аспекте *Logging* задается рекомендация, которая говорит, что перед выполнением точек соединения среза *move* (вызовом соответствующих методов) на экран должно быть напечатано сообщение.

Процесс применения аспектов к целевой программе, при котором для соответствующих срезам точек соединения выполняются указанные рекомендациями преобразования, называется *инструментированием*. Предполагается, что инструментирование программ должно выполняться автоматически.

Чтобы реализовать АОП для некоторого языка программирования необходимо определить механизм описания аспектов и разработать средства инструментирования программ. В настоящее время существует достаточно большое количество реализаций АОП для различных языков программирования. В следующем разделе приведено краткое описание инструмента C Instrumentation Framework, который является одной из реализаций АОП для языка программирования Си.

3. C Instrumentation Framework - реализация АОП для языка программирования Си

Реализация АОП для языка программирования Си C Instrumentation Framework (CIF) [5,6] была разработана в рамках проекта верификации драйверов Linux Driver Verification [2-4]. В этом проекте возможности CIF используются, во-первых, для формализации правил корректного поведения драйвера в виде аспектов, а во-вторых, для подготовки исходного кода драйверов к верификации путем его инструментирования на основе данных аспектов. Далее в статье подробнее рассмотрен процесс инструментирования CIF. Одновременно с этим рассматриваются возможности CIF для разработки аспектов.

CIF выполняет инструментирование исходного кода программы в течении 4 этапов:

- На 1-м этапе CIF позволяет дописывать текст до и после инструментируемого исходного кода. Благодаря этому, например, в начало файла можно добавить прототипы функций, необходимые перед их первым использованием, и включения дополнительных заголовочных файлов.
- На 2-м этапе выполняется стандартное препроцессирование исходного кода программы. На этом этапе CIF позволяет инструментировать соответствующие срезам макроопределения, макрофункции и их подстановки. В случае макроопределения и макрофункции в рекомендации предоставляется возможность задать собственный текст для подстановки (для макрофункции в данном тексте можно ссылаться на формальные параметры). Для подстановок макрофункций CIF позволяет использовать фактические параметры.
- На 3-м этапе CIF позволяет инструментировать вызовы и определения функций, простые использования и присваивания переменных и параметров. При этом в рекомендации возможно использовать произвольный корректный код на языке Си и ссылаться на формальные параметры, на их типы, на тип возвращаемого значения, на имя функции, переменной или параметра в зависимости от того, соответствие какой точке соединения задает срез. Данный код реализуется в виде вспомогательных функций. Также на 3-м этапе

можно расширить определения составных типов данных (структур, объединений и перечислений), дописав в начало или в конец дополнительные поля или константы.

- На 4-м заключительном этапе CIF связывает исходные конструкции с вспомогательными функциями, после чего выдает на вывод либо исходный код, либо продолжает работать дальше, как стандартный компилятор.

На каждой из 4 стадий CIF вызывает инструмент Aspectator, который представляет собой модифицированную версию компилятора GCC 4.6.1 [8]. Благодаря этому CIF может обрабатывать исходный код на языке Си с расширениями GNU и выдавать на выход те представления программы, которые поддерживаются GCC. Дополнительно к этому поддерживается вывод в виде исходного кода.

4. Использование реализации АОП для выполнения запросов по исходному коду программ

С точки зрения выполнения запросов по исходному коду реализацию АОП можно использовать следующим образом. Запросы естественным образом писать с помощью аспектов. Поскольку для выполнения запросов инструментирование кода как таковое не требуется[2], достаточно задействовать возможности АОП по заданию срезов, поиску соответствующих точек соединения в коде программы и получению информации касательно данных точек соединения. Таким образом, получается, что единственное дополнительное требование к реализации АОП, необходимое для выполнения запросов по исходному коду программ, заключается в возможности вывода полученной информации о точках соединения, соответствующих заданным в аспектах срезам.

Для того чтобы использовать C Instrumentation Framework, реализацию АОП для языка программирования Си, для выполнения запросов по исходному коду в инструменте была добавлена поддержка специальной конструкции *$fprintf*. Данная конструкция позволяет напечатать в указанный файл информацию о соответствующей точке соединения на этапе выполнения инструментирования программы. Далее в статье рассмотрены несколько примеров запросов по исходному коду, которые можно сделать с помощью доработанной версии инструмента.

[2] В статье не рассматриваются запросы к исходному коду программ, результаты которых анализируются по выводу, генерируемому в процессе выполнения данных программ.

4.1. Получение фактического параметра макрофункции

```
before: expand(module_init(init))
{
  $fprintf<out_file_name,$arg_val1>
}
```

Рис. 2. Задание функции загрузки драйвера устройства электронной защиты Parallax LiteLink..

```
linux/drivers/net/irda/litelink-sir.c:
static int __init litelink_sir_init(void)
{
        return irda_register_dongle(&litelink);
}
module_init(litelink_sir_init);
```

Рис. 3. Аспект для получения фактического параметра макрофункции module_init.

Запросы на получение фактического параметра макрофункции возникают при построении окружения для драйверов в проекте верификации драйверов Linux Driver Verification [2-4]. Имена функций загрузки драйверов, необходимые для построения окружения, передаются в качестве параметров макрофункции *module_init* (Рис. 2). Задать запрос по исходному коду драйвера можно с помощью аспекта, представленного на Рис. 3. Данный аспект требует перед выполнением подстановки макрофункции *module_init* с одним параметром напечатать в файл *out_file_name* ее первый фактический параметр. В результате применения аспекта к файлу драйвера *linux/drivers/net/irda/litelink-sir.c* в данный файл будет напечатано имя функции загрузки драйвера *litelink_sir_init*.

На Рис. 4 представлен способ задания функции загрузки драйвера ОС Linux через несколько последовательных подстановок макрофункций. При препроцессировании сначала *module_usb_driver(karma_driver)* будет заменена на *module_driver(karma_driver, usb_register, usb_deregister)*, а эта подстановка в свою очередь – на определение функции загрузки драйвера и *module_init(karma_driver_init)*. Для выполнения запроса в случае

использования данного способа задания функции загрузки аспект, представленный на Рис. 3, также применим. В результате в файл *out_file_name* будет напечатано *karma_driver_init*.

```
include/linux/device.h:
#define module_driver(__driver, __register, __unregister, ...) \
static int __init __driver##_init(void) \
{ \
        return __register(&(__driver) , ##__VA_ARGS__); \
} \
module_init(__driver##_init);
...
include/linux/usb.h:
#define module_usb_driver(__usb_driver) \
        module_driver(__usb_driver, usb_register, \
                      usb_deregister)
drivers/usb/storage/karma.c:
module_usb_driver(karma_driver);
```

Рис. 2. Задание функции загрузки драйвера цифрового аудио проигрывателя Rio Karma..

Получение списка вызываемых функций, у которых тип одного из параметров является указателем

Для того чтобы задать запрос на получение списка вызываемых функций, у которых тип одного из параметров является указателем на структуру *mutex*, можно использовать аспект, приведенный на Рис. 6. При его применении к фрагменту исходного кода, представленному на Рис. 5, в файл *out_file_name* будет напечатано *mutex_lock, mutex_lock_nested, mutex_unlock, mutex_unlock*.

```
drivers/scsi/libfc/fc_npiv.c
void fc_vport_setlink(struct fc_lport *vn_port)
{
  ...
  mutex_lock(&n_port->lp_mutex);
  mutex_lock_nested(&vn_port->lp_mutex,
LPORT_MUTEX_VN_PORT);
  __fc_vport_setlink(n_port, vn_port);
  mutex_unlock(&vn_port->lp_mutex);
  mutex_unlock(&n_port->lp_mutex);
}
```

Рис. 5. Вспомогательная функция для библиотеки функций волоконно-оптического канала, вызывающая функции, у которых тип одного из параметров является указателем на структуру mutex.

```
before: call($ $(.., struct mutex *, ..))
{
  $fprintf<out_file_name,$func_name>
}
```

Рис. 6. Аспект для получения списка вызываемых функций, у которых тип одного из параметров является указателем на структуру mutex.

4.2. Получение списка функций, в которых изменяется глобальная переменная

Предположим, что перед разработчиком стоит задача определить те функции реализации уровня записи пакетов для устройств CD-RW, DVD+RW, DVD-RW и DVD-RAM, в которых изменяется глобальная переменная *pkt_debugfs_root* (Рис. 7). С этой целью он может использовать запрос по исходному коду в виде аспекта, приведенного на Рис. 8. В результате в файл *out_file_name* будет напечатано только имя функции *pkt_debugfs_init*, поскольку в функции *pkt_debugfs_dev_new* переменная *pkt_debugfs_root* не изменяется.

```
drivers/block/pktcdvd.c
static struct dentry *pkt_debugfs_root;
static void pkt_debugfs_dev_new(struct pktcdvd_device *pd) {
  if (!pkt_debugfs_root)
    return;
  ...
}
static void pkt_debugfs_init(void)
{
  pkt_debugfs_root = debugfs_create_dir(DRIVER_NAME, NULL);
  ...
}
```

Рис. 7. Функции реализации уровня записи пакетов для устройств CD-RW, DVD+RW, DVD-RW и DVD-RAM, в которых встречается глобальная переменная pkt_debugfs_root.

```
before: set(static struct dentry *pkt_debugfs_root)
{
  $fprintf<out_file_name,$scope_func_name>
}
```

Рис. 8. Аспект для получения списка вызываемых функций, у которых тип одного из параметров является указателем на структуру mutex.

4.3. Поиск файла, в котором определяется тип данных

В качестве последнего примера рассмотрим типичную задачу поиска файла, в котором определяется тип данных. Например, для того, чтобы найти заголовочный файл в ядре операционной системы Linux, в котором определяется структура *device*, можно написать запрос в виде аспекта, представленного на Рис. 9. В результате применения данного аспекта к исходному коду ядра в файл *out_file_name* будет выведено имя заголовочного файла *include/linux/device.h*.

```
before: introduce(struct device)
{
  $fprintf<out_file_name,$scope_file_name>
}
```

Рисунок 9. Аспект для поиска заголовочного файла в ядре операционной системы Linux, в котором определяется структура device.

5. Существующие подходы

Для выполнения запросов по исходному коду программ существует много разнообразных подходов. Данные подходы продолжают развиваться по мере роста объема и сложности исходного кода. Один из первых подходов для выполнения запросов заключался в использовании регулярных выражений для поиска соответствующих текстовых фрагментов в программах. Данный подход обладает следующими преимуществами:

- высокая скорость работы;
- независимость языка запросов от языка программирования;
- отсутствие необходимости предварительного преобразования исходного кода программ в какое-либо формальное представление;
- терпимость к ошибкам в коде;
- простота использования.

Благодаря этим свойствам регулярные выражения используются для выполнения запросов по исходному коду программ и в настоящее время. Например, данный подход используется в проекте Linux Driver Verification для построения окружения драйверов. Опыт использования регулярных выражений в данном и других проектах показал, что некоторые запросы, в особенности, касающиеся сложных выражений и их взаимоотношений, сформулировать достаточно сложно. Зачастую при этом используются эвристики, которые не могут гарантировать ни то, что результаты запросов правильные, ни то, что вся необходимая информация извлекается.

Другие подходы для выполнения запросов по исходному коду программ предполагают предварительное преобразование кода к какому-либо формальному представлению, например абстрактному синтаксическому дереву или графу потока управления. Как правило, это, во-первых, позволяет существенно облегчить формулировку сложных запросов, а во-вторых, результаты выполнения данных запросов более точны по сравнению с использованием регулярных выражений. Основные отличия подходов заключаются:

- в языке написания запросов;
- в способе выполнения запросов;
- в способе хранения формального представления исходного кода программ.

Один из первых подходов, использующих предварительное преобразование исходного кода программ к формальному представлению, был реализован в инструменте SCRUPLE [9]. В качестве языка запросов в SCRUPLE используется расширение того языка, на котором написана программа. Для выполнения запросов, они преобразуются в конечные автоматы, после чего данные автоматы применяются к абстрактному синтаксическому дереву программы. В инструменте tawk [10] для написания запросов был предложен абстрактный язык, не привязанный к конкретному языку программирования.

С целью повышения эффективности хранения формального представления исходного кода программы и выполнения запросов было предложено использовать возможности реляционных баз данных. Один из первых инструментов, в котором был реализован данный подход, - это OMEGA [11]. OMEGA хранил формальное представление исходного кода и выполнял запросы неэффективно. Эти ограничения были частично устранены, например, в инструменте CodeQuest [12]. Запросы по исходному коду программ в инструменте CodeQuest делаются с помощью языка Datalog, который является подмножеством языка Prolog и используется специально для написания запросов к базам данных.

Позднее было замечено, что выполнение одного запроса по исходному коду не всегда предоставляет разработчику сразу всю необходимую информацию. Часто приходится выполнять несколько запросов, результаты которых не связывались друг с другом при использовании существующих инструментов. С целью решения данной проблемы были разработаны инструменты FEAT и JQuery [13, 14]. Отличие этих инструментов заключается в том, что JQuery позволяет привязать результаты выполнения запросов к иерархическому представлению исходного кода (например, к иерархии классов или к схеме вызываемых функций), а FEAT позволяет итеративно уточнять представление реализации сквозной функциональности в программе. FEAT и JQuery предназначены для выполнения запросов по исходному коду на языке программирования Java. Инструменты являются плагинами к интегрированной среде разработки Eclipse [15].

Для выполнения запросов к программам на языке Си формальное представление строится либо до препроцессирования кода (например, в инструменте Coccinelle [16]), либо после. Использование в качестве основы непрепроцессированного кода позволяет, во-первых, осуществлять поиск по всему исходному коду программы, а во-вторых, возвращать результаты в терминах оригинального представления программы. Однако, результаты

выполнения запросов могут оказаться менее точными, поскольку формальное представление в данном случае строится с использованием эвристик.

Одной из последних разработок в области выполнения запросов по исходному коду программ является поддержка естественных языков для написания запросов [17]. Использование данного подхода значительно упрощает процедуру написания запросов, поскольку от разработчиков не требуется знание специфичных языков запросов. Однако, результаты практического применения показали, что подход не является надежным (распознается около 80% запросов).

Предложенный в данной статье подход целесообразно сравнивать с существующими подходами, использующими предварительное преобразование исходного кода программ к формальному представлению. Данные подходы потенциально нацелены на выполнение произвольных запросов по исходному коду. По сравнению с ними предложенный подход не является универсальным, поскольку АОП поддерживает относительно простые точки соединения, что позволяет делать запросы по общей структуре программ и по структуре выражений, но не позволяет делать запросы по потоку управления и потоку данных.

К преимуществам предложенного подхода можно отнести то, что для его реализации в достаточно высокой степени переиспользуется существующая реализация АОП для соответствующего языка программирования. Для написания запросов используются практически те же средства АОП, что и для выделения сквозной функциональности в виде аспектов. Это позволяет разработчикам, которые знают или используют АОП, задействовать возможности предложенного подход для выполнения запросов по исходному коду без значительных усилий.

Ввиду того, что предложенная реализация основана на C Instrumentation Framework, который использует компилятор GCC для представления исходного кода программы в виде дерева разбора, она может быть использована для любых программ на языке Си с расширениями GNU. Для других подходов подобное зачастую является существенным ограничением, поскольку они используют собственные парсеры языка. Благодаря выполнению запросов на основе дерева разбора предложенный подход позволяет получать более точные результаты по сравнению с результатами, получаемыми на основе формального представления непрепроцессированного кода. Также можно отметить, что с помощью предложенного подхода можно делать запросы не только на основе дерева разбора программ, но и к конструкциям препроцессора, в том числе тем, которые возникают только в ходе выполнения препроцессирования. Насколько известно авторам статьи, последнее не поддерживается другими инструментами.

6. Заключение

В статье рассмотрена задача выполнения запросов по исходному коду автоматизированным образом. Данная задача остро встает перед программистами в процессе разработки и поддержки больших программ. Для решения задачи было предложено множество различных подходов. Самые первые подходы использовали для запросов по исходному коду программ регулярные выражения. Благодаря своей простоте и эффективности данные подходы используются до сих пор. Однако, с помощью регулярных выражений сложно написать запросы для получения информации о сложно взаимосвязанных фрагментах исходного кода. Поэтому в более поздних подходах было предложено осуществлять поиск по какому-либо формальному представлению исходного кода программ, например, абстрактному синтаксическому дереву. В дальнейшем было сделано много исследований по упрощению языка написания запросов (использование абстрактных и даже естественных языков) и по повышению эффективности их выполнения (использование реляционных баз данных для хранения формального представления исходного кода программы и выполнения запросов по нему).

В статье для выполнения запросов по исходному коду программ предложено использовать подход аспектно-ориентированного программирования. Для этого рассматриваются основные понятия АОП и особенности инструмента C Instrumentation Framework, который является одной из реализаций АОП для языка программирования Си. Показывается, что незначительная доработка инструмента позволяет добавить в CIF поддержку требуемой функциональности. В статье демонстрируются несколько примеров использования предложенного подхода, в том числе при построении окружения для драйверов в проекте верификации драйверов Linux Driver Verification.

Предложенный подход уступает по возможностям специализированным инструментам, поскольку он не позволяет делать запросы по потоку управления и потоку данных. К преимуществам подхода относится то, что его реализация в достаточно большой степени переиспользует реализацию АОП для соответствующего языка программирования и что подход позволяет задействовать традиционные средства АОП, используемые для выделения сквозной функциональности, для написания запросов по исходному коду программ.

Актуальная версия инструмента CIF с поддержкой выполнения запросов по исходному коду программ на языке Си доступна под лицензией GPLv3 [6].

Литература

[1] S. Paul and A. Prakash. Querying Source Code using an Algebraic Query Language. In Proceedings of the International Conference on Software Maintenance, pp. 127-136, 1994.

[2] В.С. Мутилин, Е.М. Новиков, А.В. Страх, А.В. Хорошилов, П.Е. Швед. Архитектура Linux Driver Verification. Труды Института системного программирования РАН, том 20, стр. 163-187, 2011.
[3] A. Khoroshilov, V. Mutilin, E. Novikov, P. Shved, A. Strakh. Towards an Open Framework for C Verification Tools Benchmarking. Proceedings of the Eighth International Andrei Ershov Memorial Conference «Perspectives of Systems Informatics» (PSI 2011), pp. 82-91, 2011.
[4] Проект верификации драйверов Linux Driver Verification. http://forge.ispras.ru/projects/ldv.
[5] E. Novikov. One Approach to Aspect-Oriented Programming Implementation for the C programming language. roceedings of the 5th Spring/Summer Young Researchers' Colloquium on Software Engineering, Yekaterinburg, pp. 74-81, 12-13 May, 2011.
[6] Реализация аспектно-ориентированного программирования для языка Си C Instrumentation Framework. http://forge.ispras.ru/projects/cif.
[7] Реализация аспектно-ориентированного программирования для языка Java AspectJ. http://www.eclipse.org/aspectj.
[8] GNU Compiler Collection. http://gcc.gnu.org.
[9] S. Paul, A. Prakash. A framework for source code search using program patterns. In IEEE Transactions on Software Engineering, pp. 463–475, 1994.
[10] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In WPC'96: Proceedings of the 4th International Workshop on Program Comprehension (WPC'96), Washington, DC, USA, p. 144, 1996.
[11] J. Ebert, B. Kullbach, A. Winter. Querying as an Enabling Technology in Software Reengineering, Proceedings of the Third European Conference on Software Maintenance and Reengineering, p.42, March 03-05, 1999.
[12] E. Hajiyev, M. Verbaere, O. de Moor. CodeQuest: scalable source code queries with datalog. Proceedings of the 20th European conference on Object-Oriented Programming, Nantes, France, July 03-07, 2006.
[13] M. P. Robillard, G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. Proceedings of the 24th International Conference on Software Engineering, May 19-25, Orlando, Florida, 2002.
[14] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD '03), 178-187, 2003.
[15] Интегрированная среда разработки Eclipse. http://www.eclipse.org.
[16] H. Stuart. Hunting bugs with Coccinelle. Masters Thesis, University of Copenhagen, August, 2008.
[17] M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11), IEEE Computer Society, Washington, DC, USA, 376-379, 2011.

Using Aspect-Oriented Programming for Querying Source Code

Novikov E.M., Khoroshilov A.V.
novikov@ispras.ru, khoroshilov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Querying source code helps developers to discover code fragments of interest and to determine their interrelations with each other. Different approaches exist to execute source code queries automatically. Some of them are based on a rather simple text pattern matching. More advanced approaches provide abilities to use natural languages for queries and perform matching on the basis of a formal program representation. We suggest using aspect-oriented programming for querying source code and consider advantages and disadvantages of this approach. The paper introduces main conceptions of aspect-oriented programming and briefly presents C Instrumentation Framework – an aspect-oriented programming implementation for the C programming language. We slightly extended C Instrumentation Framework so that it could execute source code queries written in form of aspects. The paper gives several examples of utilization of the suggested approach in particular for collecting information required for generating environment models for device drivers in the Linux Driver Verification project. The suggested approach is inferior to special querying tools in making queries by control and data flows. But it is simple in implementation and allows one to use traditional means of aspect-oriented programming for developing and executing source code queries.

Keywords: program developing; program maintaining; source code querying; formal program representation; aspect-oriented programming.

References

[1]. Paul S., Prakash A. Querying Source Code using an Algebraic Query Language. In Proc. International Conference on Software Maintenance (ICSM), pp. 127-136, 1994. doi: 10.1109/ICSM.1994.336782

[2]. Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).

[3]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an Open Framework for C Verification Tools Benchmarking. In Proc. Perspectives of Systems Informatics (PSI), LNCS, vol 7162, pp. 82-91, 2012. doi: 10.1007/978-3-642-29709-0_17.

[4]. Linux Driver Verification Project. http://forge.ispras.ru/projects/ldv.

[5]. Novikov E. One Approach to Aspect-Oriented Programming Implementation for the C programming language. In Proc. 5th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), pp. 74-81, 2011.

[6]. C Instrumentation Framework: aspect-oriented programming implementation for the C programming language. http://forge.ispras.ru/projects/cif.

[7]. AspectJ: aspect-oriented programming implementation for the Java programming language. http://www.eclipse.org/aspectj.

[8]. GNU Compiler Collection. http://gcc.gnu.org.
[9]. Paul S., Prakash A. A framework for source code search using program patterns. IEEE Transactions on Software Engineering, pp. 463–475, 1994. doi: 10.1109/32.295894
[10]. Griswold W.G., Atkinson D.C., McCurdy C. Fast, flexible syntactic pattern matching and processing. In Proc. 4th International Workshop on Program Comprehension (WPC), pp. 144-153, 1996. doi: 10.1109/WPC.1996.501129
[11]. Ebert J., Kullbach B., Winter A. Querying as an Enabling Technology in Software Reengineering. In Proc. 3rd European Conference on Software Maintenance and Reengineering (CSMR), pp. 42-50, 1999. doi: 10.1109/CSMR.1999.756681
[12]. Hajiyev E., Verbaere M., de Moor O. CodeQuest: scalable source code queries with datalog. In Proc. ECOOP 2006 – Object-Oriented Programming, LNCS, vol. 4067, pp. 2-27, 2006. doi: 10.1007/11785477_2
[13]. Robillard M.P., Murphy G.C. Concern graphs: finding and describing concerns using structural program dependencies. In Proc. 24th International Conference on Software Engineering (ICSE), pp. 406-416, 2002. doi: 10.1145/581339.581390
[14]. Janzen D., Volder K.D. Navigating and querying code without getting lost. In Proc. 2nd international conference on Aspect-Oriented Software Development (AOSD), pp. 178-187, 2003. doi: 10.1145/643603.643622
[15]. Eclipse IDE. http://www.eclipse.org.
[16]. Stuart H. Hunting bugs with Coccinelle. University of Copenhagen, Masters Thesis, 2008.
[17]. Kimmig M., Monperrus M., Mezini M. Querying source code with natural language. In Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 376-379, 2011. doi: 10.1109/ASE.2011.6100076

Разработка тестового набора для верификации реализаций протокола безопасности TLS

А.В. Никешин, Н.В. Пакулин, В.З. Шнитман

Аннотация. Статья посвящена разработке тестового набора для проверки соответствий реализаций узлов Интернет спецификациям протокола безопасности TLS [1-11]. Для построения тестового набора использовалась технология автоматического тестирования UniTESK [12] и программный пакет JavaTESK [13], реализующий эту технологию.

Работа выполнялась в Институте системного программирования РАН в рамках проекта «Верификация реализаций расширяемых протоколов Интернета» при поддержке гранта РФФИ № 10-07-00145. В ходе ее выполнения были выделены требования к реализациям TLS, разработаны формальные спецификации и прототип тестового набора для верификации реализаций TLS. В статье кратко описаны метод формализации требований TLS, тестовый набор, а также результаты тестирования существующих реализаций сервера TLS. Эти результаты показывают, что предложенный в данной работе метод верификации позволяет эффективно автоматизировать тестирование таких сложных протоколов, как протоколы безопасности.

Ключевые слова: тестирование, верификация, формальные методы, формальные спецификации, MBT, тестирование с использованием моделей, модели программ, TLS, SSL, JavaTESK, UniTESK.

1. Введение

Протокол TLS обеспечивает защиту передаваемых данных между приложениями, взаимодействующими по схеме клиент-сервер. Он располагается поверх протокола транспортного уровня, инкапсулируя протоколы прикладного уровня (такие как HTTP, FTP, SMTP, NNTP, XMPP, LDAP). Первоначально TLS использовался с надежными транспортными протоколами, такими как TCP. Позже появилась реализация для протоколов, ориентированных на передачу дейтаграмм (таких как UDP, DCCP), выделенная в независимый стандарт DTLS (Datagram Transport Layer Security).

В настоящее время TLS широко используется в сочетании с самыми разными протоколами:

– для защиты WWW-трафика (протокол HTTPS),

- с почтовыми протоколами SMTP, IMAP, POP3,
- при организации виртуальных защищенных сетей (OpenVPN),
- с протоколом запуска сессий (SIP: Session Initiation Protocol) и основанными на нем приложениями (например VoIP),
- с протоколом LDAP в качестве средства шифрования сеансов LDAP и защиты от спуфинга.

Задачи, решаемые протоколом, в порядке приоритетности:

1. Криптографическая безопасность: TLS используется для установления безопасного соединения между двумя участниками.
2. Совместимость: взаимодействие приложений по протоколу TLS, не зависит от внутренних особенностей реализаций.
3. Расширяемость: возможность расширения функциональности, в том числе добавления новых алгоритмов шифрования и обеспечения целостности данных.
4. Относительная эффективность: криптографические операции, особенно операции с открытым ключом, требуют значительных затрат вычислительных ресурсов. В TLS имеется механизм кэширования сессий, который позволяет уменьшить количество соединений, и, как следствие, усилить безопасность за счет уменьшения сетевой активности.

Задачу тестирования соответствия можно условно разделить на две подзадачи: построение тестовых воздействий и оценка правильности наблюдаемых результатов. К первой задаче примыкает проблема оценки полноты покрытия – чем шире будет спектр тестовых воздействий, тем шире получится охват функций протокола при тестировании. Вторая задача заключается в вынесении вердикта о соответствии тестируемой системы спецификации протокола.

Разработанный метод верификации [14] основан на автоматизированном тестировании соответствия формальным спецификациям. Требования, представленные в тексте стандарта, изложены на английском языке и представляют собой неформальный текст, описывающий желаемое поведение системы на естественном языке. Для того, чтобы автоматизировано извлечь тесты для протокола, необходимо перевести его спецификацию в вид, пригодный для решений этой задачи. В разработанном подходе в этой роли выступают формальные спецификации, в которых требования задаются как логические выражения, записанные посредством математического формализма.

В основе подхода лежит представление протоколов как асинхронных автоматов, причем автомат задаётся неявно посредством контрактных спецификаций. А именно, формальное описание поведения протокола задаётся как контрактная спецификация: набор сообщений протокола рассматривается как некоторый формальный интерфейс между реализацией

протокола и её окружением, поведение протокола описывается посредством пред- и постусловий. Такое задание протоколов позволяет описывать поведение сложных недетерминированных протоколов, таких как протоколы защиты передачи данных в сетях IP. Эти протоколы отличаются сложными структурами данных сообщений и состояния, недетерминированным поведением и неполными спецификациями – в спецификациях умышленно оставлены пробелы для облегчения реализации протоколов. Контрактные спецификации позволяют представлять требования к сложным протоколам в форме, пригодной для автоматизированного тестирования реализаций таких протоколов. Для протоколов, формальная спецификация которых задана в виде контрактной спецификации, разработан метод автоматизированного построения тестовых последовательностей с полностью автоматическим вынесением вердиктов о соответствии наблюдаемого поведения реализации её спецификации. В совокупности разработанные методы позволяют автоматизировать тестирование соответствия реализаций сложных сетевых протоколов их спецификациям.

В рамках предложенного метода тест представляет собой конечный автомат. С каждым переходом автомата сопоставлено определённое тестовое воздействие. При выполнении перехода это воздействие подаётся на тестируемую реализацию, регистрируются реакции реализации и автоматически выносится вердикт о соответствии наблюдаемого поведения спецификации. Обход автомата теста совершается автоматически во время тестирования, алгоритм обхода не зависит от протокола, тестируемой реализации или конкретного теста. Последовательность переходов определяет тестовую последовательность. В силу недетерминизма протоколов и различий в поддержке необязательных функций в реализациях конкретные тестовые последовательности, получаемые при прогоне тестов на разных реализациях, могут не совпадать друг с другом.

Предложенный подход к верификации функций безопасности включает два метода: метод формализации стандартов протоколов и метод формального задания тестовых наборов [15]. Метод формализации стандартов протоколов включает анализ спецификации протокола и извлечение требований, определение формального интерфейса протокола, формализацию функциональных требований к реализации протокола, задание критериев покрытия и разработку функции реконструкции состояния. Метод формализации тестовых последовательностей состоит из определения целей тестирования, разработки проекта автомата теста для конкретной цели тестирования, задания переходов автомата теста, задания функции определения состояния автомата теста по модельному состоянию контрактной спецификации протокола, проектирования настроечной информации автомата теста и разработки формата для представления опций, а также включает прогон тестового сценария и анализ результатов тестирования. Для описания автоматов тестов в данном методе используется специальный вид задания автоматов тестов, который называется тестовым сценарием.

2. Обзор протокола TLS

Протокол TLS разработан на основе спецификации протокола SSL 3.0 (Secure Socket Layer), опубликованной корпорацией Netscape. Ранние версии протокола описаны в RFC 2246 (версия 1.0) [1] и RFC 4346 (версия 1.1) [4]. Последняя модификация TLS версии 1.2 определена в RFC 5246 [10]. Как указано в спецификации, не смотря на то, что различия между протоколами не значительны, TLS и SSL 3.0 несовместимы. Хотя в TLS предусмотрен механизм, позволяющий реализациям TLS общаться с реализациями SSL 3.0.

Протокол TLS состоит из двух уровней: протокола Записей TLS (TLS Record Protocol) и протокола Рукопожатия (TLS Handshake Protocol). На нижнем уровне находится протокол Записей TLS (TLS Record Protocol), работающий поверх некоторого надежного транспортного протокола (например, TCP). Этот протокол обеспечивает конфиденциальность и надежность соединений:

1. Конфиденциальность соединения. Для защиты данных используются алгоритмы симметричного шифрования, ключи для которых уникальны для каждой сессии и создаются на основе секрета, согласованного другими протоколами (например, протоколом Рукопожатия). Протокол Записей TLS также может использоваться без шифрования.
2. Надежность соединения. Целостность сообщений обеспечивается путем вычисления и включения в сообщения кода аутентификации (MAC). Для этого используются криптографические алгоритмы на основе хэш-функций (такие как SHA-1, MD5 и др.). Данный протокол может использоваться без вычисления MAC.

Протокол Записей TLS используется для инкапсуляции протоколов более высокого уровня. Одним из таких протоколов является протокол Рукопожатия, который позволяет серверу и клиенту аутентифицировать друг друга и согласовать параметры безопасности сессии (такие как криптографические алгоритмы и ключи), перед началом использования протокола прикладного уровня. Для аутентификации используются криптографические алгоритмы с открытым ключом (такие как RSA, DSS и др.). Протокол Рукопожатия также может использоваться без аутентификации. Однако обычно она необходима, по крайней мере, для одного из участников.

Одно из преимуществ TLS состоит в том, что он независим от протоколов прикладного уровня. Для протоколов более высокого уровня использование TLS является прозрачным, однако спецификация TLS не определяет схему их взаимодействия. Решение о том, как инициировать TLS-диалог и как интерпретировать сертификаты аутентификации, оставляется на усмотрение разработчиков протоколов, которые работают поверх TLS.

2.1. Протокол Записей TLS

Получив данные для передачи, протокол Записей TLS фрагментирует их на блоки нужной длины, если необходимо сжимает данные, вычисляет MAC, шифрует и передает результат транспортному протоколу. Полученные данные дешифруются, проверяется их целостность, выполняется декомпрессия и дефрагментация, и результат передается протоколу верхнего уровня.

По протоколу TLS происходит обмен блоками данных, называемыми Записи TLS, инкапсулирующими данные протокола верхнего уровня. Каждая Запись TLS содержит поле “Content type”, которое определяет тип протокола верхнего уровня.

Спецификация TLS определяет четыре протокола, работающие поверх TLS: протокол Рукопожатия (Handshake protocol, content type 22), протокол Оповещения (Alert protocol, content type 21), протокол Изменения состояния (Change cipher spec protocol, content type 20) и протокол прикладных данных(content type 23). Другие протоколы могут поддерживаться TLS, однако они должны быть зарегистрированы в соответствующем реестре IANA (Internet Assigned Numbers Authority) ([10], раздел 12).

2.2. Состояние соединения

Состояние соединения TLS определяет набор параметров для работы протокола Записей TLS. К этим параметрам относятся криптографические алгоритмы сжатия, шифрования и вычисления MAC, а также соответствующие ключи. Существуют четыре состояния соединения: текущие состояния чтения и записи и ожидаемые состояния чтения и записи. Для любой обработки данных используются установки текущего состояния. Параметры ожидаемых состояний устанавливаются через протокол Рукопожатия. Протокол Изменения состояния (Change Cipher Spec) заменяет текущее состояние ожидаемым, а ожидаемое состояние инициализируется пустым состоянием. Начальное текущее состояние всегда определяется без использования шифрования, сжатия и вычисления MAC.

2.3. Протокол Рукопожатия

Протокол рукопожатия используется для согласования параметров безопасности соединения.

Полная схема обмена выглядит следующим образом:

```
  Client                                  Server

  ClientHello            -------->
                                         ServerHello
                                         Certificate*

ServerKeyExchange*
```

```
CertificateRequest*
                         <--------      ServerHelloDone
  Certificate*
  ClientKeyExchange
  CertificateVerify*
  [ChangeCipherSpec]
  Finished               -------->

[ChangeCipherSpec]
                         <--------      Finished
  Application Data       <------->      Application Data
```

* необязательные или зависящие от ситуации сообщения, которые посылаются не всегда.

- Клиент посылает сообщение ClientHello, содержащее максимальный номер версии протокола TLS, который он поддерживает, случайное число ClientHello.random, предлагаемые криптографические алгоритмы (Cipher Suite) и метод сжатия (Compression Method).
- Сервер отвечает сообщением ServerHello, содержащим выбранные их предложенных клиентом версию TLS, идентификатор сессии (Session ID), криптографические алгоритмы (Cipher Suite) и метод сжатия (Compression Method),а также свое случайное число ServerHello.random. Выбранная версия TLS должна быть максимальной из поддерживаемых. Сервер также может ответить критической ошибкой и разорвать соединение.

Для обмена ключами используются четыре сообщения: сертификат сервера (server Certificate), ServerKeyExchange, сертификат клиента (client Certificate) и ClientKeyExchange.

- После Hello сервер посылает свой сертификат для аутентификации (если он есть). Дополнительно может быть послано сообщение ServerKeyExchange (если сервер не имеет сертификата или его сертификат служит только для цифровой подписи). Если сервер аутентифицирован, он может запросить сертификат клиента (CertificateRequest).
- Сервер посылает сообщение ServerHelloDone, указывающее, что фаза приветствия завершена. После чего ждет ответа клиента.
- Если был соответствующий запрос сервера, клиент посылает свой сертификат. Затем посылается сообщение ClientKeyExchange, содержимое которого зависит от выбранного в сообщениях ClientHello и ServerHello алгоритма с открытым ключом. Если сертификат клиента используется для цифровой подписи, то посылается подписанное сообщение CertificateVerify для явной проверки подлинности сертификата.
- Клиент посылает сообщение об изменении состояния ChangeCipherSpec, и заменяет текущее состояние ожидаемым.

- В завершение клиент посылает сообщение Finished, защищенное только что согласованными алгоритмами и содержащее код аутентификации (MAC), вычисленный над всеми предыдущими сообщениями обмена.
- Сервер расшифровывает полученное сообщение Finished, проверяет правильность MAC. В случае ошибки соединение разрывается.
- Наконец сервер отвечает своим сообщением изменения состояния ChangeCipherSpec, заменяет текущее состояние ожидаемым и посылает заключительное сообщение Finished, применяя согласованные алгоритмы и ключи.
- Клиент также дешифрует и проверяет последнее сообщение

На этом обмен протокола Рукопожатие считается завершенным, и клиент и сервер могут начать обмен данными прикладного уровня. Сообщения Finished являются первыми сообщениями, защищенными только что согласованными алгоритмами и ключами.

Если клиент и сервер хотят возобновить предыдущую сессию или дублировать существующую (вместо согласования новых параметров безопасности), используется сокращенная схема обмена:

```
  Client                                Server

  ClientHello           -------->
                                        ServerHello

[ChangeCipherSpec]
                        <--------       Finished
  [ChangeCipherSpec]
  Finished              -------->
  Application Data      <------->       Application Data
```

Клиент посылает ClientHello, используя идентификатор сессии (Session ID), которую требуется возобновить. Сервер ищет соответствующий идентификатор сессии в своем кэше сессий. Если идентификатор найден, и политика сервера разрешает возобновить сессию, то сервер посылает ServerHello с этим идентификатором сессии. После этого сразу происходит обмен сообщениями изменения состояния ChangeCipherSpec и завершающими сообщениями Finished. Обмен Рукопожатия считается завершенным, и можно отправлять данные прикладного уровня.

Если соответствующий идентификатор Session ID не найден, сервер создает новый ID, и выполняется полная схема Рукопожатия.

2.4. Протокол Изменения состояния

Протокол состоит из единственного сообщения, которое зашифровано и сжато, в соответствии с параметрами текущего состояния соединения. Сообщение об изменении состояния может посылаться как клиентом, так и

сервером для уведомления партнера о том, что следующие сообщения будут защищены только что согласованными алгоритмами и ключами. При этом получатель данного сообщения должен немедленно заменить текущее состояние чтения ожидаемым состоянием чтения, а отправитель - текущее состояние записи ожидаемым состоянием записи.

2.5. Протокол Оповещения

Сообщение содержит код оповещения (предупреждающее или критическое) и его описание. Критическое сообщение приводит к немедленному закрытию соединения. При этом другие соединения данной сессии могут быть продолжены, но новые соединения в рамках данной сессии создавать запрещено. Сообщения оповещения зашифрованы и сжаты, как определено в текущем состоянии соединения.

2.6. Механизмы расширения протокола TLS

Протокол TLS допускает расширения для добавления новой функциональности. Для согласования расширений используются сообщения ClientHello и ServerHello протокола Рукопожатия. Каждое расширение содержит тип расширения (поле “extension type”) и данные, формат которых зависит от конкретного расширения. Список расширений управляется IANA и доступен по адресу <http://www.iana.org/assignments/tls-extensiontype-values>. Механизм расширений обеспечивает обратную совместимость, т.е. реализации, поддерживающие расширения могут взаимодействовать с теми, которые их не поддерживают. Если конкретного типа расширения не было в запросе ClientHello, он не должен присутствовать в ответном сообщении ServerHello.

Первые расширения (как и сами механизмы расширений) описаны в спецификации RFC 4366. Там же определены два новых сообщения протокола Рукопожатия: CertificateURL и CertificateStatus. Дальнейшие расширения были определены в RFC 4680, 4681, 5077 и RFC 5246. Ниже перечислены некоторые конкретные расширения TLS:

- Указание имени сервера. Тип расширения: server_name. Значение: 0 (RFC 4366).
- Согласование максимального размера фрагментов. Тип расширения: max_fragment_length. Значение: 1 (RFC 4366).
- Универсальные локаторы ресурсов сертификатов клиента. Тип расширения: client_certificate_url. Значение: 2 (RFC 4366).
- Указание доверенного центра сертификации. Тип расширения: trusted_ca_keys. Значение: 3 (RFC 4366).
- Усеченный HMAC. Тип расширения: truncated_hmac. Значение: 4 (RFC 4366).
- Запрос статуса сертификата. Тип расширения: status_request. Значение: 5 (RFC 4366).

- Отображение пользователей. Тип расширения: user_mapping. Значение: 6 (RFC 4681).
- Алгоритмы цифровой подписи. Тип расширения: signature_algorithms. Значение: 13 (RFC 5246).
- Возобновление сессии без сохранения состояния на сервере. Тип расширения: SessionTicket TLS. Значение: 35 (RFC 5077).
- Указание повторного согласования. Тип расширения: renegotiation_info. Значение: 65281 (RFC 5746).

3. Формальная спецификация TLS

Подробное описание модели тестирования приведено в [16,17]. В соответствии с этой моделью тестирования формальная спецификация TLS состоит из нескольких компонентов:

- модельного состояния, которое содержит набор структур данных, моделирующих концептуальные структуры данных из стандартов TLS;
- формального интерфейса TLS, включающего спецификационные стимулы, формализующие требования к изменению состояния реализации TLS при внешнем воздействии на систему и спецификационные реакции, которые формализуют требования к реакциям реализации TLS на внешние воздействия;
- критериев покрытия, идентифицирующих различные ветви функциональности TLS.

3.1. Модельное состояние

Модельное состояние представлено множеством TLS-соединений и множеством TLS-сессий на узле, моделирующем состояние целевой реализации.

TLS-соединение содержит параметры безопасности конкретного соединения, такие как криптографические алгоритмы, ключи. TLS-сессия содержит данные, необходимые для повторного использования согласованных ранее параметров безопасности, такие как сертификаты, криптографические алгоритмы, мастер-ключ (master secret).

Полное описание этих структур приведено в RFC 5246.

Рассмотрим, как эти концептуальные структуры данных представлены в формальной спецификации.

Модельный тип TLS-соединения включает следующие блоки данных:

selector	селекторы трафика (адреса и порты TCP соединения), являющиеся идентификатором соединения

current read state	текущее состояние чтения
current write state	текущее состояние записи
pending read state	ожидаемое состояние чтения
pending write state	ожидаемое состояние записи

Для каждого соединения TLS определяются четыре состояния: текущие состояния чтения и записи и ожидаемые (pending) состояния чтения и записи. Все сообщения обрабатываются, используя параметры текущего состояния. Параметры ожидаемых состояний устанавливаются с помощью обмена рукопожатия (TLS Handshake Protocol).

Модельный тип TLS состояния содержит следующие структуры данных:

sessionID	идентификатор сессии, соответствующей данному соединению
keys	ключи криптографических алгоритмов
sequence number	порядковый номер сообщений
security parameters	параметры безопасности

Модельный тип параметров безопасности содержит следующие поля:

connectionEnd	данный флаг определяет, является узел сервером или клиентом для данного соединения
prf_algorithm	алгоритм, используемый для создания криптографических ключей из мастер-ключа
bulk_cipher_algorithm	алгоритм шифрования и необходимые параметры
enc_key_length	
block_length	
fixed_iv_length	
record_iv_length	
mac_algorithm	алгоритм защиты целостности сообщений и необходимые параметры
mac_length	
mac_key_length	
compression_algorithm	алгоритм сжатия

master_secret	мастер-ключ
client_random	одноразовые массивы байт клиента и сервера
server_random	

Модельный тип TLS-сессии включает:

id	идентификатор сессии
peer_certificate	сертификат партнера, если такой используется
compression_method	метод сжатия
cipher_spec	криптографические алгоритмы
master_secret	мастер-ключ
isResumable	данный флаг определяет, может ли сессия использоваться для инициализации новых соединений

3.2. Модель TLS-сообщений

Протокол TLS для передачи данных использует структуры, называемые TLS-записями (TLS Records), инкапсулирующие весь TLS-трафик. Спецификация определяет четыре типа передаваемых данных:

- Обмен Рукопожатия (TLS Handshake Protocol), который используется для согласования новых параметров безопасности;
- Протокол изменения состояния (Change Cipher Spec Protocol), единственное сообщение ChangeCipherSpec заменяет параметры текущего состояния параметрами соответствующего ожидаемого состояния;
- Протокол Оповещения (Alert Protocol), предназначенный для передачи информационных сообщений и сообщений об ошибках;
- Данные протокола верхнего уровня, использующего TLS в качестве транспорта.

Каждое TLS-сообщение может содержать данные только одного из перечисленных выше типов, однако структур данных этого типа может быть несколько (например, TLS-сообщение может содержать несколько сообщений протокола Рукопожатия). Тип данных указывается в заголовке TLS-записи.

Для модельного представления TLS-сообщений разработана библиотека соответствующих спецификационных типов, позволяющая моделировать различные варианты сообщений.

3.3. Спецификационные функции

В спецификации каждое входящее TLS-сообщение рассматривается как последовательность стимулов. Каждый стимул в этой последовательности соответствует обработке отдельного блока данных в TLS-сообщении (напомним, что TLS-сообщение может содержать несколько структур данных конкретного типа).

Предусловие каждого стимула проверяет допустимость отправки сообщения, правильность значений полей блоков данных и прогнозирует ожидаемую реакцию целевой системы.

Каждое исходящее TLS-сообщение рассматривается как последовательность реакций. Каждая реакция в этой последовательности соответствует отдельному блоку данных в TLS-сообщении.

Постусловия реакций проверяют допустимость получения конкретного сообщения, и правильность значений полей блоков данных.

4. Тестирование реализаций TLS

4.1. Каталог функциональных требований к реализации протокола TLS

Спецификация протокола TLS написана на естественном языке. Для тестирования реализации на соответствие необходимо выделить из стандарта отдельные требования и затем их формализовать. В результате анализа текста стандарта был составлен полный список требований (около 300 требований). Эти требования разбиты на несколько групп:

1. Требования, относящиеся к протоколу записей TLS (TLS Record Protocol);
2. Требования, относящиеся к протоколам рукопожатия (TLS Handshaking Protocols);
3. Требования, относящиеся к правилам формирования ключевого материала.

4.2. Устройство тестового стенда и тестовый набор

В состав тестового стенда входят инструментальный узел и целевой узел. На инструментальном узле исполняется основной поток управления тестовой системы. На целевом узле функционирует тестируемая реализация. Инструментальный и целевой узлы могут располагаться в разных сегментах сети.

Стимулами в разработанном тестовом наборе являются сообщения от инструментального узла, а реакциями - сообщения со стороны тестируемого узла. Основная часть требований спецификации TLS проверяется в постусловиях реакций.

4.3. Тестирование TLS-сервера

В роли TLS-сервера реализация не генерирует запросы, а лишь поддерживает информационный обмен, инициированный другим узлом. Стимулами являются сообщения от инструментального узла.

В тестовом сценарии создается TLS-соединение, в рамках которого формируются запросы в модельном представлении, передаваемые затем функции отправки сообщений. В предусловии спецификационных функций стимулов проверяется правильность структуры тестового сообщения и его своевременность, и на основании этого делается вывод о том, должен ли на него быть ответ, сообщение об ошибке или реализация должна его проигнорировать. Из модельного представления тестового сообщения строится реализационное, которое и отправляется в сеть.

Сборщик реакций в течение заданного времени собирает ответные сообщения целевой системы. Из реализационных TLS сообщений строятся их модельные представления. Последовательность блоков данных в полученных сообщениях рассматривается как последовательность реакций целевой системы.

В постусловии реакций данные проверяются на соответствие требованиям спецификации. Проверка разделена на несколько стадий. Сначала проверяется допустимость такого сообщения от реализации и его своевременность, затем - структура самого сообщения (присутствующие поля и их значения должны соответствовать текущему обмену).

После проверки всех требований, результат передается тестовому сценарию, где в зависимости от плана сценария, принимается решение о продолжении или завершении информационного обмена. В случае выявления нарушения требований принимается решение о критичности ошибки и возможности отправки следующих запросов.

В случае успешного завершения обмена рукопожатия, создается новая TLS-сессия (за исключением сокращенного варианта обмена, в котором используется уже существующая сессия), параметры которой могут в дальнейшем использоваться для инициализации других TLS соединений.

4.4. Выбор реализации для тестирования

Для тестирования на соответствие стандарту были выбраны следующие реализации:

1. Почтовый сервер Postfix.2.9.3 с открытой реализацией протокола TLS openssl.1.0.1c под управлением операционной системы Red Hat Enterprise Linux 5.5;
2. Реализация TLS в виртуальной машине Java 1.7.0_05 (Java Secure Socket Extension);
3. Тестовый сервер TLS интернет ресурса https://www.mikestoolbox.net.

4.5. Результаты тестирования реализаций TLS в роли сервера

При выполнении тестового набора был выявлен ряд особенностей, нарушений требований RFC 5246 и шибок реализаций. Ниже дано описание этих особенностей.

1. Почтовый сервер Postfix.2.9.3:

- при получении сообщения оповещения (Alert) CLOSE_NOTIFY сервер разрывает соединение без отправки ответного уведомления;
- если вслед за сообщением ChangeCipherSpec клиент отправляет любое оповещение (Alert), сервер отвечает ошибкой UNEXPECTED_MESSAGE (неожиданное сообщение);
- если в сообщении ClientHello присутствуют дубликаты неизвестных серверу расширений, сервер их игнорирует и продолжает обмен данными;
- в сообщении ClientHello при наличии расширений присутствуют несколько полей длины: длина сообщения, общая длина расширений, длина каждого расширения. Сервер не проверяет поле общей длины расширений;
- если в TLS сообщении после блока ClientHello имеются какие-то неправильные данные (например неизвестный заголовок), сервер сначала отвечает на ClientHello последовательностью сообщений ServerHello, ServerCertificate, ServerHelloDone и только затем отправляет сообщение об ошибке и разрывает соединение;
- согласно спецификации размер каждого TLS-сообщения не должен превышать (2^14 + 2048) байт. Реализация принимает и отвечает на такие сообщения;
- если, после завершения обмена рукопожатия, не разрывая соединения, отправить сообщение ClientHello, сервер ответит на него не критичным (т.е. без разрыва соединения) сообщением Alert NO_RENEGOTIATION. После этого на любые сообщения протокола Рукопожатия и сообщения Alert (в том числе критичные, которые должны приводить к закрытию соединения) сервер отвечает оповещением NO_RENEGOTIATION, при этом продолжая принимать, обрабатывать и отвечать на сообщения с прикладными данными (APPLICATION data). Если клиент отправляет в прикладном сообщении команду “quit” (команда завершения сеанса связи с почтовым сервером протокола SMTP), сервер корректно отвечает на нее и отправляет завершающее сообщение Alert CLOSE_NOTIFY. Однако на ответное сообщение Alert CLOSE_NOTIFY (как предусмотрено спецификацией) сервер также отвечает сообщением Alert NO_RENEGOTIATION.

2. Реализация TLS в виртуальной машине Java 1.7.0_05:

- реализация игнорирует дубликаты расширений в сообщении ClientHello;
- реализация не проверяет поле длины во входящих сообщениях протокола Рукопожатия (по крайней мере в сообщениях ClientHello, ClientKeyExchange, Finished);
- в сообщении ClientHello при наличии расширений присутствуют несколько полей длины: длина сообщения, общая длина расширений, длина каждого расширения. Сервер не проверяет поле общей длины расширений.

3. Тестовый сервер TLS интернет ресурса https://www.mikestoolbox.net:

- если в обмене рукопожатия используется алгоритм RSA, клиент создает и отправляет серверу 48-байтный предварительный ключ (premaster secret), два первых байта которого содержат версию протокола TLS из предшествующего сообщения ClientHello. Сервер должен проверить правильность номера версии. Данная реализация не выполняет эту проверку. Данное требование предназначено для противодействия атакам rollback и является критичным.
- после установления соединения, в ответ на сообщения протокола Рукопожатия отличные от ClientHello, сервер возвращает сообщение об ошибке, при этом версия протокола в TLS сообщении установлена 3.0.
- если в сообщении ClientHello значение поле CipherSuite (криптографический набор) равно TLS_NULL_WITH_NULL_NULL, или в сообщении присутствуют дубликаты расширений, сервер, как и требуется, возвращает ошибку, однако TLS-запись содержит два одинаковых сообщения об ошибке.

Тестирование реализаций на соответствие требованиям спецификации RFC 5746 (TLS Renegotiation Indication Extension)

Данный документ рекомендует отказаться от стандартной схемы переустановки TLS соединений (RFC 5246) и использовать только предложенную безопасную схему.

1. Почтовый сервер Postfix.2.9.3:

- поддерживает только безопасную переустановку TLS соединений.

2. Реализация TLS в виртуальной машине Java 1.7.0_05:

- с настройками по умолчанию поддерживает только безопасную переустановку TLS соединений, дополнительные настройки позволяют использовать стандартный вариант переустановки.

3. Тестовый сервер TLS интернет ресурса https://www.mikestoolbox.net:

- поддерживает оба варианта переустановки TLS соединений.

Следует отметить, что, несмотря на некоторые особенности и нарушения, реализации в целом соответствуют спецификации. Однако вторая реализация нарушает критичное требование проверки версии протокола TLS в сообщении ClientKeyExchange с использованием алгоритма RSA.

5. Заключение

В ходе выполнения работы были выделены требования к реализациям TLS, разработаны формальные спецификации обработки входящих и исходящих сообщений протокола TLS, медиаторы для доставки тестовых воздействий на целевую реализацию, а также тестовые сценарии для проверки обработки входящих и исходящих сообщений подсистемой TLS. В статье описаны также результаты тестирования существующих реализаций.

Данная работа показала, что разработанный метод верификации, основанный на контрактных спецификациях, позволяет эффективно автоматизировать тестирование таких сложных протоколов, как протоколы безопасности. При этом тестовые наборы обладают формально определенным и прослеживаемым покрытием требований, что в значительной степени улучшает качество тестирования.

Список литературы

[1] IETF RFC 2246. Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", January 1999.
[2] IETF RFC 3268. Chown, P., "Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)", June 2002.
[3] IETF RFC 3546. Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", June 2003.
[4] IETF RFC 4346. Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", April 2006.
[5] IETF RFC 4366. Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", April 2006.
[6] IETF RFC 4507. Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", May 2006.
[7] IETF RFC 4680. Santesson, S., "TLS Handshake Message for Supplemental Data", October 2006.
[8] IETF RFC 4681. Santesson, S., Medvinsky, A., and J. Ball, "TLS User Mapping Extension", October 2006.
[9] IETF RFC 5077. Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", January 2008.
[10] IETF RFC 5246. Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", August 2008.

[11] IETF RFC 5746. E. Rescorla, M. Ray, S. Dispensa, N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. February 2010.
[12] Bourdonov I., Kossatchev A., Kuliamin V., Petrenko A. UniTesK Test Suite Architecture // Proceedings of FME, LNCS 2391. Springer-Verlag, 2002. P. 77-88.
[13] Bourdonov I.B., Demakov A.V., Jarov A.A., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. and Zelenov S.V. Java Specification Extension for Automated Test Development // Proceedings of PSI'2001. Novosibirsk, Russia July 2-6 2001, LNCS 2244:301-307. Springer-Verlag, 2001.
[14] Н.В. Пакулин. Формализация стандартов и тестовых наборов протоколов Интернета. Автореферат диссертации на соискание учёной степени кандидата физико-математических наук. Москва, 2006.
[15] Н.В. Пакулин, А.В. Хорошилов "Разработка формальных моделей и тестирование соответствия для систем с асинхронными интерфейсами и телекоммуникационных протоколов", Журнал "Программирование" № 5, 2007 г., ISSN 0132-3474, с. 1-29.
[16] А.В. Никешин, Н.В. Пакулин, В.З. Шнитман "Разработка тестового набора для верификации реализаций протокола безопасности IPsec v2", Труды Института системного программирования РАН, т. 18, 2010, стр. 151-182.
[17] А.В. Никешин, Н.В. Пакулин, В.З. Шнитман "Верификация функций безопасности протокола IPsec v2", Журнал "Программирование" № 1, 2011, стр. 36-56.

Верификация драйверов операционной системы Linux[1]

Бейер Д. (Университет Пассау, Германия), Петренко А. К. (ИСП РАН)
dirk.beyer@sosy-lab.org, petrenko@ispras.ru

Аннотация. Верификация драйверов ОС Linux – это широкая область для применения различных методов верификации, в частности, методов проверки свойств безопасности и надежности программ, а также функциональной верификации. Драйверы Linux – это индустриальное программное обеспечение, на стабильность которого полагаются ИТ-инфраструктуры. В силу этого к надежности и корректности их работы предъявляют жесткие требования, в свою очередь, это означает, что если инженер-верификатор обнаружил ошибку в драйвере, он может рассчитывать на быструю реакцию сообщества разработчиков в плане подтверждения и исправления этой ошибки. Драйверы Linux – сложное низкоуровневое системное программное обеспечение, и его характеристики требуют применения различных техник анализа программ: использования SMT-решателей, методов верификации моделей (model checking) и других методов верификации. Точность и эффективность этих методов за последнее время значительно повысилась, и поэтому сложная задача верификации драйверов ОС Linux становится реальной по мере использования этих достижений в инструментах верификации.

Ключевые слова. Ядро операционной системы, Linux, инструменты верификции, анализ программ, верификация моделей, анализ указателей, анализ структур данных, свойства безопасности, ограничиваемая проверка моделей, символьная верификация, анализ с явными значениями, условная верификация моделей, анализ завершимости, верификация параллельных программ, сравнение методов верификации.

1. Введение

Ядро ОС Linux в настоящее время является одной из наиболее важных программных систем в нашем обществе. Linux используется в качестве ядра в ряде распространенных настольных операционных систем (таких как Ubuntu, Fedora, Debian, Gentoo), и поэтому в надежности и стабильности ядра заинтересовано большое число пользователей этих систем. Возможно, еще

[1] Данная работа была .поддержана Государственным контрактом с Министерством образования и науки РФ № 11.519.11.4006 от 18 августа 2011 г., программное мероприятие 1.9 «Проведение научно-исследовательских работ совместно с иностранными научными организациями».

важнее то, что доминирующие на рынке серверные операционные системы основаны на Linux. Практически все суперкомпьютеры (90% в 2010 году) работают под управлением операционной системы на основе этого ядра. Все большее число встраиваемых устройств, таких как смартфоны, используют Linux в качестве ядра (например, Android, Maemo, WebOS). Все это объясняет возрастающую потребность в автоматической верификации компонентов операционной системы Linux.

Компания Microsoft установила, что ошибки в драйверах устройств являются одной из наиболее важных причин сбоев в работе выпускаемых ею операционных систем. Как следствие, компания существенно увеличила надежность Windows, интегрировав инструмент Static Driver Verifier (SDV) в свой производственный цикл. Основные принципы, лежащие в основе работы этого инструмента, были разработаны в рамках исследовательского проекта SLAM [1]. Сейчас SDV по умолчанию входит в состав набора инструментов для разработки драйверов ОС Windows – Windows Driver Kit (WDK).

Для ОС Linux в настоящее время не существует индустриального проекта по верификации, сопоставимого по размерам с SDV. Но сообщество разработчиков постоянно ищет средства автоматической верификации наиболее критичных аспектов работы этой ОС, а сообщество исследователей в области верификации использует драйверы Linux как область для применения новых подходов анализа. За последние годы было создано три среды верификации на основе драйверов ОС Linux: Linux Driver Verification (LDV)[2] [23, 31], Avinux [27] и DDVerify[3] [34].

Код ядра Linux является распространенным источником задач верификации [17, 22, 24, 25, 30]. Драйверы Linux предоставляют уникальное сочетание характеристик, которые привлекают исследователей и практиков возможностью испытать на них свои инструменты. Наиболее важные преимущества использования кода ядра Linux для постановки задач верификации следующие:

- Данное программное обеспечение является востребованным – многие люди заинтересованы в результатах его верификации.
- Любая ошибка в драйвере является потенциально критической, потому что драйверы работают с тем же уровнем привилегий, что и остальное ядро ОС Linux, в адресном пространстве ядра.
- Суммарный объем исходного кода драйверов весьма значителен (около 10 млн. строк) и при этом постоянно возрастает.
- Задачи верификации драйверов достаточно трудны и интересны, но не настолько сложны, чтобы быть безнадежными.

[2] http://linuxtesting.org/project/ldv
[3] http://www.cprover.org/ddverify

- Большинство драйверов Linux имеют открытый исходный код и поэтому их легко использовать в проектах по верификации и в исследовательских проектах.

Хотя в области верификации программного обеспечения было сделано много новых достижений, требуются дополнительные усилия для того, чтобы использовать их на практике и применять их к сложному индустриальному коду, такому как драйверы устройств ОС Linux. Недавно проведенное соревнование по верификации программного обеспечения (SV-COMP'12)[4] [3] показало, что категория драйверов устройств представляет сложность даже для самых современных инструментов анализа программ.

2. Направления исследований

2.1. Анализ указателей и динамические структуры данных

Многие свойства надежности и безопасности драйверов устройств зависят от точного анализа указателей и различных структур данных в куче. Анализ указателей изучен достаточно хорошо, однако использование низкоуровневого кода, характерное для системного программирования, существенно осложняет реализацию существующих подходов. Проект LDV достиг существенных успехов в этой области благодаря реализации более точного анализа указателей в инструменте верификации BLAST [29]. Улучшенная версия оригинального инструмента BLAST стала победителем соревнований SV-COMP'12 на задачах верификации, полученных из драйверов ОС Linux [28].

Анализ структур данных является актуальной исследовательской темой, по которой в последние годы были получены заметные результаты. Следует отметить, что достаточно большого набора общедоступных тестовых задач для практического сравнения существующих реализаций не существует. Одним из примеров является инструмент Predator – современный статический анализатор, предоставляющий средства проверки структур данных и безопасности работы с памятью [16].

2.2. Символьная верификация

Благодаря развитию SMT-решателей, символьное представление абстрактных состояний программы на основе формул в настоящее время стало эффективным и результативным. Инструменты SDV и SLAM компании Microsoft [1], а также ряд современных академических инструментов основаны на предикатной абстракции [5, 8, 12, 18]. Несколько инструментов сочетают такие концепции, как уточнение абстракции по контрпримерам (counterexample-guided abstract refinement, CEGAR) [11], различные виды

[4] http://sv-comp.sosy-lab.org

анализа динамической памяти (так называемого шейп-анализа, shape analysis), абстрактные деревья достижимости [5], ленивую абстракцию [21], интерполяцию [20] и крупноблочное кодирование (large-block encoding) [4, 9]. Ограничиваемая проверка моделей (bounded model checking) [10] также имеет большое практическое значение и показывает впечатляющие результаты на соревнованиях по верификации [14, 32].

Недостаточно исследованными остаются проблемы определения интерполянтов (существует большой диапазон между слабыми и сильными интерполянтами), проблемы размеров блоков для кодирования (какой критерий нужно использовать для определения конца блока, который целиком кодируется в формулу в ходе одной операции вычисления абстрактного постусловия) и проблемы выбора порядка обхода графа потока управления программы (нацеленность на максимизацию покрытия (coverage-directed verification), поиск в ширину, в глубину и др.). Другим важным и многообещающим подходом, который до недавнего времени в значительной степени игнорировался в верификации программного обеспечения, является возможность представления абстрактных состояний и отношения переходов полностью в виде двоичных решающих диаграмм (Binary Decision Diagrams, BDD). Некоторый успех в этой области наблюдается, например, в опыте расширения инструментов CPAchecker и Java PathFinder механизмами использования BDD для представления пространства состояний, которое образуют булевы переменные, задающие параметры построения конфигураций, например, операционных систем и других систем, образующих так называемые product lines [33].

2.3. Анализ с явными значениями (Explicit-State Verification)

Некоторые инструменты проверки моделей программ с явными значениями являются достаточно успешными в своих областях применения (например, Spin и Java PathFinder). Чтобы применить эту технологию для верификации драйверов операционных систем масштабируемым образом, представляется интересным включить в нее современные успешные подходы из области символьной верификации. Например, CEGAR должен быть использован для автоматического построения абстракции, а интерполяция Крейга для явных значений может указать, какие части пространства состояний необходимо анализировать.

2.4. Комбинация приемов верификации

В прошлом было предложено несколько методов параллельной комбинации различных существующих видов анализа программ [7, 15]. Эти методы чрезвычайно эффективны и должны получить большее внимание и воплощение на практике. Их практическому применению препятствуют технические трудности: объединяемые виды анализа должны использовать одинаковый алгоритм обхода графа потока управления, они должны быть

реализованы на одном и том же языке программирования в совместимых средах и должны работать на одной и той же физической машине.

Последовательная комбинация различных видов анализа с использованием условной верификации моделей (conditional model checking) представляет эффективное решение этой проблемы [6]. Разные инструменты и методы могут выполняться последовательно один за другим и пытаться решить задачу верификации, используя свои наиболее сильные стороны. Инструменту условной верификации моделей сообщается, когда нужно отказаться от продолжения верификации (при помощи так называемого входного условия, input condition). Входные условия дают гибкий способ ограничения или сдерживания хода верификации одним из выбранных методов. Выходные условия предоставляют успешно верифицированную часть пространства состояний. Следующий верификатор может использовать условия, полученные от предыдущих запусков других инструментов, чтобы не выполнять повторно ту же самую работу, а сосредоточить свои усилия на оставшейся части задания.

2.5. Анализ завершимости

Область верификации, требующая большего внимания исследователей, – это анализ завершимости. Существует несколько инструментов проверки завершимости программ (наиболее заметным среди которых является ARMC [26]), но метод пока еще не настолько широко распространен, как это следовало бы ожидать. Подход был заимствован и в дальнейшем улучшен компанией Microsoft в рамках проекта Terminator [13].

2.6. Параллелизм (Concurrency)

В силу возрастающего распространения многоядерных компьютеров, верификация многопоточного программного обеспечения со временем становится все более важной областью исследований. Поиск гонок и взаимных блокировок является неотъемлемой частью средств контроля качества. Поэтому данные виды проверок необходимо применять в том числе и к драйверам ОС Linux. Сообщество исследователей в области верификации активно открывает новые концепции и реализует новые инструменты для решения этой задачи (например, ESBMC [14], SATabs [2], Threader [19]). Интересно заметить, что наилучшим инструментом проверки задач на параллелизм на последнем соревновании оказался инструмент ограничиваемой проверки моделей [14].

3. Заключение

Мы обрисовали основные аргументы в пользу рассмотрения драйверов операционной системы Linux в качестве области для практического применения различных методов и инструментов верификации программ. Важно разрабатывать инструменты верификации, которые достаточно

эффективны и результативны для того, чтобы успешно справиться с задачей проверки программных компонентов, столь сложных, как драйверы устройств. Выгоды двойные: обществу важно получать такое ответственное программное обеспечение верифицированным, а сообществу исследователей в области верификации важно получать практические задачи по верификации для дальнейшей разработки и улучшения своих методов. Мы представили обзор современного состояния дел в области верификации и указали направления исследований, больше всего нуждающиеся в их дальнейшем развитии.

Литература

[1] T. Ball, S. K. Rajamani. The Slam Project: Debugging System Software via Static Analysis // Proc. POPL, pp. 1–3. ACM (2002)

[2] G. Basler, A. Donaldson, A. Kaiser, D. Kröning, M. Tautschnig, T. Wahl. SATABS: A Bit-Precise Verifier for C Programs // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 552–555. Springer, Heidelberg (2012)

[3] D. Beyer. Competition on Software Verification // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012)

[4] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, R. Sebastiani. Software Model Checking via Large-block Encoding // Proc. FMCAD, pp. 25–32. IEEE (2009)

[5] D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar. The Software Model Checker Blast // Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)

[6] D. Beyer, T. A. Henzinger, M. E. Keremoglu, P. Wendler. Conditional Model Checking: A Technique to Pass Information Between Verifiers // Proc. FSE. ACM (2012)

[7] D. Beyer, T. A. Henzinger, G. Theoduloz. Program Analysis with Dynamic Precision Adjustment // Proc. ASE, pp. 29–38. IEEE (2008)

[8] D. Beyer, M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification // G. Gopalakrishnan, S. Qadeer (eds.). CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)

[9] D. Beyer, M. E. Keremoglu, P. Wendler. Predicate Abstraction with Adjustable-Block Encoding // Proc. FMCAD, pp. 189–197. FMCAD (2010)

[10] A. Biere, A. Cimatti, E. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. // W. R. Cleaveland (ed.). TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking // J. ACM 50(5), 752–794 (2003)

[12] E. Clarke, D. Kröning, N. Sharygina, K. Yorav. SatAbs: SAT-Based Predicate Abstraction for ANSI-C // N. Halbwachs, L. D. Zuck (eds.). TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)

[13] B. Cook, A. Podelski, A. Rybalchenko. Terminator: Beyond Safety // T. Ball, T., R. B. Jones (eds.). CAV 2006. LNCS, vol. 4144, pp. 415–418. Springer, Heidelberg (2006)

[14] L. Cordeiro, J. Morse, D. Nicole, B. Fischer. Context-Bounded Model Checking with ESBMC 1.17 // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 534–537. Springer, Heidelberg (2012)

[15] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival. Combination of Abstractions in the ASTREE Static Analyzer // M. Okada, I. Satoh (eds.). ASIAN 2006. LNCS, vol. 4435, pp. 272–300. Springer, Heidelberg (2008)

[16] K. Dudka, P. Müller, P. Peringer, T. Vojnar. Predator: A Verification Tool for Programs with Dynamic Linked Data Structures // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 545–548. Springer, Heidelberg (2012)
[17] A. Galloway, G. Lüttgen, J. T. Mühlberg, R. I. Siminiceanu. Model-Checking the Linux Virtual File System // N. D. Jones, M. Müller-Olm (eds.). VMCAI 2009. LNCS, vol. 5403, pp. 74–88. Springer, Heidelberg (2009)
[18] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, A. Rybalchenko. HSF(C): A Software Verifier Based on Horn Clauses // C. Flanagan, B. Koenig (eds.). TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012)
[19] A. Gupta, C. Popeea, A. Rybalchenko. Threader: A Constraint-Based Verifier for Multi-threaded Programs // G. Gopalakrishnan, S. Qadeer (eds.). CAV 2011. LNCS, vol. 6806, pp. 412–417. Springer, Heidelberg (2011)
[20] T. A. Henzinger, R. Jhala, R. Majumdar, K. L. McMillan. Abstractions from Proofs // Proc. POPL, pp. 232–244. ACM (2004)
[21] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Lazy Abstraction // Proc. POPL, pp. 58–70. ACM (2002)
[22] A. Khoroshilov, V. Mutilin, E. Novikov, P. Shved, A. Strakh. Towards an Open Framework for C Verification Tools Benchmarking // E. Clarke, I. Virbitskaite, A. Voronkov (eds.). PSI 2011. LNCS, vol. 7162, pp. 179–192. Springer, Heidelberg (2012)
[23] A. Khoroshilov, V. Mutilin, A. Petrenko, V. Zakharov. Establishing Linux Driver Verification Process // A. Pnueli, I. Virbitskaite, A. Voronkov (eds.). PSI 2009. LNCS, vol. 5947, pp. 165–176. Springer, Heidelberg (2010)
[24] J. T. Mühlberg, G. Lüttgen. Blasting Linux Code // L. Brim, B. R. Haverkort, M. Leucker, J. van de Pol (eds.). FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 211–226. Springer, Heidelberg (2007)
[25] W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, F. Piessens. Sound Formal Verification of Linux's USB BP Keyboard Driver // A. E. Goodloe, S. Person (eds.). NFM 2012. LNCS, vol. 7226, pp. 210–215. Springer, Heidelberg (2012)
[26] A. Podelski, A. Rybalchenko. Transition Predicate Abstraction and Fair Termination // Proc. POPL, pp. 132–144. ACM (2005)
[27] H. Post, C. Sinz, W. Küchlin. Towards Automatic Software Model Checking of Thousands of Linux Modules — A Case Study with Avinux // Softw. Test., Verif. Reliab. 19(2), 155–172 (2009)
[28] P. Shved, M. Mandrykin, V. Mutilin. Predicate Analysis with BLAST 2.7 // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg (2012)
[29] П. Е. Швед, В. С. Мутилин, М. У. Мандрыкин. Опыт развития инструмента статической верификации BLAST // Программирование. 2012. Т. 3. с. 24–35.
[30] М. У. Мандрыкин, В. С. Мутилин, Е. М. Новиков, А. В. Хорошилов, П. Е. Швед. Использование драйверов устройств операционной системы Linux для сравнения инструментов статической верификации // Программирование. 2012. Т. 5. С. 54–71.
[31] В. С. Мутилин, Е. М. Новиков, А. В. Страх и др. Архитектура Linux Driver Verification // Труды Института системного программирования РАН. 2011. Т. 20. С. 163–187.
[32] C. Sinz, F. Merz, S. Falke. LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 542–544. Springer, Heidelberg (2012)

[33] A. von Rhein, S. Apel, F. Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code // Proc. Java Pathfinder Workshop (2011)
[34] T. Witkowski, N. Blanc, D. Kröning, G. Weissenbacher. Model Checking Concurrent Linux Device Drivers // Proc. ASE, pp. 501–504. ACM (2007)

Linux Driver Verification

D. Beyer (University of Passau, Germany) and
A. K. Petrenko (ISP RAS, Moscow, Russia)
dirk.beyer@sosy-lab.org, petrenko@ispras.ru

Abstract. Linux driver verification is a large application area for software verification methods, in particular, for functional, safety, and security verification. Linux driver software is industrial production code — IT infrastructures rely on its stability, and thus, there are strong requirements for correctness and reliability. Linux driver software is complex, low-level systems code, and its characteristics make it necessary to bring to bear techniques from program analysis, SMT solvers, model checking, and other areas of software verification. These areas have recently made a significant progress in terms of precision and performance, and the complex task of verifying Linux driver software can be successful if the conceptual state-of-the-art becomes available in tool implementations.

The paper is based on experience of the research groups led by authors in verification of industrial software. It is important to develop verification tools that are efficient and effective enough to successfully check software components that are as complex as device drivers. In this area verifiers/researchers and Linux society find mutual benefits in cooperation because: for the society it is important to get such crucial software verified; for the verification community it is important to get realistic verification tasks in order to tune and further develop the technology. The paper provides an overview of the state-of-the-art and pointed out research directions in which further progress is essential. In particularly the paper considers most promising verification techniques and tools, including predicate abstraction, counter example generation, explicit-state verification, termination and concurrency analysis. One of main topic of Linux driver verification research is combination of verification techniques.

Keywords: Kernel, Linux, verifier, program analysis, model checking, pointer analysis, shape analysis, safety properties, bounded model checking, symbolic model checking, explicit-state verification, conditional model checking, termination analysis, concurrent program verification, verification tool benchmarking.

References

[1]. T. Ball, S. K. Rajamani. The Slam Project: Debugging System Software via Static Analysis. Proc. POPL, pp. 1–3. ACM (2002).
[2]. G. Basler, A. Donaldson, A. Kaiser, D. Kröning, M. Tautschnig, T. Wahl. SATABS: A Bit-Precise Verifier for C Programs. C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 552–555. Springer, Heidelberg (2012).
[3]. D. Beyer. Competition on Software Verification. C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012).
[4]. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, R. Sebastiani. Software Model Checking via Large-block Encoding. Proc. FMCAD, pp. 25–32. IEEE (2009).
[5]. D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar. The Software Model Checker Blast. Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007).
[6]. D. Beyer, T. A. Henzinger, M. E. Keremoglu, P. Wendler. Conditional Model Checking: A Technique to Pass Information Between Verifiers. Proc. FSE. ACM (2012).

[7]. D. Beyer, T. A. Henzinger, G. Theoduloz. Program Analysis with Dynamic Precision Adjustment. Proc. ASE, pp. 29–38. IEEE (2008).
[8]. D. Beyer, M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. G. Gopalakrishnan, S. Qadeer (eds.). CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011).
[9]. D. Beyer, M. E. Keremoglu, P. Wendler. Predicate Abstraction with Adjustable-Block Encoding. Proc. FMCAD, pp. 189–197. FMCAD (2010).
[10]. A. Biere, A. Cimatti, E. Clarke, Y. Zhu. Symbolic Model Checking without BDDs.. W. R. Cleaveland (ed.). TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999).
[11]. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. J. ACM 50(5), 752–794 (2003).
[12]. E. Clarke, D. Kröning, N. Sharygina, K. Yorav. SatAbs: SAT-Based Predicate Abstraction for ANSI-C. N. Halbwachs, L. D. Zuck (eds.). TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005).
[13]. B. Cook, A. Podelski, A. Rybalchenko. Terminator: Beyond Safety. T. Ball, T., R. B. Jones (eds.). CAV 2006. LNCS, vol. 4144, pp. 415–418. Springer, Heidelberg (2006).
[14]. L. Cordeiro, J. Morse, D. Nicole, B. Fischer. Context-Bounded Model Checking with ESBMC 1.17. C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 534–537. Springer, Heidelberg (2012).
[15]. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival. Combination of Abstractions in the ASTREE Static Analyze. M. Okada, I. Satoh (eds.). ASIAN 2006. LNCS, vol. 4435, pp. 272–300. Springer, Heidelberg (2008).
[16]. K. Dudka, P. Müller, P. Peringer, T. Vojnar. Predator: A Verification Tool for Programs with Dynamic Linked Data Structures. C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 545–548. Springer, Heidelberg (2012).
[17]. A. Galloway, G. Lüttgen, J. T. Mühlberg, R. I. Siminiceanu. Model-Checking the Linux Virtual File System. N. D. Jones, M. Müller-Olm (eds.). VMCAI 2009. LNCS, vol. 5403, pp. 74–88. Springer, Heidelberg (2009).
[18]. S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, A. Rybalchenko. HSF(C): A Software Verifier Based on Horn Clauses. C. Flanagan, B. Koenig (eds.). TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012).
[19]. A. Gupta, C. Popeea, A. Rybalchenko. Threader: A Constraint-Based Verifier for Multi-threaded Programs. G. Gopalakrishnan, S. Qadeer (eds.). CAV 2011. LNCS, vol. 6806, pp. 412–417. Springer, Heidelberg (2011).
[20]. T. A. Henzinger, R. Jhala, R. Majumdar, K. L. McMillan. Abstractions from Proofs. Proc. POPL, pp. 232–244. ACM (2004).
[21]. T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Lazy Abstraction. Proc. POPL, pp. 58–70. ACM (2002).
[22]. A. Khoroshilov, V. Mutilin, E. Novikov, P. Shved, A. Strakh. Towards an Open Framework for C Verification Tools Benchmarking // E. Clarke, I. Virbitskaite, A. Voronkov (eds.). PSI 2011. LNCS, vol. 7162, pp. 179–192. Springer, Heidelberg (2012).
[23]. A. Khoroshilov, V. Mutilin, A. Petrenko, V. Zakharov. Establishing Linux Driver Verification Process // A. Pnueli, I. Virbitskaite, A. Voronkov (eds.). PSI 2009. LNCS, vol. 5947, pp. 165–176. Springer, Heidelberg (2010) .
[24]. J. T. Mühlberg, G. Lüttgen. Blasting Linux Code // L. Brim, B. R. Haverkort, M. Leucker, J. van de Pol (eds.). FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 211–226. Springer, Heidelberg (2007).

[25]. W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, F. Piessens. Sound Formal Verification of Linux's USB BP Keyboard Driver. A. E. Goodloe, S. Person (eds.). NFM 2012. LNCS, vol. 7226, pp. 210–215. Springer, Heidelberg (2012).
[26]. A. Podelski, A. Rybalchenko. Transition Predicate Abstraction and Fair Termination. Proc. POPL, pp. 132–144. ACM (2005).
[27]. H. Post, C. Sinz, W. Küchlin. Towards Automatic Software Model Checking of Thousands of Linux Modules — A Case Study with Avinux. Softw. Test., Verif. Reliab. 19(2), 155–172 (2009).
[28]. P. Shved, M. Mandrykin, V. Mutilin. Predicate Analysis with BLAST 2.7 // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg (2012).
[29]. P.E.Shved, V.S.Mutilin, M.U.Mandrykin. Opyt razvitiya instrumenta staticheskoj verifikatsii BLAST [An Expirience in Static Verification Tool BLAST Improving]. Programming and Computer Software, No. 3, 2012, pp. 24-35 (in Russian).
[30]. V.S.Mutilin, M.U.Mandrykin, E.M.Novikov, A.V.Khoroshilov, P.E.Shved. Ispol'zovanie drajverov ustrojstv operatsionnoj sistemy Linux dlya sravneniya instrumentov staticheskoj verifikatsii [Usage of Linux Drivers for Comparison of Static Verification Tools]. Programming and Computer Software, No. 5, 2012, pp. 54-71(in Russian).
[31]. V.S.Mutilin, E.M.Novikov, A.V.Strakh et al. Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 20, pp. 163-187 (in Russian).
[32]. C. Sinz, F. Merz, S. Falke. LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 542–544. Springer, Heidelberg (2012).
[33]. A. von Rhein, S. Apel, F. Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code // Proc. Java Pathfinder Workshop (2011).
[34]. T. Witkowski, N. Blanc, D. Kröning, G. Weissenbacher. Model Checking Concurrent Linux Device Drivers // Proc. ASE, ACM (2007), pp. 501–504.

Тестирование драйверов файловых систем в ОС Linux[1]

А. В. Цыварев, В. А. Мартиросян
tsyvarev@ispras.ru, vmartirosyan@gmail.com

Аннотация. В статье исследуется проблема тестирования драйверов файловых систем ОС Linux. По результатам рассмотрения существующих систем тестирования, применимых к драйверам файловых систем, формулируются требования к системе тестирования, способной вывести решение этой задачи на качественно новый уровень. В первую очередь, для этого необходимо помимо проверки стандартной функциональности файловых систем, обеспечить тестирование поведения драйверов в редко встречающихся ситуациях, таких как сбои в нижележащем устройстве хранения, условия нехватки памяти, а также выявлять ошибки, связанные с утечкой ресурсов.

Ключевые слова. Linux; файловые системы; драйвер; модуль ядра ОС Linux; тестирование

1. Введение

Операционные системы, основанные на ядре Linux, широко используются в мире. Они обеспечивают работу более 90% мощнейших суперкомпьютеров из международного рейтинга TOP500 и являются лидером рынка операционных систем для мобильных устройств в составе ОС Android. Как и во многих других операционных системах, понятие файловой системы в Linux является одним из ключевых.

Linux поддерживает множество файловых систем, включая не только разработанные специально для него (ext2, ext3, ext4) и характерные для других Unix-подобных ОС (Xfs, Btrfs, Jfs), но и, например, характерные для ОС Microsoft Windows (FAT32, NTFS). Каждая из этих файловых систем имеет свои достоинства, делающие ее предпочтительной в определенных областях применения.

В Linux ответственность за работу файловых систем и операций с ними разделена между подсистемой виртуальной файловой системы (Virtual File System, VFS) ядра ОС, которое транслирует запросы пользователя в

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 07.519.11.4024)

последовательность операций над некоторым универсальным низкоуровневым представлением файловой системы, и драйверами файловых систем, которые реализуют эти операции в соответствии с собственной архитектурой и форматом хранения данных на диске. Если код ядра, отвечающий за работу файловых систем, достаточно отлажен и надежен, то ошибки в драйверах файловых систем все еще распространены. Об этом можно судить, в том числе, по журналу изменений [1] в ядре Linux, в котором достаточно часто встречаются записи об исправлениях ошибок в этих драйверах.

Наиболее распространенным способом проверки корректности функционирования драйверов является тестирование.

Здесь следует учесть, что в ОС Linux драйверы устройств, в том числе и драйверы файловых систем, реализованы в виде модулей ядра. Код этих модулей, как и данные, находится в одном адресном пространстве с ядром, и выполняется в привилегированном режиме. В тоже время, код ядра устроен таким образом, чтобы корректно работать при вызове из пространства пользователя через механизм так называемых системных вызовов — специальных функций, при вызове которых меняется режим выполнения кода с обычного (характерного для пространства пользователя) в привилегированный (характерный для пространства ядра).

По этой причине, тестирование драйвера обычно устроено не как непосредственный вызов функций драйвера в предварительно созданном окружении, ибо корректно создать это окружение «вручную» в пространстве ядра очень сложно. Обычный же тест работает в пространстве пользователя и использует системные вызовы, которые в свою очередь вызывают те или иные функции драйвера файловой системы. При этом корректное окружение при вызове функций драйвера создается самим ядром. Более того, сами системные вызовы чаще всего вызываются из теста не напрямую, а через более высокоуровневый интерфейс, например, через функции библиотеки libc, которые представляют собой удобные обертки для системных вызовов.

Системные вызовы представляют собой обобщенный интерфейс к функциональности операционной системы. Поэтому, чтобы системный вызов в итоге развернулся бы в вызов функции нужного драйвера, ему нужно передать определенные параметры. Для драйвера файловой системы такими параметрами может быть путь к файлу, принадлежащему файловой системе, обслуживаемой этим драйвером, или дескриптор этого файла. На рис.1 показана стандартная схема тестирования драйверов файловых систем.

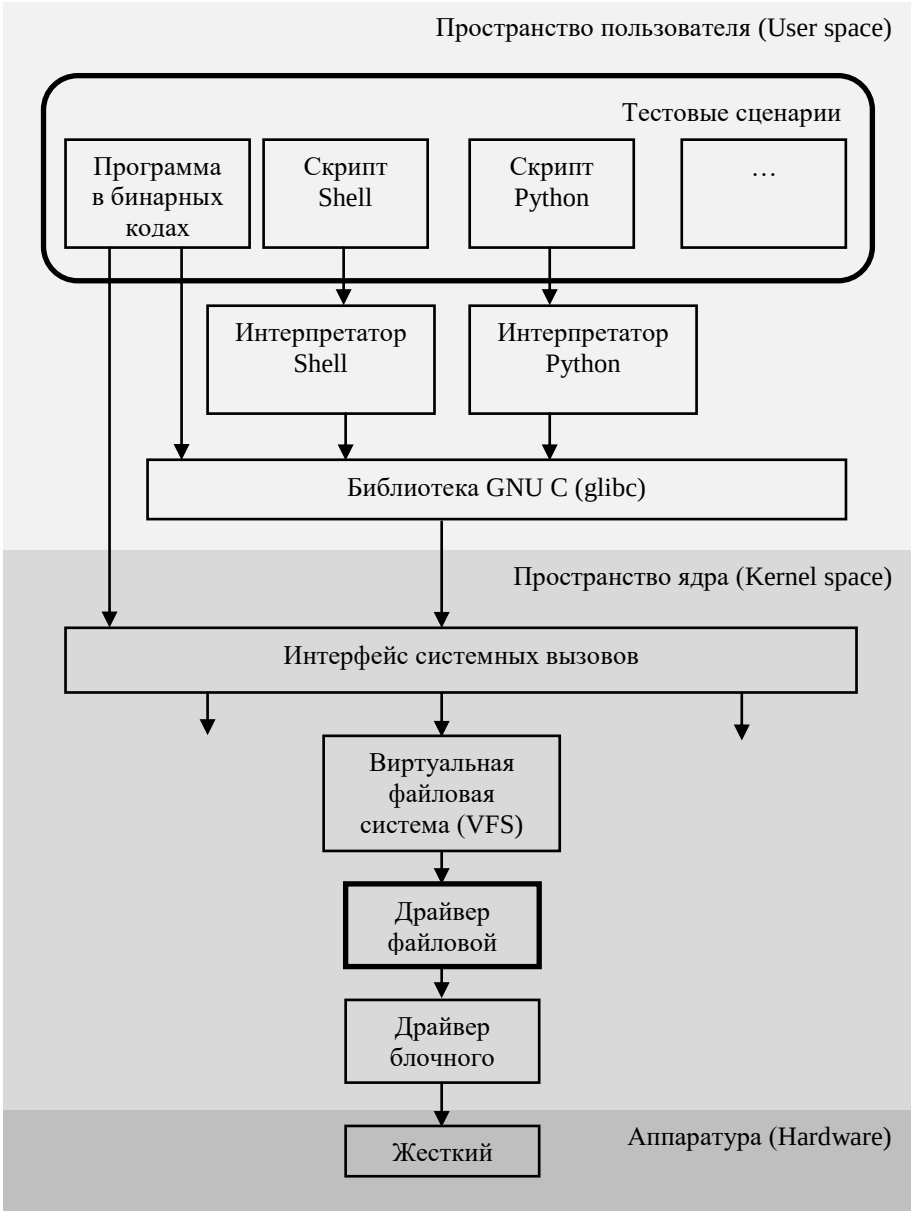


Рис. 1. Схема тестирования драйверов файловых систем.

Хотя за счет перебора параметров системных вызовов можно проверить большую часть функциональности драйвера, простого использования

системных вызовов с различными параметрами недостаточно для обеспечения качественного тестирования драйвера файловой системы. Во втором разделе статьи мы рассмотрим другие особенности устройства драйверов файловых систем ОС Linux. В третьем разделе мы покажем, какое влияние эти особенности оказывают на подходы к тестированию драйверов файловых систем. Далее будут рассмотрены существующие системы тестирования, используемые для поверки корректности функционирования драйверов, и сформулированы требования к системе тестирования, способной вывести решение этой задачи на качественно новый уровень.

2. Особенности устройства драйверов файловых систем

Рассмотрим особенности драйверов файловых систем, которые должны учитываться при их тестировании.

2.1. Реентерабельная программа с множеством входов

Как и многие другие типы драйверов, драйвер файловой системы представляет собой программу с множеством входов. Эти входы соответствуют функциям обратного вызова (callbacks), зарегистрированных определенным образом для обработки операций над файлами, директориями и другими объектами файловой системы (точнее, объектами ее низкоуровнего представления).

Отличительной особенностью драйверов файловых систем является очень большое количество таких функций: один драйвер может реализовывать до нескольких десятков функций, которые вызываются из ядра в и являются частью выполнения определенных системных вызовов. Более того, при выполнении некоторых системных вызовов вызывается не единственная функция драйвера, а целое множество, причем в определенной последовательности. Например, для системного вызова open() сначала ищется файл, соответствующий этому пути, если не найден — то файл создается, если найден — то для него создается и инициализируется дескриптор, и все эти операции выполняются за счет вызова функций драйвера.

Кроме того, драйверные функции обратного вызова могут вызываться из разных процессов, которые в свою очередь могут выполняться параллельно на различных ядрах процессора. Хотя подсистема виртуальной файловой системы VFS реализует определенную часть синхронизации между вызовами функций драйвера, основная нагрузка по обеспечению синхронизации ложится на драйвер файловой системы и занимает значительную часть его кода. Как показал анализ исправлений ошибок в стабильных ветках ядра за последний год [2] проблемы синхронизации в драйверах составляют самую большую долю ошибок (около 20%).

2.2. Работа в редких ситуациях

Основным сценарием использования драйвера является его работа при «нормальных» условиях. При таких условиях драйвер в состоянии выполнять свое основное предназначение; для драйверов файловых систем это может быть: создание файлов и директорий, чтение из файла и запись в него, получение различного рода информации о файловой системе и т. д. Помимо того, что это основной сценарий использования драйвера, это и наиболее частый сценарий.

Вместе с тем, иногда встречаются ситуации, когда выполнение драйвером основной задачи невозможно. Формально можно считать, что эти ситуации встречаются в ходе выполнения нетипичных или «ненормальных» сценариев использования драйвера. Такие сценарии встречаются относительно редко, поэтому и такие ситуации мы будем называть «редкими».

Одним из примеров таких редких ситуаций является вызов функций драйвера с некорректными пользовательскими данными. Хотя такой способ использования часто является следствием некорректной пользовательской программы, поведение драйвера в таком случае строго специфицировано, и драйвер обязан следовать этой спецификации.

Другой вариант редких ситуаций — нештатные условия, в которых выполняются функции драйвера. Под нештатными условиями чаще всего понимается состояние ОС, в котором часть ее возможностей не может использоваться (faulty environment). Для драйвера файловой системы нештатным состоянием ОС, препятствующим выполнению драйвером основных задач, может быть:

1. Нехватка свободной памяти в системе, или других ресурсов.
2. Невозможность писать/читать с устройства, на котором развернута файловая система (например, по причине сбойного жесткого диска)
3. Некорректное состояние файловой системы на устройстве, например, некорректная таблица файлов.

Требование корректной работы в нештатных ситуациях - это одно из принципиальных отличий компонентов ядра, к которым относятся и драйверы, от пользовательских приложений и библиотек.

Если пользовательская программа обнаруживает, что ее окружение не позволяет ей функционировать как положено, то обычная ее реакция — немедленное завершение работы. В некоторых случаях, возможны дополнительные шаги перед завершением работы: например, когда программа работает с важными данными, и их потеря или повреждение очень нежелательны, она может попытаться сохранить эти данные, или привести в согласованное состояние.

Ситуация меняется, когда речь идет о ядре и его модулях. Эти компоненты операционной системы обязаны корректно работать в любых ситуациях. Если

состояние системы не позволяет последней выполнить основную операцию, то функция должна корректно информировать об этом вызывающую сторону, при этом оставляя драйвер в согласованном и работоспособном состоянии. Для драйвера файловой системы реакцией на нештатные условия может быть:

1. Возврат файловой системы к состоянию, которое она имела до вызова функции драйвера, и возврат индикатора ошибки. Это самая «безобидная» реакция, и поэтому считается желательной реакцией драйвера на ситуацию, когда его функция не может выполнить свою основную задачу.
2. Возврат результата, наиболее согласующегося с состоянием файловой системы. Например, если какие-то данные дублируются, то в случае невозможности извлечь оригинал данных можно попробовать извлечь их дубликат и обработать его.
3. Перемонтирование файловой системы с опцией «только на чтение». Это обычная реакция драйвера в случаях, когда не удается записать данные, но вернуться в начальное состояние невозможно (например, удалось записать данные в файл, но не удалось отразить этот факт в журнале операций над файловой системой).

3. Особенности тестирования драйверов файловых систем

Принимая во внимание особенности устройства драйверов файловых систем, описанных в предыдущем разделе, рассмотрим, как это отразится на тестировании этих драйверов.

3.1. Тестирование параллельного выполнения кода драйвера

Так как код драйвера может выполняться параллельно, эту возможность надо проверять в тестах. Непосредственно проверить работу драйвера во всех ситуациях параллельного выполнения его кода практически невозможно из-за огромного количества таких состояний. Это обычная особенность тестирования параллельных программ. Одним из способов преодолеть это ограничение является анализ трассы выполнения кода драйвера на предмет так называемой конкуренции данных (data races).

3.2. Тестирование поведения драйверов файловых систем в редких ситуациях

Если редкая ситуация заключается в некорректных параметрах, переданных в функцию драйвера из пространства пользователя, то достаточно использовать соответствующий системный вызов, передав ему некорректный (с точки зрения обычной функциональности) параметр.

Если же редкая ситуация заключается в нештатном состоянии системы, то тестирование работоспособности драйвера в такой ситуации возможно несколькими способами. Наиболее прямолинейный подход для реализации тестирования этого требования — приведение системы к такому нештатному состоянию и последующий запуск тестовых сценариев.

Однако в случае нехватки свободной оперативной памяти такой подход не позволяет достичь качественного тестирования драйвера. Дело в том, что при системном вызове управление попадает в драйвер не сразу, а через промежуточные функции ядра. А эти функции, вообще говоря, также зависимы от наличия свободной памяти. И если в системе глобальная нехватка памяти, то очень вероятно, что управление даже не дойдет до тестируемого драйвера, а будет возвращена ошибка, например, о невозможности выделить память под дескриптор файла, что выполняется как раз промежуточной функцией ядра. В реальности, однако, возможна ситуация, когда нехватка памяти возникнет перед самым вызовом функции драйвера (если эту нехватку вызвал другой процесс, работающий параллельно).

Одним из способов протестировать поведение драйвера в ситуации нехватки памяти сохраняя при этом воспроизводимость сценариев тестирования, является эмуляция нехватки памяти в функциях, вызываемых непосредственно из драйвера. Так как все остальные функции ядра работают в «нормальных условиях», то управление из них всегда будет передаваться функциям драйвера (что важно, по предсказуемому сценарию). А сами функции драйвера уже будут «чувствовать» нехватку памяти, и должны показывать соответствующую реакцию. Для применения такого способа надо тем или иным способом работать в пространстве ядра (чтобы реализовать эмуляцию нехватки памяти).

Тестировать драйвер файловой системы в случае сбойного блочного устройства также лучше с использованием эмуляции.

3.3. Дополнительные проверки драйвера при тестировании

Не всегда по результату системного вызова можно определить, корректно ли себя ведет драйвер или нет. Примером некорректного поведения, ненаблюдаемого непосредственно из пространства пользователя, является утечка ресурсов ядра. Под утечкой ресурсов здесь подразумеваются запрошенные программой (пользовательским приложением, модулем ядра и пр.) ресурсы, которые с определенного момента не используются программой, но и не отдаются назад системе. Одним из часто используемых драйвером ресурсов является память, поэтому далее будем рассматривать именно ее, хотя большинство рассуждений применимы и к другим ресурсам.

Современные компьютеры обычно обладают большим объемом оперативной памяти, а поэтому ее небольшие потери чаще всего никак не влияют на работоспособность системы. Тем не менее, ситуация с утечками памяти не может считаться корректной для драйвера. Во-первых, даже если в результате

одной операции утечки памяти незначительны, то будучи выполненной много раз, операция может привести к значительной потери памяти, что может замедлить работу операционной системы, а то и вовсе привести к ее неработоспособности вследствие нехватки памяти, приводящей к перезагрузке. Во-вторых, сам факт «нецелевого» использования ресурсов не может считаться хорошим качеством драйвера.

Следовательно, драйвер не должен допускать утечек памяти, и это свойство стоит проверять при тестировании.

4. Обзор существующих систем тестирования, применимых к драйверам файловых систем

Задача тестирования Linux не нова, и существует немало систем, предназначенных для верификации тех или иных частей этой операционной системы. Рассмотрим некоторые наиболее популярные системы тестирования для ОС Linux, которые можно применить к файловым системам.

4.1. Системы тестирования общего назначения

Autotest [3] – система для автоматизированного тестирования. Эта система была разработана в первую очередь для тестирования ядра Linux, и включает тесты для этой цели.

Linux Test Project (далее LTP) [4] – проект, также направленный на тестирование Linux и его ядра.

Phoronix Test Suite [5] – платформа для оценки производительности различных компонентов Linux.

OLVER[6] - система для глубокого тестирования дистрибутивов на основе Linux на соответствие стандарту LSB Core.

Каждая из этих систем тестирования состоит из подсистемы управления и запуска тестов, и некоторого множества тестов и их наборов. Часть из этих тестов/наборов разработаны специально под соответствующую систему тестирования. Другие являются сторонними тестами/наборами, приспособленными к запуску под данной системой.

Исследуем тестовые сценарии из этих систем, которые применимы к файловым системам и их драйверам. В большинстве таких сценариев обычно выполняется некоторая характерная операция с файлами и директориями. Например, копирование файла, запись больших файлов и т. д. Другие сценарии заключаются в проверке работы того или иного системного вызова, который связан с файлами/директориями. Некоторые сценарии (например, compilebench в Autotest) проверяют работоспособность системы в случае параллельной работы с файлами. Но делают это на базовом уровне, просто запуская несколько потоков, выполняющих сходные действия.

В системе тестирования LTP можно включить режим тестирования, при котором симулируется общесистемная нехватка памяти и сбойное блочное устройство. Сценарий такой эмуляции чисто вероятностный — любой запрос памяти в ядре можно вернуть ошибку с константной вероятностью, тоже самое к запросам к блочному устройству. При этом сами тесты ничего не знают про такую эмуляцию, поэтому их вердикт «ошибка»(fail) теряет достоверность, а значит в таком режиме тестирования можно проверить только отказоустойчивость драйвера: упал/ не упал.

Минус вероятностного сценария симуляции ошибок заключается в отказе от точной воспроизводимости тестов (это отмечается, например, в статье [7]). Кроме того, в текущей реализации эмуляции системных ошибок в LTP эмуляция ошибок производится не только для тестируемого модуля, но и для всех других модулей и самого ядра. Это ведет к увеличению числа прогонов теста, необходимых для проверки обработки ошибок в тестируемом модуле: меньше вероятность, что ошибка будет симулирована именно в тестируемом модуле, и возможность ситуации, когда из-за симуляции ошибки в другом модуле, некоторый код из тестируемого модуля вообще не будет выполнен.

4.2. Сертификационные системы

Помимо систем тестирования, применимых к Linux в целом, также существуют системы тестирования предназначенные для запуска на определенных ОС, основанных на Linux. Такие системы в основном разрабатываются для сертификации программного обеспечения и физического оборудования на соответствие определенному дистрибутиву Linux. Рассмотрим наиболее известные сертификационные системы:

SUSE YES Certified Program [8] (в прошлом — Novel Yes Tools) – программа сертификации физического оборудования на совместимость с линейкой продуктов SUSE, в том числе SUSE Linux Enterprise, SUSE Open Enterprise Server.

Red Hat Hardware Program [9] – программа сертификации физического оборудования на совместимость с Red Hat Enterprise Linux.

Oracle Hardware Certification Program [10] – программа сертификации физического оборудования и систем на совместимость с ОС Oracle Linux.

Certification Program в Canonical [11]– программа сертификации физического оборудования на совместимость с ОС Ubuntu и поддержки качества.

В этих сертификационных программах используются как внешние тестовые системы и наборы (в том числе и упомянутые выше), так и специально разработанные для них.

Сертификационные системы для Red Hat и Canonical тестируют поведение оборудования, а следовательно и драйвера для этого оборудования, только в нормальных условиях. Две другие сертификационные системы проверяют

поведение драйвера также в случае нехватки памяти. Причем ситуация нехватки памяти создается в них разными способами.

В Oracle Hardware Certification Program ситуация нехватки памяти создается предварительным «отъемом» этой памяти у системы. Как уже писалось при рассмотрении тестирования при нехватке памяти, при таком подходе, теряется нацеленность тестирования на конкретный драйвер.

В SUSE YES Certified Program ситуация нехватки памяти симулируется только для тестируемого драйвера. Это достигается путем подмены вызовов функций, выделяющих память, из кода драйвера. Эти вызовы подменяются на вызовы специальных функций-оберток, которые с определенной вероятностью возвращают ошибку даже не пытаясь выделить эту память. Как и в случае с LTP, вероятностный сценарий симуляции ошибок ведет к меньшей воспроизводимости тестирования и к большему количеству прогонов теста.

Помимо симуляции нехватки памяти, в SUSE YES Certified Program отслеживаются операции запросов памяти и ее освобождения тестируемым драйвером. После завершения тестов и выгрузки драйвера из ядра проверяется, что все запрошенные драйвером куски памяти были им освобождены. Если это не так, то все неосвобожденные участки интерпретируются как утечки памяти. Из минусов этой реализации проверки утечек памяти стоит отметить небольшое количество перехватываемых функций (из-за этого запрос памяти через некоторые механизмы не отслеживается), и неотчуждаемость реализации от самой системы сертификации.

4.3. Системы тестирования нацеленные на определенные файловые системы

Характерным примером системы, нацеленной на тестирование драйверов файловых систем определенного типа, является Xfstests [12]. Эта система тестирования разрабатывается в рамках проекта по реализации на Linux самой файловой системы Xfs. Помимо файловой системы Xfs, эта система тестирования может применяться и для некоторых других журналируемых ФС, например, Btrfs и Ext4. Тесты в этой системе используют специальные последовательности операций с файлами, которые заставляют драйвер выполнять код, специально предназначенный для такого использования. На целевых файловых системах такая система обеспечивает гораздо более качественное тестирование, чем в системах тестирования общего назначения. Например, на дистрибутиве Debian 6 (ядро 2.6.32) покрытие по строкам кода драйвера Xfs составляет примерно 77%, тогда как в системах тестирования общего назначения это покрытие составляет порядка 50%.

5. Выбор требований и путей реализации системы тестирования драйверов файловых систем

На основе материала, рассмотренного выше, можно сформулировать требования к системе тестирования, которая обеспечивала бы более тщательное тестирование драйверов файловых систем, и предложить некоторые способы реализации этих требований.

5.1. Тестирование нормальной функциональности драйвера

Безусловно, функциональность драйвера в обычных условиях должна быть проверена. Тестирование этой функциональности может быть реализовано с использованием системных вызовов, параметры для которых подбираются такими, чтобы вынудить драйвер выполнить тот или иной код. Системные вызовы могут вызываться косвенно — через функции библиотеки libc, через команды shell, и т. д.

5.2. Тестирование работы драйвера в параллельных процессах

В сценариях тестирования должна быть также заложена работа драйвера в параллельных процессах. В качестве дополнительной проверки могут быть использованы подходы на основе проверки трассы выполнения кода модуля на предмет конкуренции данных (data races). В ИСП РАН в процессе разработки находится несколько инструментов, которые могут быть применены для выполнения такого рода проверок.

5.3. Тестирование системных вызовов с некорректными параметрами

В результате таких системных вызовов будут вызываться функции драйвера и будет проверяться их устойчивость к некорректным данным. Здесь стоит отметить, что некоторые некорректные данные проверяются и откидываются не доходя до вызова функций драйвера.

5.4. Тестирование работы драйвера при нехватки памяти

Один из эффективных способов такого тестирования — симуляция нехватки памяти в функциях, вызываемых из тестируемого драйвера. Такая симуляция может быть реализована с использованием инструмента fault injection [13], включенного в ядро ОС Linux, или KEDR Fault Simulation, разработанного в рамках проекта KEDR [14] в ИСП РАН.

5.5. Тестирование работы драйвера в случае сбойного блочного устройства

Драйвер файловой системы должен быть устойчив к сбойному блочному устройству, на котором развернута файловая система. Один из эффективных

способов такого тестирования — симуляция такого устройства в функциях, вызываемых из тестируемого драйвера. Для этого может быть использован как инструмент fault injection, так и KEDR Fault Simulation. В последнем случае такого рода симуляция должна быть реализована дополнительно, в стандартную поставку KEDR она не входит.

5.6. Тестирование работы драйвера в случае некорректного состояния файловой системы

Один из эффективных способов такого тестирования — иметь набор файлов с образами некорректных файловых систем, или иметь способ генерации такого набора образов. В качестве генератора образов некорректных файловых систем может быть связка утилиты dd, позволяющей перезаписывать любые участки блочного устройства, с утилитой, позволяющей осуществлять навигацию по элементам файловой системы. Утилиты для навигации есть для большинства широко используемых файловых систем: debugfs для Ext2/Ext3/Ext4, xfs_db для Xfs, jfs_debugfs для Jfs и т. д. Для некоторых файловых систем существуют готовые утилиты для направленной «порчи» файловых систем: fswreck [15] для OCFS, xfs_db[16] (команда blocktrash) для Xfs. Эти утилиты работают по сходным принципам — меняют некоторые байты/биты в заданном элементе файловой системы. Отличие состоит в том, fswreck модифицирует только элементы с заданными номерами, а xfs_db позволяет модифицировать сразу несколько элементов, выбрав их номера по псевдослучайному алгоритму с заданными параметрами.

5.7. Проверка утечек памяти в драйвере

Во время тестирования драйвера должна выполнятся проверка на утечки памяти. Для такой проверки подходит инструмент kmemleak(включен в ядро) [17]. Также можно воспользоваться инструментом KEDR Leak Check.

6. Заключение

Существующие тестовые наборы обеспечивают неплохой уровень тестирования, по крайней мере, наиболее распространенных драйверов файловых систем. Уровень покрытия в 77% строк кода драйвера XFS заметно выше среднего уровня покрытия по коду как в индустрии, так и среди компонентов ядра ОС Linux. Тем не менее, остаются сложности с обеспечением покрытия значительной части кода драйверов файловых систем ОС Linux, которые ввиду их особенностей требуют реализации целого комплекса специальных методов и подходов. В настоящей статье были рассмотрены эти особенности, сформулированы требования к формированию такого комплекса и предложены возможные направления решения поставленных задач.

Список литературы

[1]. Linux kernel bugzilla, https://bugzilla.kernel.org, 01.10.2012(дата обращения)
[2]. В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов. Анализ типовых ошибок в драйверах операционной системы Linux. Труды Института системного программирования РАН, 2012 г., том 22, стр. 349-374.
[3]. Autotest Framework, http://autotest.kernel.org.
[4]. Linux Test Project, http://ltp.sourceforge.net.
[5]. Phoronix Test Suite, http://www.phoronix-test-suite.com.
[6]. Open Linux VERification Project, http://linuxtesting.org/olver.
[7]. Subrata.M, Balbir S., Masatake Y., Putting LTP to test – Validating both the Linux kernel and Test-cases. http://ltp.sourceforge.net/documentation/technical_papers/Putting_LTP_to_Test.pdf, 2009.
[8]. SUSE YES Certified Program, http://www.novell.com/developer/yes/, 2012.
[9]. Red Hat Hardware Program, http://www.redhat.com/rhel/compatibility/hardware, 2012.
[10]. Oracle Hardware Certification Program, http://www.oracle.com/webfolder/technetwork/hcl/hcts/index.html, 2012.
[11]. Canonical's certification service, http://www.canonical.com/engineering-services/certification/hardware-certification, 2012.
[12]. Xfstests sources, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfstests.git.
[13]. Fault injection capabilities infrastructure, http://www.mjmwired.net/kernel/Documentation/fault-injection/
[14]. KEDR Project, http://linuxtesting.org/kedr.
[15]. OCFS2 tools sources, https://github.com/jjzhang/ocfs2-tools.
[16]. XFS user utilities sources, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfsprogs.git.
[17]. Kernel memory Leak Detector, http://www.mjmwired.net/kernel/Documentation/kmemleak.txt.

Testing of Linux File System Drivers

A.V. Tsyvarev, tsyvarev@ispras.ru, ISP RAS, Moscow, Russia
V.A. Martirosyan, vmartirosyan@gmail.com, RAU, Erevan, Armenia

Annotation. The paper investigates issues of Linux file system driver testing. Linux file system drivers are implemented as kernel modules, which works in the same address space as kernel core. For that reason, the driver should be very reliable. It should react adequately to incorrect file system images, to faults in operating system, etc. Also drivers should free all resources it requests as far as kernel have to work for long time without restart. Another important feature of file system drivers is that they are programs with multiple entry points, which may be executed simultaneously in respect with each other and with other kernel code.

Most of existing test systems verify only driver's behavior in normal situations by calling system calls, which are eventually dispatched by the kernel into the driver's functions. Some of them also check driver in concurrent scenarios but only at very basic level. Some test systems also verify driver's behavior under faulty environment, when request for memory allocation or disk read/write may fail. But fault scenarios used in those systems are probabilistic, which leads to problems with tests' reproducibility and requires to repeat tests many times to achieve better coverage. There is one test system, which checks for memory leaks in the driver under test.

The paper concludes by statement of requirements for more throughout file system driver testing. According to the requirements a test system have to cover the following aspects:

1. Normal scenarios on system calls level.
2. Parallel scenarios with additional checks for data races.
3. Fault scenarios with insufficient memory and faulty block devices using such techniques as fault injection.
4. Handling of incorrect file system images.
5. Driver testing on system calls with invalid arguments.
6. Check leaks of memory and other resources requested by driver under test.

Keywords. Linux; file systems; driver; Linux kernel module; testing

References

[1]. Linux kernel bugzilla, https://bugzilla.kernel.org, 01.10.2012(дата обращения)
[2]. V.S.Mutilin, E.M. Novikov, A.V. Khoroshilov. Analiz tipovykh oshibok v drajverakh operatsionnoj sistemy Linux [Analysis of typical faults in Linux operating system drivers]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 349-374 (in Russian).
[3]. Autotest Framework, http://autotest.kernel.org.
[4]. Linux Test Project, http://ltp.sourceforge.net.
[5]. Phoronix Test Suite, http://www.phoronix-test-suite.com.
[6]. Open Linux VERification Project, http://linuxtesting.org/olver.
[7]. Subrata.M, Balbir S., Masatake Y., Putting LTP to test – Validating both the Linux kernel and Test-cases. http://ltp.sourceforge.net/documentation/technical_papers/Putting_LTP_to_Test.pdf, 2009.

[8]. SUSE YES Certified Program, http://www.novell.com/developer/yes/, 2012.
[9]. Red Hat Hardware Program, http://www.redhat.com/rhel/compatibility/hardware, 2012.
[10]. Oracle Hardware Certification Program, http://www.oracle.com/webfolder/technetwork/hcl/hcts/index.html, 2012.
[11]. Canonical's certification service, http://www.canonical.com/engineering-services/certification/hardware-certification, 2012.
[12]. Xfstests sources, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfstests.git.
[13]. Fault injection capabilities infrastructure, http://www.mjmwired.net/kernel/Documentation/fault-injection/
[14]. KEDR Project, http://linuxtesting.org/kedr.
[15]. OCFS2 tools sources, https://github.com/jjzhang/ocfs2-tools.
[16]. XFS user utilities sources, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfsprogs.git.
[17]. Kernel memory Leak Detector, http://www.mjmwired.net/kernel/Documentation/kmemleak.txt.

Об одном методе построения схемы полного гомоморфного шифрования

А.В. Шокуров (shok@ispras.ru), К.В. Сергеев (kosts88@mail.ru)

Аннотация. Предложен вариант метода Джентри для построения полного гомоморфного шифрования.

Ключевые слова: шифрование, гомоморфное шифрование, открытый ключ, секретный ключ.

1. Введение.

Гомоморфное шифрование позволяет производить вычисления над секретными данными, заменяя их вычислениями над соответствующими данными в зашифрованном виде. Гомоморфность шифрования относительно операции умножения целых чисел по некоторому модулю достигается, например, в криптосистемах RSA, Эль-Гамаля, Гольдвассер-Микали (см., например, [5]) .

Полностью гомоморфные схемы шифрования, обладающие свойством гомоморфности относительно операций сложения и умножения, были предложены недавно. Первая из них была представлена Крейгом Джентри в [1,2,3]. Эта криптосистема использует идеальные решетки. Позже Крейгом Джентри и другими в [4] была предложена еще одна полностью гомоморфная схема шифрования, обладающая схожими свойствами, но проводящая операции над целыми числами. В данной работе предложена новая система перешифрования внешне похожая на предложенную в [4] схему, однако не требующая введения дополнительной информации о секретном ключе.

2. Основные обозначения и определения.

Определение 1. Схемой шифрования с открытым ключом называется тройка алгоритмов *E=(Gen, Decr, Encr)* такая, что

- *Gen* - полиномиальный вероятностный алгоритм, *Encr* - полиномиальный алгоритм (и, возможно, вероятностный, тогда схема шифрования называется вероятностной), - *Decr* - полиномиальный алгоритм;

- алгоритм генерации ключа *Gen*, получая на вход некоторый параметр λ, называемый параметром безопасности , создает пару ключей: секретный ключ $k_s \in \Sigma^\lambda$ и открытый ключ $k_p \in \Sigma^\lambda$, $\Sigma = \{0,1\}$ где .

- алгоритм шифрования Encr, получая на вход открытый k_p ключ и открытый текст m, выдает на выходе щифротекст c.

- алгоритм дешифрования, получая на вход секретный k_s ключ и шифротекст c, выдает открытый текст.

- для любых открытых текстов $m \in \Sigma^{f(\lambda)}$, где f полиномиальная функция, и любой пары ключей, (k_s, k_p) полученной с помощью алгоритма Gen, выполняется соотношение

$$Decr\left(k_s, Encr\left(k_p, m\right)\right) = m$$

Введем следующие обозначения и определения.

Определение 2. Пусть дана схема шифрования с открытым ключом *(Gen, Decr, Encr)* и пусть даны алгоритм *Eval* и множество функций *F* такие, что для $f \in F$ любой функции от t переменных и для любых $c_1, \ldots, c_t$ $c_i = Encr(k_p, m_i),\ i = 1, \ldots, t$ шифротекстов алгоритм $Eval(k_p, f, c_1, \ldots, c_t)$ вычисляет шифротекст *c* такой, что $Decr(k_s, c) = f(m_1, \ldots, m_t)$. Тогда четверка алгоритмов *E=(Gen, Decr, Encr, Eval)* называется схемой шифрования, обладающей свойством гомоморфности на множестве фунций F.

Определение 3. Гомоморфной схемой шифрования на множестве функций F называется такая четверка алгоритмов *E=(Gen, Decr, Encr, Eval)*, для которой шифротексты обладают свойством компактности и алгоритм Eval является эффективным. Если множество F совпадает с множеством всех функций, то такая схема называется полностью гомоморфной схемой шифрования.

3. Частично гомоморфная схема шифрования.

Пусть заданы некоторые параметры λ, N, P, Q и τ, $N \ll P$ $Q = f(P, \log P)$ причем , а , где функция f полином. Для любых целых *x* и *y* $x \bmod y$ через будем понимать такое число z, что и *x* - *y* $-y/2 < z \le y/2$ делится на *y*. Считаем, что битовая строка секретного ключа имеет старший k_s $2^P \le k_s < 2^{P+1}$ разряд 1, и .Рассмотрим следующую открытую схему шифрования *E**.

Алгоритм 1. $Gen_{E^*}(\lambda)$ выдает на выход случайную *P*-битовую нечетную строку k_s – секретный ключ и $Q \cdot \tau$ -битовую строку k_p - открытый ключ.

Алгоритм 2. $Encr_{E^*}(k_p, m)$ - выбирает для бита *m* случайную строку *c* вида

$$c = m' + k_s \cdot q,$$

где число m' имеет ту же четность, что и бит *m*, а число бит слова m' не превосходит *N*, а число битов слова *c* не превосходит *Q* .

Алгоритм 3. $Decr_{E^*}(k_s, c)$ выдает на выход $(c \bmod k_s) \bmod 2$.

Алгоритм 4. $Eval_{E^*}(f, c_1, \dots, c_t)$ переходит от представления функции $f(x_1, \dots, x_t)$ в виде схемы к представлению в виде полинома $F(m_1, \dots, m_t)$ в кольце многочленов над $\mathbb{Z}_2$. Заменим теперь все операции над битами в этом полиноме, соответствующими им целочисленного сложения и умножения над строками. Мы получим новый полином

$$F_Q(c_1, \dots, c_t)$$

от *t* переменных строк длины *Q*. На выход алгоритма выдается

$$c = F_Q(c_1, \dots, c_t).$$

Утверждение 1. (Корректность шифрования) Для любого шифротекста $c = Encr_{E^*}(k_p, m)$ выполнено $Decr_{E^*}(k_s, c) = m$.

Утверждение 2. (Свойство гомоморфности) Схема шифрования E^* обладает свойством гомоморфности на множестве фунций *F*, которые могут быть представлены многочленами степени не выше k и содержащими не более *l* слагаемых, для которых выполнено соотношение $Nk + \log l < P - 4$.

4. Алгоритм перешифрования.

Определение 4. Если схема шифрования *E* гомоморфна относительно собственной

функции расшифрования $Decr_E(k_s, c)$, а также функций

$$Decr_E(k_s, c_1) + Decr_E(k_s, c_2) \bmod 2 \text{ и}$$
$$Decr_E(k_s, c_1) \cdot Decr_E(k_s, c_2) \bmod 2,$$

то она называется расширяемой.

Утверждение 3. Для того, чтобы схема шифрования *E=(Gen, Decr,Encr,Eval)* являлась полностью гомоморфной схемой шифрования достаточно, чтобы она была гомоморфна относительно операций сложения и умножения по модулю 2 и расширяема.

5. Заключение. Построение полностью гомоморфной схемы шифрования.

Обозначим через *LSB(c)* функцию четности числа $c \in \mathbb{Z}$, т.е. такую, что *LSB(c)*=0, если *c* четно, и *LSB*(c)=1, если *c* нечетно. Обозначим через $[\![x]\!]$ ближайшее целое к числу $x \in \mathbb{R}$, если его дробная часть не равна $1/2$.

Утверждение 4. Функцию расшифрования в криптосистеме *E* можно представить в виде

$$Decr_E(k_s, c) = LSB(c)\ XOR\ LSB\left(\left[\!\left[{}^{c}/_{k_s} \right]\!\right]\right). \qquad (1)$$

Утверждение 5. Пусть заданы $m_i \in \{0,1\}$, где $i = 1, \dots, n$ и $h: \mathbb{Z} \longrightarrow \mathbb{Z}_2$ гомоморфизм колец, преобразующий 1 в 1. Положим $t_i = h(m_i)$ при $i = 1, \dots, n$. Тогда существуют симметрические функции $f_k(x_1, \dots, x_n) \in \mathbb{Z}_2[x_1, \dots, x_n]$ степени не выше 2^k *k*, для которых выполняется равенство

$$\sum_{i=1}^{n} m_i = \sum_{j=1}^{j<\log n} f_j\left(t_{1,\dots,}t_n\right) \cdot 2^{j-1} \qquad (2)$$

Применим формулу (2) для суммирования *Q K*-разрядных чисел в двоичном представлении

$$\sum_{i=1}^{Q}\sum_{j=1}^{K} m_{i,j} \cdot 2^{j-1} = \sum_{j=1}^{K}\left(\sum_{i=1}^{Q} m_{i,j}\right) = \sum_{j=1}^{K}\left(\sum_{i=1}^{j<\log Q} f_i(t_{1,k}, \dots, t_{Q,k}) \cdot 2^{i-1}\right) \cdot 2^j. \qquad (3)$$

Для вычислений с помощью формулы (1), достаточно использовать приближенное значение величины $r \approx {}^{1}/_{k_s}$ с *2Q* двоичными битами. Для вычисления значения функции $\left[\!\left[{}^{c}/_{k_s} \right]\!\right]$ требуется знание младшего целого разряда и первого двоичного дробного разряда произведения *rc*. В этом случае сумма по модулю 2 этих битов дает значение функции $\left[\!\left[{}^{c}/_{k_s} \right]\!\right]$. Для вычисления этих двух битов достаточно использовать формулу (3) для значений $K = O(\log Q)$.

Следствие. Предложенная в разделе 3 частично гомоморфная схема расширяема при значениях
$N = \omega(\log\lambda), P \geq N \cdot \Theta(\lambda\ log^2\ \lambda),\ Q = \omega(P^2 \log\lambda), \tau = Q + \omega(\log\lambda)$
и поэтому является полностью гомоморфной.

Список литературы

[1] Craig Gentry, Computing arbitrary functions of encrypted data. ACM, 2010.
[2] Craig Gentry, Fully homomorphic encryption using ideal lattices. 41st ACM STOC, 2009.
[3] Craig Gentry, A fully homomorphic encryption scheme. Stanford University, Ph.D. thesis. 2009.
[4] M. van Dijk, C. Gentry, S. Halevi, V. Vaikuntanathan, Fully homomorphic encryption over the integers. International Association for Cryptographic Research, 2010.
[5] Н. П. Варновский, А. В. Шокуров, Гомоморфное шифрование. Труды Института Системного Программирования. Том 12. М: ИСП РАН, 2007, с. 27-36.

On Constructing a Fully Homomorphic Encryption

A.V. Shokurov
(shok@ispras.ru),
K.V. Sergeev
(kosts88@mail.ru)

Abstract. Suggested a new construction for full homomorphic encryption which doesn't use an extra information on secret key.

Key words. Encryption scheme, homomorphic encryption scheme, public key, secret key.

Энергоэффективные вычисления для группы кластеров[1]

Д.А. Грушин, Н.Н. Кузюрин
grushin@ispras.ru, nnkuz@ispras.ru

Аннотация. Рассмотрена проблема балансировки нагрузки для множества параллельных задач на группе географически распределенных кластеров уменьшающая количество энергии при вычислениях. Предложены несколько алгоритмов распределения задач и проведена экспериментальная проверка их эффективности.

Ключевые слова: энергоэффективность; балансировка нагрузки; распределенные вычисления

1. Введение

В последние годы во всем мире происходит значительный рост потребности в вычислительных ресурсах. Если раньше суперкомпьютеры были крайне дороги и доступны единицам, то с появлением вычислительных кластеров, собранных из общедоступных компонентов, наука и промышленность получили в своё распоряжение простой и недорогой способ использования высокопроизводительных вычислений.

Типичный вычислительный кластер (Beowulf кластер[2]) состоит из широко распространённого аппаратного обеспечения и работает под управлением операционной системы GNU/Linux или FreeBSD. Если кластер предназначен для использования многими пользователями, то управление кластером осуществляет менеджер ресурсов. Пользователи отправляют свои задания менеджеру, который ставит их в очередь и, по мере высвобождения вычислительных узлов, осуществляет запуск заданий.

[1] Выполнено при финансовой поддержке Минобрнауки РФ, контракт 07.514.11.4001

[2] Одна из типичных конфигураций -- набор компьютеров, собранных из общедоступных компонентов, с установленной на них операционной системой Linux, и связанных сетью Ethernet, Myrinet, InfiniBand или другими относительно недорогими сетями. Такую систему принято называть кластером Beowulf.

От количества поступающих задач зависит сколько узлов кластера будет занято выполнением задач, а сколько простаивать. Согласно статистике большинство кластеров испытывает периодическую нагрузку – когда интенсивность потока задач различается в несколько раз в разное время суток. Это означает, что даже при относительно плотной загрузке заданиями в среднем, существуют периоды, когда большая часть узлов кластера не выполняет заданий и простаивает.

В работе [1] мы показали, что временный перевод простаивающих узлов кластера в спящий режим приводит к существенной экономии электроэнергии. В данной работе мы хотим рассмотреть возможность снижения расхода электроэнергии для группы кластеров, находящихся под управлением одного менеджера ресурсов – брокера, который, получая поток заданий от пользователей, распределяет их между кластерами. В такой системе существует несколько возможностей для экономии электроэнергии (как в количественном смысле, так и в денежном -- снижая стоимость энергии):

- Различная энергоэффективность (отношение производительности к энергопотреблению) кластеров;
- Географическое положение кластеров. Стоимость энергии в разных регионах может существенно различаться. Отправляя задачи на кластер, с более низкой ценой на электроэнергию, можно уменьшить общую стоимость энергии. Стоимость энергии различается также в разное время суток, что даёт дополнительные возможности выбора если кластеры находятся в разных часовых поясах.

В данной статье мы оцениваем с помощью моделирования насколько возможно снизить общее количество энергии и её стоимость в подобной вычислительной системе. При этом задача снижения энергопотребления группы кластеров должна рассматриваться совместно с общей проблемой повышения эффективности использования вычислительных ресурсов. Мы считаем, что её исследование позволит обеспечить в будущем значительную экономию энергопотребления.

2. Оптимизация энергопотребления одного кластера

Следует отметить, что в литературе рассматриваются разные модели и постановки задач энергосберегающих вычислений [2]. Наибольшее количество работ посвящено задаче оптимизации энергопотребления вычислительной системы, состоящей из одного процессора [2,3,4,5,6,7]. При этом основным способом уменьшения затрачиваемой энергии является выключение простаивающих компонент вычислительной системы и их последующее включение по мере надобности. Дополнительными способами могут быть:

- Перераспределение вычислительных заданий по времени при условии наличия многотарифной схемы оплаты электроэнергии (например день-ночь). Тогда за счёт повышения загрузки системы ночью, днем вычислительная нагрузка будет снижена и простаивающие компоненты вычислительной системы отключены. Общее количество потреблённой электроэнергии не снижается, однако уменьшается её стоимость, что также рассматривается как повышение энергоэффективности;
- Программное управление производительностью компонент вычислительной системы. Современные процессоры и оперативная память имеют возможность динамически изменять свою частоту и рабочее напряжение. Такой механизм носит название DVS (dynamic voltage and frequency scaling). Основной принцип данного механизма заключается в том, что при понижении напряжения процессора время вычислений увеличивается, однако общее количество энергии потраченной на вычисления уменьшается.

Важный аспект динамического управления энергопотреблением состоит в том, что смена состояния системы (включение, изменение производительности и т.п.) имеет стоимость, выраженную в дополнительном количестве потреблённой энергии, задержки или потери производительности, что, вообще говоря, не гарантирует снижения энергозатрат при переводе системы в спящий режим в отсутствие работы и обратно по мере надобности. Для того, чтобы компенсировать данные потери энергии, система должна находиться в спящем состоянии не менее определённого промежутка времени. Такой промежуток времени называется ``минимальным временем сна'' (таблица 1).

Так, не более чем в 2 раза худший результат по сравнению с оптимальным по критерию минимизации затраченной энергии гарантируется, если отключать устройство через время $\tau = T_{be}$ [8]. И не более чем в $e/(e-1) = 1.58$ раза, если выключать устройство через время t с вероятностью $p_t = Ke^{t/E_{wu}}$, где $K = 1/(E_{wu}(e-1))$ [8], что является наилучшей возможной оценкой в классе онлайновых вероятностных алгоритмов [2].

Величина	Значение
E_{wu}	количество энергии, расходуемое при включении
E_{sd}	количество энергии, расходуемое при выключении
T_{be}	минимальное время сна, компенсирующее потери энергии от включения и выключения (break-even time)
T_{wu}	wakeup delay --- задержка при включении
τ	время простоя

Таблица 1. Основные обозначения

Рассматривая задачу отключения/включения узлов вычислительного кластера следует отметить, что задача является многокритериальной, где основным критерием является сэкономленная энергия, а вспомогательными время ожидания и число включений узлов. При этом важно учитывать следующие особенности:

- Каждая многопроцессорная задача требует одновременного включения нескольких узлов.
- Каждый многопроцессорный узел может выполнять одновременно несколько задач. Выключить узел можно только тогда, когда все его процессоры простаивают.

Из проведённых нами измерений [1] видно, что в состоянии простоя узел кластера потребляет в 12 раз больше энергии чем в выключенном состоянии. Моделирование показывает, что использование простого алгоритма, отключающего узлы существенно снижает расход энергии.

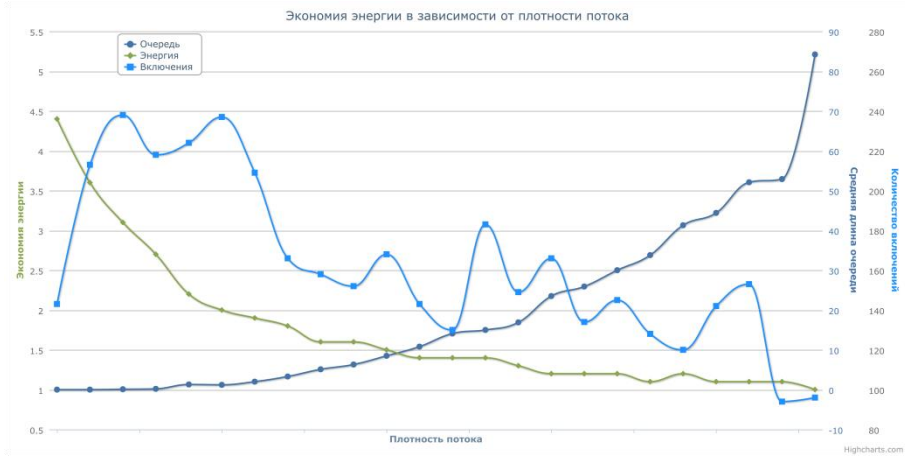


Рисунок 1. Результаты моделирования для одного кластера

На рисунке 1 показаны результаты моделирования для одного кластера. Мы сравнивали количество потраченной энергии, среднюю длину очереди, количество включений узлов для одного кластера в двух случаях: без отключения простаивающих узлов и с отключением. При этом проводилась серия испытаний, в каждом последующем плотность потока задач увеличивалась.

Эксперименты проводились для потоков с различным соотношением 1-процессорных и многопроцессорных задач, но существенного влияния на полученные результаты это не оказало, так как мы не использовали в модели узлы с несколькими процессорами.

Результаты показывают, что при очень слабых потоках (примерно 10 задач в день в нашем эксперименте) отключение узлов снижает потребление энергии в 4-5 раз. При меньшем количестве задач эта величина будет ещё больше. Средние значения находятся в середине графика. Потребление энергии в данном случае сокращается на 20-80% в зависимости от интенсивности потока задач.

3. Оптимизация энергопотребления группы кластеров

Возникающая задача управления ресурсами групп кластеров является многокритериальной и затрагивает много различных факторов.

Рассмотрим типичную двухуровневую схему где центральный планировщик (брокер ресурсов) решает на какой кластер направить поступающую к нему задачу.

В распределенной вычислительной системе кластеры располагаются в различных географических регионах. Стоимость энергии в разных регионах отличается, и в каждом отдельном регионе изменяется в течении суток. Следовательно, в каждый момент времени брокер имеет возможность отправить поступившую к нему задачу на кластер с минимальной стоимостью энергии. Мы провели серию экспериментов для того, чтобы оценить насколько возможно сократить стоимость затраченной на вычисления энергии в такой системе.

Для создания экспериментальной модели были выбраны несколько кластеров из списка TOP500 за 2011 год [9]. Для выбранных кластеров мы взяли из списка значения энергоэффективности и размера, минимальное значение каждого параметра было принято за единицу.

Для упрощения модели мы предположили, что все кластеры имеют одинаковую производительность вычислительных узлов. Базовую величину пикового энергопотребления одного узла кластера мы предположили равной 520 Вт, исходя из данных, полученных нами ранее для узла кластера установленного в ИСП РАН (HP ProLiant DL380 G3) [1]. Значения энергопотребления в состояниях простоя и сна составили 0,43 и 0,03 от пикового соответственно. Величина энергопотребления для каждого узла вычислялась как произведение базового энергопотребления и коэффициента энергоэффективности кластера.

Список кластеров представлен в таблице 2. Стоимость электроэнергии в представленных странах была взята из публично доступных данных [10].

Страна, часовой пояс (GMT)	**Год установки**	**Относи-тельный размер**	**Относительная энерго-эффективность (Mflop/ Watt)**	**Стоимость энергии (Цент США за 1kWh)**
Spain, +2	2011	1.0	1.9	19.69
United Kingdom, 0	2011	1.0	1.7	18.59
United States, -7	2011	1.8	1.4	11.2
Canada, -4	2011	2.1	1.5	6.18
Australia, +8	2011	2.4	1.2	28.88

United States, -5	2011	4.2	8.4	11.2
Russia, +3	2011	8.4	1.0	9.49
Saudi Arabia, +3	2009	16.7	1.6	13.1
Japan, +9	2010	18.7	3.6	12
China, +7	2010	30.8	2.1	16

Таблица 2. Набор кластеров для моделирования

В моделируемой системе задачи поступали в очередь брокера и затем распределялись по кластерам. Для потока задач, поступающих на единственный кластер, характерными особенностями являются переменная интенсивность поступления задач и периодичность. В среднем вероятность появления задачи в ночное время суток в два раза меньше, чем в дневное. Для брокера, распределяющего задачи между большим количеством кластеров, находящихся в разных часовых поясах, мы считаем, характерна равномерная интенсивность потока.

Для моделируемой системы мы выбрали поток задач, характеристики которого представленны в таблице 3.

Минимальная и максимальная ширина одной задачи	1 -- 10
Минимальная и максимальная длительность одной задачи	10 мин -- 2 часа
Доля однопроцессорных задач	0.8

Таблица 3. Характеристики потока задач

Приведем далее результаты моделирования для двух случаев. В первом случае брокер распределяет задачи между кластерами не учитывая стоимость энергии, используя алгоритм S. Во втором случае стоимость энергии учитывается -- алгоритм P. Оценка эффективности распределения проводилась по следующим критериям:

- Минимизация общей стоимости затраченной энергии;
- Минимизация среднего времени ожидания задачи в очереди.

Алгоритм S. Выбирается кластер с минимальным отношением общей площади задач в очереди к ширине кластера -- $h_j = (\sum_{k=1}^{N} S_k)/W_j$, где N -- число задач, стоящих в очереди, S_k -- площадь задачи, W_j -- число узлов кластера. Данную величину можно рассматривать как оценку времени пребывания задачи в очереди -- время, через которое задача, поступившая в очередь, будет запущена.

Алгоритм P. Входным параметром алгоритма является значение H_{diff}, которое определяет допустимую разницу между минимальным и максимальным временем ожидания в очередях кластеров при которой алгоритм будет экономить энергию -- алгоритм будет выбирать кластер, который выполнит задачу с минимальным расходом энергии.

Для каждого кластера C_j брокер определяет стоимость электроэнергии в данный момент времени и значение $h_j = (\sum_{k=1}^{N} S_k)/W_j$ -- отношение общей площади задач в очереди к ширине кластера (оценка времени пребывания задачи в очереди). Если $max(h_j) - min(h_j) > H_{diff}$, то задача отправляется на кластер с минимальным значением h_j. В противном случае задача отправляется на кластер с минимальной стоимостью энергии.

На рисунке 2 показана зависимость между плотностью потока, долей однопроцессорных задач в потоке и отношением значений реального времени ожидания к величине $h_j = (\sum_{k=1}^{N} S_k)/W_j$ (оценка времени пребывания задачи в очереди). Значения получены моделированием работы одного кластера, на котором применялся алгоритм распределения задач Backfill **Ошибка! Источник ссылки не найден.**. Плотность потока и доля однопроцессорных задач в потоке постепенно увеличивались. Результаты показали, что оценка h_j отличается от реального времени ожидания задачи в очереди не более чем на 30%. При этом большая часть значений h_j попадает в диапазон $[0.8,1]$. Эти значения соответствуют потокам со средней долей однопроцессорных задач -- $[0.3,0.6]$.

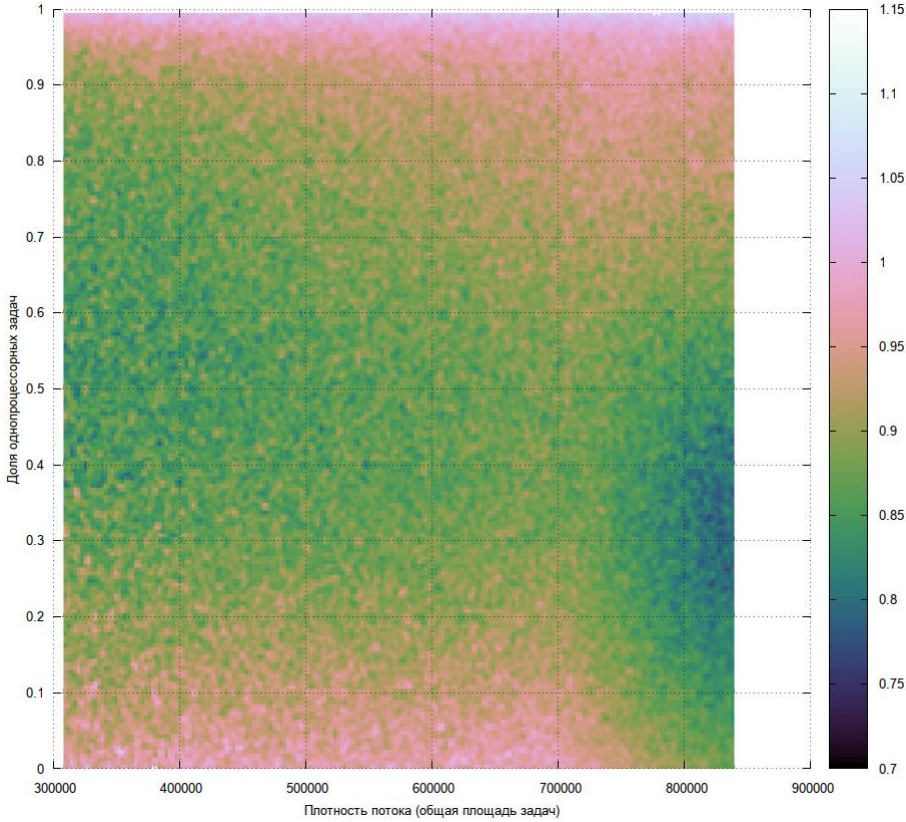


Рисунок 2. Оценка времени ожидания -- отношение значений реального времени ожидания к величине $h_j = (\sum_{k=1}^{N} S_k)/W_j$ *. Результаты показали, что оценка* h_j *отличается от реального времени ожидания задачи в очереди не более чем на 30%. При этом большая часть значений* h_j *попадает в диапазон* [0.8,1]*. Эти значения соответствуют потокам со средней долей однопроцессорных задач* -- [0.3,0.6].

4. Эксперименты

Для каждого из двух вариантов алгоритма распределения задач брокером: S и P мы провели серию экспериментов с описанной выше группой кластеров (таблица 2). В обеих сериях использовались одинаковые потоки задач. В

одной серии использовался поток с постоянными характеристиками (таблица 3), но различной интенсивности -- начиная с небольшой плотности задач (500 задач в сутки) и затем каждый последующий раз увеличивая плотность на равное количество задач. В каждой серии использовался также второй параметр -- H_{diff}, который увеличивался с шагом 5 минут. На каждом кластере простаивающие узлы отключались. В ходе экспериментов измерялось среднее время ожидания задач в очередях кластеров и общая стоимость затраченной энергии. Далее будем обозначать две серии S и P соответственно.

На рисунке 3 показана разница между средним временем ожидания в серии P и S и отношение стоимости затраченной энергии в серии S к P.

Эксперименты показали, что время ожидания в серии P всегда увеличивается. Обратим внимание на отмеченные на рисунке области: 1, 2, 3. Наибольшее увеличение времени ожидания наблюдается в области 1 при плотности потока от 2000 до 5000 задач в сутки и значениях H_{diff} 7000-9000 секунд. Максимальное увеличение времени ожидания составило 2659 секунд при значениях плотности и H_{diff} 3140 и 9000 соответственно.

Минимальное увеличение времени ожидания составляет не более 300 секунд и наблюдается в области 3 при наибольшей плотности потока и наименьших значениях H_{diff}.

Область средних значений отмечена номером 2. Максимальное увеличение времени ожидания в данном случае составляет около 800 секунд при значении H_{diff} 4000.

Изменение стоимости затраченной энергии показано на второй части рисунка 3. Наибольшая экономия достигается на потоках с самой маленькой плотностью в рамках данного эксперимента (500 задач в сутки) и значениях H_{diff} 1000 и выше. При этом, по мере увеличения значения H_{diff}, уже после 2000 стоимость затраченной энергии не изменяется. Максимальное значение составило 1.87 при значениях плотности и H_{diff} 620 и 8700 соответственно.

Таким образом, в данном эксперименте значения входного параметра H_{diff} в интервале от 2000 до 4000 дают наилучшее соотношение между стоимостью затраченной энергии и временем ожидания.

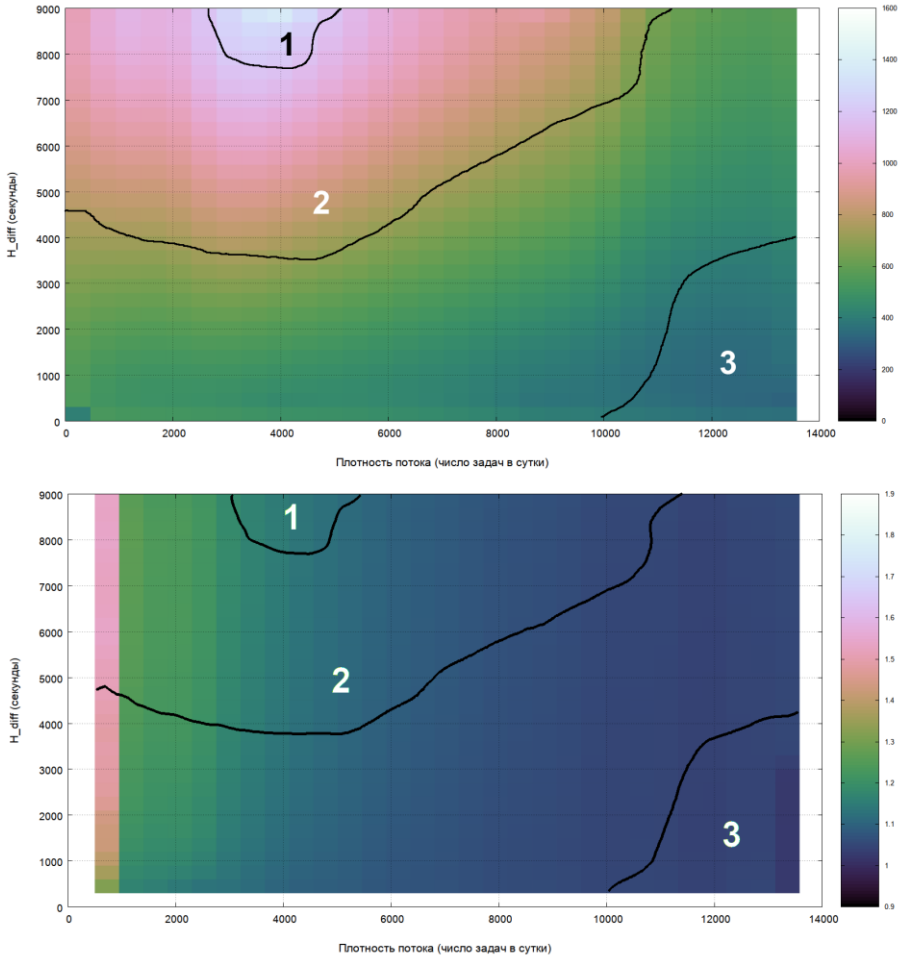


Рисунок 3. Увеличение среднего времени ожидания в серии P по отношению к серии S и отношение стоимости затраченной энергии в серии S к P

5. Заключение

В статье была рассмотрена задача снижения стоимости энергии вычислительной системы, состоящей из нескольких географически распределенных кластеров. В такой системе существует несколько возможностей для снижения стоимости энергии. Это различная

энергоэффективность кластеров и различная стоимость энергии в зависимости от географического положения кластера и времени суток.

Мы провели моделирование вычислительной системы, состоящей из 10 кластеров (данные энергоэффективности и размера были взяты из списка TOP500 за 2011 год), и сравнили результат работы двух алгоритмов --- S и P. Алгоритм S распределял задачи не учитывая расход энергии, а алгоритм P, в зависимости от значений входного параметра H_{diff}, в определенных случаях направлял задачу на кластер с минимальной стоимостью энергии.

Результаты проведенного эксперимета показали, что стоимость энергии возможно снизить, однако при этом увеличивается среднее время ожидания в очереди. Конкретные значения зависят от плотности потока задач --- чем больше плотность, тем меньше возможностей для выбора кластера, и тем меньше величина экономии. В нашем эксперименте наибольшая величина экономии составила 50%. Это соответствует потокам с минимальной плотностью --- около 100 задач в сутки. Для потоков со средней плотнотью (около 2-4 тыс задач в сутки) величина экономии составила 20-15%. Для потоков с большой плотностью (от 10 тыс задач в сутки) экономия составляет не более 3%.

Таким образом, мы считаем, что использование информации о стоимости энергии каждого кластера при распределении задач способно существенно снизить расходы на электроэнергию для владельца распределенной вычислительной системы. Однако, стоит отметить, что в реальной жизни такое перераспределение задач не всегда возможно. Обычно дата-центры строятся для обслуживания пользователей в определенном регионе, чтобы уменьшить ``время отклика''. Также задачи могут работать с большими объемами локальных данных, которые не так просто переместить с одного кластера на другой. Несмотря на это многие компании уже используют как альтернативные источники энергии для питания своих дата-центров, так и особенности географического расположения для снижения расходов на электроэнергию [12,13]. Так, известная интернет компания Facebook планирует запустить в 2012 году дата-центр на севере Швеции в городе Лулео (Luleå), удалённом почти на тысячу километров к северу от Стокгольма и находящегося в 100 километрах от Полярного круга. В 2009 году компания Google приобрела здание бумажного комбината в Финляндии в городе Хамина и переоборудовала его в дата-центр, где для охлаждения используется вода из Балтийского моря, что также снижает расходы на электроэнергию. Перечисленные факты говорят о важности энергоэффективности в современных распределенных высокопроизводительных вычислительных системах и тенденции учета географического расположения для снижения стоимости вычислений.

Список литературы

[1] Иванников В.П., Грушин Д.А., Кузюрин Н.Н. и др. Программная система увеличения энергоэффективности вычислительного кластера // Программирование. — 2010. — Т. 6. — С. 28–40.

[2] Albers S. Algorithms for Energy Saving // Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday. — 2009. — P. 173–186.

[3] S. Albers, H. Fujiwara. Energy-Efficient Algorithms for Flow Time Minimization // Lecture Notes in Computer Science. — 2006. — Vol. 3884. — P. 621–633.

[4] Augustine J, Irani S, Swamy C. Optimal power-down strategies // SIAM Journal on Computing. — 2008. — Vol. 37. — P. 1499–1516.

[5] Irani S, Shukla S K, Gupta R. Algorithms for power savings // ACM Transactions on Algorithms. — 2007. — Vol. 3.

[6] Irani, Pruhs. Algorithmic problems in power management // SIGACT News. — 2005. — Vol. 36, no. 2. — P. 63–76.

[7] Zhang, Chatha. Approximation algorithm for the temperature-aware scheduling problem // ICCAD ’07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design. — Piscataway, NJ, USA: IEEE Press, 2007. — P. 281–288.

[8] A Karlin, M Manasse, L McGeoch, S Owicki. Randomized competitive algorithms for nonuniform problems // ACM-SIAM Symposium on Discrete Algorithms. — 1990. — P. 301–309.

[9] Top500 supercomputer sites. — 2011. — November. — www.top500.org.

[10] Energy price statistics. — 2011. — November. — http://epp.eurostat.ec.europa.eu.

[11] David Jackson, Quinn Snell, Mark Clement. Core Algorithms of the Maui Scheduler // Job Scheduling Strategies for Parallel Processing / Ed. by D. Feitelson, L. Rudolph. — Springer Berlin / Heidelberg, 2001. — Vol. 2221 of Lecture Notes in Computer Science. — P. 87–102.

[12] Yevgeniy Sverdlik. Microsoft gets wind power for Dublin data center // http://www.datacenterdynamics.com. — 2011.

[13] Ward Van Heddeghem, Willem Vereeckena, Didier Collea et al. Distributed computing for carbon footprint reduction by exploiting low-footprint energy availability // Future Generation Computer Systems. — 2012. — Vol. 28. — P. 405– 414.

Energy-efficient computations on a group of clusters

D.A. Grushin (ISP RAS, Moscow, Russia, dgrushin@gmail.com),
N.N. Kuzyurin (ISP RAS, Moscow, Russia, nnkuz@ispras.ru)

Abstract. The problem of scheduling parallel tasks on a group of geographically distributed clusters optimizing energy-efficiency of computations is considered. Some scheduling algorithms are proposed and experimentally investigated. Methods for reducing the energy consumption of a uniform computer cluster due to flexible control strategies of the node states (waking them up or shutting down) and of the execution order of the awaiting tasks are considered. A software system developed in the Institute for System Programming of the Russian Academy of Sciences (ISP RAS) for the dynamic control of nodes in order to reduce the energy consumption is described. Several strategies for controlling the states of the nodes are proposed and investigated. Our simulation showed that when the average density of tasks is 0.5, the energy saving is about 10%. When the density of the flow of tasks decreases, the effect of using the proposed system drastically increases: when the average density is 0.3, the saving is 30%; when the average density is 0.2, the saving is 50%; and when the average density is 0.1, the saving is 70%.

Keywords: energy-efficient computations; load balancing; distributed computing

References

[1]. Ivannikov V. P., Grushin D. A., Kuzyurin N. N., Pospelov A. I., and Shokurov A. V. Software for Improving the Energy Efficiency of a Computer Cluster. Programming and Computer Software, November 2010, Volume 36, Issue 6, pp 327-336
[2]. Albers S. Algorithms for Energy Saving. Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday. 2009. P. 173–186.
[3]. Albers S., Fujiwara H. Energy-Efficient Algorithms for Flow Time Minimization. Lecture Notes in Computer Science. 2006. V. 3884. P. 621–633.
[4]. Augustine J., Irani S., Swamy C., Optimal power-down strategies. SIAM Journal on Computing. 2008. V. 37. P. 1499–1516.
[5]. Irani S., Shukla S. K., Gupta R. Algorithms for power savings. ACM Transactions on Algorithms. 2007. V. 3.
[6]. Irani S., Pruhs. Algorithmic problems in power management. SIGACT News. 2005. V. 36, no. 2. P. 63–76.
[7]. Zhang, Chatha. Approximation algorithm for the temperature-aware scheduling problem. ICCAD ’07: Proceedings of the 2007 IEEE-ACM international conference on Computer-aided design. Piscataway, NJ, USA: IEEE Press, 2007. P. 281–288.
[8]. Karlin A., Manasse M., McGeoch L., Owicki S. Randomized competitive algorithms for nonuniform problems. ACM-SIAM Symposium on Discrete Algorithms. 1990. P. 301–309.
[9]. Top500 supercomputer sites. 2011. November. www.top500.org.
[10]. Energy price statistics. 2011. November. http://epp.eurostat.ec.europa.eu.

[11]. Jackson D., Snell Q., Clement M.. Core Algorithms of the Maui Scheduler.Job Scheduling Strategies for Parallel Processing. Ed. by D. Feitelson, L. Rudolph. Springer Berlin. Heidelberg, 2001. V. 2221 of Lecture Notes in Computer Science. P. 87–102.
[12]. Sverdlik Y. Microsoft gets wind power for Dublin data center. http://www.datacenterdynamics.com. 2011.
[13]. Van Heddeghem W., Vereeckena W., Collea D. et al. Distributed computing for carbon footprint reduction by exploiting low-footprint energy availability. Future Generation Computer Systems. 2012. V. 28. P. 405– 414.

О построении расписаний выполнения параллельных задач на группах кластеров с различной производительностью

Жук С.Н.,
Princeton University (szhuk@princeton.edu)

Аннотация. Предложен онлайновый алгоритм распределения параллельных задач на группе кластеров с различными производительностями процессоров и показано, что он гарантирует для любого потока задач построение расписания, отличающегося от оптимального не более чем в 2e раз.

Ключевые слова: параллельные вычисления, балансировка нагрузки, расписания выполнения задач для кластеров с различной производительностью.

1. Введение

Рассмотрена задача построения расписаний выполнения параллельных задач на группах кластеров с различной производительностью процессоров. Имеется группа кластеров с различным числом процессоров и различной производительностью процессоров (у разных кластеров) и некоторое число многопроцессорных задач, каждая из которых характеризуется требуемым числом процессоров и требуемым нормализованным временем выполнения (временем выполнения на кластере единичной производительности). Необходимо составить расписание распределения задач по кластерам с целью минимизации общего времени выполнения. Предполагается, что выполнение задачи не может быть прервано во время своего выполнения.

Данная задача является NP-трудной (даже для частных случаев). Соответственно, неизвестны эффективные алгоритмы ее точного решения. В настоящей статье предложен приближенный алгоритм, позволяющий быстро находить приближенное решение и получены оценки его точности.

Задача составления расписания для мультипроцессорных задач на группе кластеров тесно связана с задачей упаковки прямоугольников в несколько полос. Кластеры можно рассматривать как полубесконечные полосы с шириной, соответствующей общему числу процессоров, а задачи – как прямоугольники с шириной, соответствующей требуемому числу

процессоров, и высотой, соответствующей требуемому времени. Единственным существенным отличием является то, что в том случае, когда рассматриваются прямоугольники, соответствующая задача должна занимать последовательные процессоры, в то время как в действительности задача может занимать произвольное множество процессоров. Принципиальным для данной работы является тот факт, что кластеры могут иметь различную производительность. Это существенно, поскольку даже для задачи построения расписаний для однопроцессорных машин с различными производительностями результаты далеки от окончательных [1,2].

В данной работе предложен онлайновый алгоритм распределения параллельных задач на группе кластеров с различными производительностями процессоров и показано, что он гарантирует для любого потока задач построение расписания, отличающегося от оптимального не более, чем в 2e раз. Этот результат является обобщением результатов, полученных для распределения параллельных задач на группе кластеров с одинаковыми процессорами [3].

2. Обозначения и полученные результаты

Для того чтобы определить, насколько хорошим является предложенный приближенный алгоритм, используем следующие понятия и обозначения.

Пусть T – некоторая последовательность мультипроцессорных задач, C – множество кластеров: $Opt(T,C)$, и $A(T,C)$ – оптимальное время выполнения и время выполнения для расписания (соответственно), полученного при помощи алгоритма A. Далее предположим, что требуемое нормализованное время ограничено сверху некоторой константой h_{max}. Тогда абсолютная точность может быть вычислена по следующей формуле:

$$R_A = \sup_{T,C} \left\{ A(T,C) \Big/ Opt(T,C) \right\}$$

а асимптотическая точность определяется следующим образом:

$$R_A^{\infty} = \lim_{k \to \infty} \sup_{T,C} \left\{ A(T,C) \Big/ Opt(T,C) \,\middle|\, Opt(T,C) \geq k \right\}$$

Особый интерес представляют так называемые онлайновые алгоритмы (on-line algorithms). Такие алгоритмы, получая задачи одну за другой, каждую задачу распределяют сразу, и не меняют в дальнейшем ее распределение. В онлайновых алгоритмах предполагается, что заранее не существует никакой

информации об остальных задачах и последовательности их поступления. Обычно онлайновые алгоритмы более удобны для использования на практике, так как далеко не всегда с самого начала известны все размещаемые объекты. Однако достаточно сложно получить онлайновый алгоритм с хорошим качеством аппроксимации.

В данной работе получены следующие результаты:

- Показано (Теорема 1), что любой онлайновый алгоритм, распределяющий многопроцессорные задания на группе кластеров с произвольными производительностями процессоров не может иметь асимптотическую точность, меньшую чем *e* (здесь *e* – основание натурального логарифма).
- Предложен полиномиальный онлайновый алгоритм с асимптотической точностью близкой к *2e* (Теорема 3).

Введем некоторые обозначения. Пусть $T=\{T_1,\ldots,T_n\}$ - конечная последовательность задач, $h(T_j)$ и $w(T_j)$ - требуемое нормализованное время и требуемое число процессоров для *j*-й задачи. Требуемое нормализованное время ограничено сверху величиной h_{max}.

Пусть $C=\{C_1,\ldots,C_m\}$ - множество кластеров, w_i - число процессоров в кластере, α_i - производительность кластера (это означает, что кластер может обработать задачу с требуемым нормализованным временем *h* за $\frac{h}{\alpha_i}$).

Предположим, что производительности ограничены снизу некоторой константой $\alpha_{\min}$.

Задача: Найти распределение для заданных мультипроцессорных задач *T* на заданном множестве кластеров *C,* минимизируя общее время выполнения.

Ниже будем предполагать, что $w_1 \geq w_2 \geq \ldots \geq w_m$.

Обозначим $\tilde{w}_i = w_i \cdot \alpha_i$ (*i*=1,…,*m*) общую производительность некоторого кластера.

Предположим, что существуют две последовательности задач $T^1=\{T_1^1,\ldots,T_n^1\}$ и $T^2=\{T_1^2,\ldots,T_n^2\}$. Обозначим через T^1+T^2 объединенную последовательность:

$$T^1+T^2=\{T_1^1,\ldots,T_n^1,T_1^2,\ldots,T_n^2\}$$

Предположим, R – некоторая многопроцессорная задача. Обозначим:

$$last(R)=\max k \ : \ w_k \geq w(R)$$

Теперь можно разбить все задачи по классам $M_1,\ldots,M_m$ следующим образом:

$$R\in M_i \Leftrightarrow last(R)=i$$

Предположим, что $S(R)=h(r)\cdot w(R)$ - общий объем вычислений, требуемый для некоторой задачи *R*. Если *Q* – некоторое множество задач, тогда через $S(Q)$ обозначим общий объем вычислений, требуемый для всех задач из множества *Q*.

Предположим, что *T* – последовательность задач, тогда обозначим

$$S_i(T)=S(\{R\in T \mid R\in M\})$$

Предположим, что имеется некоторый алгоритм *A*, который осуществляет распределение задач по кластерам. Обозначим через *A(T)* вектор, содержащий *m* значений $y=(y_1,\ldots y_m)$, где y_k - общий объем требуемых вычислений для задач из последовательности *T*, которые распределены алгоритмом *A* на кластер с номером *k*.

Введем следующие обозначения:

$$d_i = \sum_{j=1}^{i} \tilde{w}_i,$$

$$D_i(T) = \sum_{j=1}^{i} S_j(T),$$

$$h(T) = \max_i \frac{D_i(T)}{d_i}.$$

Когда добавляется новая задача, $D_i(T)$ может только возрасти. Следовательно, $h(T)$ также может только возрасти. Приведем сейчас достаточно простую нижнюю оценку.

Лемма 1. Для любого множества кластеров *C* и любой последовательности многопроцессорных задач *T*

$$Opt(T,C) \geq h(T)$$

Достаточно простым обобщением нижней оценки из [3] является следующая

Теорема 1. Для любого онлайнового алгоритма *A* выполняется следующее неравенство

$$R_A^{\infty} \geq e$$

Однако основная трудность состоит в получении верхней оценки асимптотической точности для некоторого онлайнового алгоритма. Эта задача будет рассмотрена в следующем разделе.

3. Алгоритм с асимптотической точностью 2e.

Для задачи составления расписания внутри одного кластера используем так называемый «шельфовый» алгоритм. Он был подробно представлен в работах по упаковке прямоугольников в полубесконечную полосу.

Предположим, имеется некоторая константа $r \in (0,1)$ Шельфом будем называть конкретный интервал времени, в течение которого задача может быть выполнена на данном кластере. Когда приходит новая задача, алгоритм определяет целое число *k*, такое что $r^{k+1} < h(R) \leq r^k$ и далее эта задача помещается в шельф длиной r^k (под длиной шельфа понимается длина интервала, соответствующего данному шельфу в терминах нормализованного времени). Задача может быть помещена в один из существующих шельфов (если в нем существует достаточное число свободных процессоров) или может быть создан новый шельф. Внутри каждого шельфа задачи располагаются последовательно горизонтально (каждая новая задача занимает требуемое число процессоров на весь период времени шельфа).

Таким образом распределение многопроцессорных задач, которые попадают в один класс (каждой из этих задач соответствует одно значение *k*) выполняется при помощи некоторой эвристики, которая представляет собой одномерную упаковку в контейнеры (one-dimensional bin-packing). Будем использовать эвристику первого подходящего (First Fit), которая помещает задачу в первый шельф, который подходит или создает новый шельф, если подходящего шельфа не оказалось.

Описанный выше алгоритм будем называть *Shelf(r).*

3.1. Алгоритм A_r.

Алгоритм распределяет многопроцессорные задачи в режиме онлайн. Предположим, что последовательность задач *T* уже распределена и предположим, что

$$A_r(T) = y = (y_1, \ldots y_k)$$

Каждая новая задача *R* обрабатывается алгоритмом следующим образом:

- Вычисляется *h=h(T+|R|)*.
- Определяется номер кластера

$$k = \max_{w(R)<w_i} i \quad \text{и} \quad \frac{y_i}{\tilde{w}_i} \le eh$$

Если такое число *k* существует, то задача помещается на кластер *k*, в противном случае алгоритм останавливается. В последнем случае остается некоторое число нераспределенных задач (что означает, что алгоритм работает некорректно).

- Для распределения задач внутри кластеров используется алгоритм *Shelf(r)*.

Теорема 2. Для любого множества кластеров *C* и любой последовательности многопроцессорных задач *T* алгоритм A_r распределяет все задания из *T*.

Теорема 3. Для алгоритма A_r выполняется следующее неравенство

$$R_{A_r}^{\infty} \le 2e/r$$

Для доказательства данной теоремы необходимо следующее свойство шельфового алгоритма.

Лемма 2. Рассмотрим некоторый кластер. Предположим, *S* – общий объем вычислений требуемый для выполнения всех задач, за исключением последнего, которое помещено в кластер. Пусть *w* – число процессоров, α - производительность кластера, h_{max} – наибольшее из возможных нормализованное требуемое время для некоторой задачи. Тогда для времени выполнения на этом кластере, полученном в соответствии с алгоритмом распределения *Shelf(r)*, выполняется следующее неравенство:

$$Shelf(r) \le \frac{2}{r} \cdot \frac{S}{w\alpha} + \frac{h_{\max}}{\alpha}\left(\frac{1}{r(1-r)} + 1\right)$$

Лемма 3. Для любого множества кластеров *C* и любой последовательности многопроцессорных задач *T* выполняется следующее неравенство:

$$A_r(T,C) \le \frac{2e}{r} \cdot Opt(T,C) + \left(\frac{1}{r(1-r)} + 1\right) \cdot \frac{h_{\max}}{\alpha_{\min}}$$

Доказательство. Предположим, максимальное время выполнения достигается на кластере с номером *j*. Пусть *R* – последняя задача, распределенная на этот кластер, и предположим, что общий объем вычислений, требуемых для всех остальных задач - *S*. Поскольку задача распределена на данный кластер

$$\frac{S}{\tilde{w}_j} \le eh(T) \le e \cdot Opt(T,C)$$

Используя лемму 2 для кластера с номером *j*, получаем:

$$A_r(T,C) \le \frac{2}{r} \cdot \frac{S}{\alpha_j w_j} + \left(\frac{1}{r(1-r)} + 1 \right) \frac{h_{\max}}{\alpha_j} \le \frac{2e}{r} \cdot Opt(T,C) + \left(\frac{1}{r(1-r)} + 1 \right) \cdot \frac{h_{\max}}{\alpha_{\min}}$$

Лемма 3 доказана. Теорема 3 прямо следует из леммы 3.

Список литературы

[1] Bernam P., Charikar M., Karpinski M., On-line load balancing for related machines, Proc. WADS, 1997, LNCS, 1272, Springer-Ferlag, p. 116-125.

[2] Aspens J., Azar Y., Fiat A., Plotkin S., Waarts O., On-line load balancing with applications to machine scheduling and virtual circuit routing, Proc. 25th ACM STOC, 1993, p. 623-631.

[3] Жук С.Н., Приближенные онлайновые алгоритмы упаковки прямоугольников в несколько полос, Дискретная математика, 2007, т. 19, N 4.

Унификация программ

Т.А. Новикова, В.А.Захаров
{taniaelf@mail.ru}{zakh@cs.msu.su}

Аннотация. Унифицировать два алгебраических выражения t_1 и t_2 означает отыскать такую подстановку θ термов вместо переменных этих выражений, чтобы оба терма $t_1\theta$ и $t_2\theta$ имели одинаковое значение. Задачу унификации можно распространить и на программы. Унифицировать две программы π_1 и π_2 означает отыскать такие цепочки присваиваний $\rho_1 : x_1 := t_1; x_2 := t_2; \ldots x_n := t_n$ и $\rho_2 : y_1 := s_1; x_2 := s_2; \ldots x_m := s_m$, для которых композиции программ $\rho_1;\pi_1$ и $\rho_2;\pi_2$ эквивалентны (т.е. вычисляют одну и ту же функцию). В данной работе в качестве эквивалентности программ рассматривается отношение логико-термальная эквивалентности, одно из наиболее слабых разрешимых отношений эквивалентности программ, аппроксимирующих отношение функциональной эквивалентности. Опираясь на алгоритм проверки логико-термальной эквивалентности программ, мы предлагаем полиномиальную по времени процедуру вычисления наиболее общего унификатора (ρ_1, ρ_2) для произвольной пары (π_1, π_2) последовательных императивных программ относительно логико-термальной эквивалентности.

Ключевые слова: программа, логико-термальная эквивалентность программ, подстановка, унификация, сложность.

1. Введение

Задача унификации выражений (формул, атомов, термов) языка предикатов первого порядка состоит в том, чтобы для заданной пары выражений $E_1(x_1, \ldots, x_n)$ и $E_2(x_1, \ldots, x_n)$ отыскать такую подстановку $\theta = \{x_1 / t_1, \ldots, x_n / t_n\}$, для которой выражения $E_2(x_1, \ldots, x_n)\theta$ и $E_1(x_1, \ldots, x_n)\theta$ становятся синтаксически одинаковыми. Впервые задачу унификации исследовал Дж. Робинсон в статье [1] при разработке метода

резолюций для систем автоматического доказательства теорем. Метод резолюций послужил отправной точкой для разработки концепции логического программирования, и алгоритмы унификации стали основным средством вычисления логических программ. За прошедшие годы задача унификации была детально исследована. В частности, были разработаны эффективные алгоритмы унификации [2-4], имеющие почти линейную сложность, а также были найдены подходящие структуры данных для практической реализации этих алгоритмов.

Наряду с задачей синтаксической унификации в алгебре свободных термов изучалась также и более общая задача семантической унификации (или, иначе, E-унификации). В алгебре термов можно ввести определяющие тождества (например, законы коммутативности, ассоциативности, дистрибутивности и др.) для некоторых функций, и индуцировать тем самым отношение конгруэнтности на множестве термов. Тогда задача семантической унификации состоит в том, чтобы для заданной пары термов E_1 и E_2 отыскать такую подстановку θ, для которой термы $E_2\theta$ и $E_1\theta$ эквивалентны. Задача семантической унификации исследовалась для алгебраических операций, подчиняющихся законам ассоциативности и коммутативности [5-7]. Некоторые из методов, предложенных в этих статьях, сводят задачу унификации к решению систем диофантовых уравнений. Было установлено также, что задача унификации выражений в логиках высокого порядка алгоритмически неразрешима (см. [8]). С методами и результатами решения задачи унификации термов и формул можно ознакомиться в обзорной статье [9].

Алгебраический терм является, в определенном смысле, простейшей разновидностью вычислительных программ, при построении которых используется только операция композиции. Более сложные программы конструируются при помощи более широкого набора операций. Применительно к вычислительным императивным программам задачу унификации можно сформулировать в такой постановке. Предположим, что на множестве программ введено некоторое отношение эквивалентности и выделен некоторый класс «простых» программ Π. Тогда для произвольной заданной пары программ π_1 и π_2 требуется выяснить, существует ли две такие пары программ (π'_{in}, π''_{in}) и $(\pi'_{out}, \pi''_{out})$ из класса Π, для которой последовательные композиции программ $\pi'_{in}; \pi_1; \pi'_{out}$ и $\pi''_{in}; \pi_2; \pi''_{out}$ эквивалентны. В качестве отношения эквивалентности разумно выбрать отношение функциональной эквивалентности программ или любую его аппроксимацию; в этом случае из эквивалентности программ будет следовать равенство вычисляемых этими программами функций. Пары программ (π'_{in}, π''_{in}) и $(\pi'_{out}, \pi''_{out})$ в постановке задачи унификации могут мыслиться

как интерфейсы, преобразующие формат представления входных и выходных данных. Можно предполагать, что преобразования такого рода выполняются программами, имеющими сравнительно простое устройство. Тогда эквивалентность композиций $\pi'_{in};\pi_1;\pi'_{out}$ и $\pi''_{in};\pi_2;\pi''_{out}$ означает, что программы π_1 и π_2 вычисляют схожие функции, и отношение унифицируемости программ является одной из возможных формализаций отношения подобия, обобщающего отношение эквивалентности программ.

Различные отношения подобия программ находит применение в решении задач реорганизации (рефакторинга) программ (см. [10]). В частности, актуальными задачами являются проблемы формализации понятия программного клона и эффективного выделения клонов [11]. Содержательно, программный клон – это совокупность фрагментов программы, осуществляющих «похожие» преобразования данных. Корректное определение программного клона, сопровождаемое эффективным методом обнаружения клонов, позволяет проводить упрощения программного код путем замены нескольких больших фрагментов программы вызовом одной и той же процедуры или макроса [12]. Одно из возможных определений клона можно сформулировать на основе отношения унифицируемости: клоном называется совокупность фрагментов программы $\pi_0,\pi_1,\ldots,\pi_n$, обладающая следующим свойством – любая пара программ $\pi_0,\pi_i,\ \ 1\le i\le n,$ унифицируема, причем унифицирующими интерфейсами служат такие пары программ (π'_{in},π''_{in}) и (π'_{out},π''_{out}), в которых π'_{in} и π'_{out} – это пустые программы, вычисляющие тождественные функции. В этом случае при обнаружении клона мы можем ввести в программе новую процедуру $PROC$, телом которой служит фрагмент π_0, а все фрагменты $\pi_i, 1\le i\le n,$ заменить вызовом этой процедуры в составе композиций вида $\pi'_{in};call\ PROC;\pi'_{out}, 0\le i\le n$.

Наиболее важным параметром в формулировке задачи унификации программ является отношение эквивалентности программ. Содержательный смысл задачи унификации требует, чтобы это отношение аппроксимировало отношение функциональной эквивалентности. В то же время, для эффективного решения задачи унификации необходимо, чтобы это выбранное отношение эквивалентности было разрешимым. Обоим требованиям удовлетворяет отношение логико-термальной (л-т) эквивалентности, введенное в статье [13]. Две программы π_1 и π_2 считаются л-т эквивалентными, если для любой синтаксически допустимой трассы tr' в одной из программ существует такая трасса tr'' в другой программе, что в обеих трассах логические условия (предикаты) проверяются в одной и той же последовательности для одних и тех же наборов значений переменных. Как

было установлено в [13], л-т эквивалентность программ влечет их функциональную эквивалентность в любой интерпретации базовых операций и предикатов. В статье [14] показано, что проверку л-т эквивалентности программ можно провести за время, полиномиально зависящее от размеров программ. Еще более быстрый алгоритм проверки л-т эквивалентности программ был предложен в статьях [15,16]. Опираясь на эти результаты, мы покажем, что для одного класса интерфейсов-унификаторов задачу л-т унификации императивных программ также можно решить за полиномиальное время.

Содержание статьи таково. Во втором и третьем разделах мы напомним основные понятия алгебры конечных подстановок [17-18] и теории стандартных схем программ [20], которые понадобятся для формулировки и описания решения задачи л-т унификации программ. В четвертом разделе приведено упрощенное описание алгоритма вычисления наиболее общего л-т унификатора программ и обоснована его корректность. В пятом разделе показано, каким образом можно улучшить вычислительные качества описанного алгоритма л-т унификации программ, для того чтобы получить полиномиальную верхнюю оценку сложности модифицированного алгоритма. В заключение обсуждаются перспективы дальнейшего развития предложенного нами метода для решения более общих случаев проблемы унификации программ.

2. Алгебра конечных подстановок

Для заданных множеств переменных Var, функциональных символов F и предикатных символов P обозначим записью $Term(F,Var)$ множество термов, а записью $Atom(P,F,Var)$ множество атомарных формул (атомов). Для каждого терма t обозначим записью Var_t множество переменных, входящих в терм t.

Для представления термов будем использовать размеченные ациклические графы (АОГ). Рассмотрим произвольный конечный ациклический ориентированный граф $G=(V,E)$ с множеством вершин V и множеством дуг E. Размером $|G|$ графа G будем считать суммарное количество его вершин и дуг. *Разметкой* АОГ G назовем отображение, которое приписывает каждой вершине графа v символ s_v, который является либо функциональным символом из множества F, либо переменной из множества Var, а каждой дуге – некоторое натуральное число. *Правильной разметкой* АОГ G назовем *разметку*, удовлетворяющую следующим двум требованиям:

- если из вершины v не исходит ни одной дуги, то эта вершина помечена либо константой, либо переменной;
- если из вершины v исходят n дуг, где $n > 0$, то эта вершина помечена функциональным символом местности n, а все исходящие дуги помечены попарно различными числами из множества $\{1,2,\ldots,n\}$.

Если в правильно помеченном АОГ G из вершины v в вершину u ведет дуга, помеченная числом m, то вершину u будем называть m *-наследником* вершины v. Каждой вершине v правильно размеченного АОГ G можно однозначно сопоставить терм t_v, который *реализуется в вершине* v, руководствуясь следующими правилами:

- если из вершины v не исходит ни одной дуги, то в этой вершине реализуется терм s_v;
- если из вершины v исходят n дуг, ведущие в вершины $u_1, u_2, \ldots, u_n$, и при этом для любого $i, 1 \le i \le n,$ дуга, ведущая в вершину u_i, помечена натуральным числом i, то в вершине v реализуется терм $s_v(t_{u_1}, t_{u_2}, \ldots, t_{u_n})$.

Правильно размеченный АОГ G *реализует* конечное множество термов $T, T \subseteq Term(F,Var),$ если для каждого терма $t, t \in T,$ в графе G существует вершина v, для которой $t_v = t$.

АОГ G, реализующий множество термов T, называется *приведенным*, если он удовлетворяет двум требованиям:

- в каждой вершине АОГ G реализуется некоторый подтерм какого-либо терма из множества T;
- в разных вершинах АОГ G реализуются разные термы.

Для построения приведенного АОГ из произвольного графового представления множеств термов нужно выполнить специальную процедуру редукции, которая за время, линейное относительно размера исходного графового представления строит приведенный АОГ, реализующий то же самое множество термов (см. [16]).

Пусть X и Y – два множества переменных. Подстановкой назовем всякое отображение $\theta : X \to Term(F,Y)$, сопоставляющее каждой переменной из множества X некоторый терм. Множество всех таких подстановок

обозначим записью $Subst(X,F,Y)$. Если $X=\{x_1,x_2,\ldots,x_n\}$ и $\theta(x_i)=t_i$ для всех $i,\ 1\le i\le n,$ то подстановка θ однозначно определяется множеством (списком) пар $\{x_1/t_1,x_2/t_2,\ldots,x_n/t_n\}$. Запись Var_θ будет использоваться для обозначения множества переменных $\bigcup_{i=1}^{n} Var_{t_i}$, входящих в состав всех термов подстановки θ. Результатом применения подстановки θ к терму t является терм $t\theta$, получающийся одновременной заменой в t каждой переменной x_i термом $\theta(x_i)$. Композиция $\theta\eta$ подстановок $\theta\in Subst(X,F,Y)$, $\eta\in Subst(Y,F,Z)$ – это подстановка из множества $Subst(X,F,Z)$, которая определяется равенством $\theta\eta(x)=(\theta(x))\eta$ для каждой переменной $x,\ x\in X$. Всякая биекция $\rho:Y\to Y$ называется переименованием. Подстановки θ_1 и θ_2 считаются эквивалентными (и это отношение обозначается записью $\theta_1\approx\theta_2$), если для некоторого переименования ρ выполняется равенство $\theta_2=\theta_1\rho$.

Размеченные АОГ можно использовать также и для реализации подстановок. Пусть заданы подстановка $\theta=\{x_1/t_1,x_2/t_2,\ldots,x_n/t_n\}$ из множества $Subst(X,F,Y)$ и АОГ G, реализующий множество термов $\{t_1,t_2,\ldots,t_n\}$. Сопоставим каждой переменной $x_i,1\le i\le n$, ту вершину v графа G, в которой реализуется соответствующий терм t_i; переменную x_i будем называть *заголовком* вершины v. Размеченный таким образом АОГ G будем называть *графовой реализацией* подстановки θ и использовать запись $|\theta|$ для обозначения размера этого графа.

На множестве подстановок $Subst(X,F,Y)$ определим отношения предпорядка $\prec$: для пары подстановок θ_1,θ_2 отношение $\theta_1\prec\theta_2$ выполняется, если есть такая подстановка $\eta\in Subst(Y,F,Y)$, что $\theta_2=\theta_1\eta$. В этом случае подстановку θ_1 будем называть *прототипом* подстановки θ_2, подстановку θ_2 – *примером* подстановки θ_1. Подстановку η будем называть *пополнением* прототипа θ_1 до примера θ_2 и использовать для ее обозначения запись $\frac{\theta_2}{\theta_1}$.

Прототип $\theta_1, \theta_1 \in Subst(X, F, X \cup Y),$ подстановки $\theta_2, \theta_2 \in Subst(X, F, X \cup Y)$ б назовем *редуцированным*, если для любой переменной $y,\ y \in Var_{\theta_1} \cap Y,$ существует такая переменная x, для которой $\theta_1(x) = y$. *Наиболее специальной редукцией* подстановки θ_2 называется такая ее редукция θ_1, которая удовлетворяет отношению $\theta' \prec \theta_1$ для любой редукции θ' подстановки θ_2. В статье [16] показано, что для каждой подстановки θ существует единственная наиболее специальная редукция $msr(\theta)$ (с точностью до переименования переменных); $msr(\theta)$ может быть построена за время, линейное относительно размера графового представления подстановки θ.

Отношение предпорядка $\prec$ позволяет ввести на множестве подстановок $Subst(X, F, X \cup Y)$ операции точной верхней грани $\mathrm{lub}(\theta_1, \theta_2)$ и точной нижней грани $\mathrm{glb}(\theta_1, \theta_2)$. Первая из них вычисляет наиболее общий пример подстановок θ_1 и θ_2 и позволяет решать задачу унификации в свободной алгебре термов. Из описания алгоритма унификации термов [3] следует, что $|\mathrm{lub}(\theta_1, \theta_2)| = O(|\theta_1| + |\theta_2|)$ и сложность вычисления $\mathrm{lub}(\theta_1, \theta_2)$ пропорциональна максимальному из размеров подстановок $|\theta_1|$ и $|\theta_2|$. Вторая операция вычисляет наименее общий прототип подстановок θ_1 и θ_2 и называется антиунификацией. Чтобы операция антиунификации была всюду определенной, к множеству подстановок добавлена в качестве наибольшего элемента специальную мнимую подстановку τ, удовлетворяющую равенствам $\tau\theta = \theta\tau = \tau$ и $t_1\tau = t_2\tau$ для любой подстановки θ и термов t_1, t_2. В статье [20] показано, что сложность вычисления $\mathrm{glb}(\theta_1, \theta_2)$ и размер $|\mathrm{glb}(\theta_1, \theta_2)|$ оцениваются сверху величиной $O(|\theta_1| \times |\theta_2|)$. Такая сложность вычисления точной нижней грани подстановок затрудняет использование этой операции в итеративных алгоритмах статического анализа программ (см. [18]). Для преодоления этой трудности в статье [16] была предложена комбинированная операция *редуцированной антиунификации* подстановок $\theta_1 \Downarrow \theta_2 = msr(\mathrm{glb}(\theta_1, \theta_2))$. Как было установлено в [16], размер редуцированной антиунификации $\theta_1 \Downarrow \theta_2$ и сложность ее вычисления пропорциональны максимальному из размеров $|\theta_1|$ и $|\theta_2|$.

Для решения задачи унификации программ нам понадобится проводить унификацию атомов. Для заданного множества пар атомов

$H = \{(A_i', A_i'') : i \in I\}$ из класса $Atom(P, F, X)$ *унификатором* этого множества называется подстановка $\theta, \theta \in Subst(X, F, X)$, для которой равенство $A_i'\theta = A''\theta$ выполняется для любой пары атомов (A_i', A_i'') из множества H. Унификатор θ множества пар атомов H называется *наиболее общим унификатором*, если для любого унификатора θ' множества H выполняется соотношение $\theta \prec \theta'$. Задача унификации конечного множества пар атомов решается за время, линейное относительно суммарного размера атомов в множестве H (см. [3,4]).

3. Модель последовательных императивных программ

Мы рассматриваем модель последовательных императивных программ, заимствованную из работ [21,22] и адаптированную для работы с подстановками. Подробное описание этой модели приведено в статье [15].

Пусть задано некоторое конечное множество переменных X. Модель программы представляет собой размеченный ориентированный граф $\pi(X)$. Каждой вершине v этого графа приписана атомарная формула $A[v]$ из множества $Atom(P, F, X)$. Из каждой вершины исходят две дуги, одна из которых помечена символом 0, а другая – символом 1. Кроме того, каждой дуге, ведущей в графе $\pi(X)$ из вершины u в вершину v, приписана подстановка $\theta[uv]$ из множества $Subst(X, F, X)$. Одна из вершин v_{in} графа $\pi(X)$ особо выделена в качестве входа в программу, а другая вершина v_{out} играет роль выхода из программы. Предполагается также, что через каждую вершину графа $\pi(X)$ проходит некоторый маршрут, ведущий из входа программы в ее выход.

Пусть задан некоторый маршрут (трасса) из входа в программу $\pi(X)$ в ее выход

$$\alpha = v_0 \xrightarrow{\sigma_1, \theta_1} v_1 \xrightarrow{\sigma_2, \theta_2} v_2 \xrightarrow{\sigma_1, \theta_1} \cdots v_{n-1} \xrightarrow{\sigma_n, \theta_n} v_n,$$

где $\sigma_i \in \{0,1\}, \theta_i \in Subst(X, F, X), 1 \le i \le n$. Тогда последовательность пар

$$lth(\alpha) = (A[v_0]\mu_0, \sigma_1), (A[v_1]\mu_1, \sigma_2), \ldots, (A[v_{n-1}]\mu_{n-1}, \sigma_n), (A[v_n]\mu_n, 1),$$

где $\mu_0 = \varepsilon$ и $\mu_i = \theta_i \mu_{i-1}$ для каждого $i, 1 \le i \le n,$ называется *логико-термальной историей* трассы α. *Детерминантом* программы $\pi(X)$ называется множество

$$Det(\pi(X)) = \{lth(\alpha) : \alpha \text{ - трасса в программе } \pi(X) \}.$$

Две программы $\pi_1(X)$ и $\pi_2(X)$ считаются *логико-термально* (*л-т*) *эквивалентными*, если справедливо равенство $Det(\pi_1(X)) = Det(\pi_2(X))$.

Опишем операцию применения подстановок к программам. Пусть задана программа $\pi(X)$ и подстановка $\theta, \theta \in Subst(X, F, X)$. Тогда *результатом применения подстановки* θ *к программе* $\pi(X)$ является программа $\pi(X)\theta$, которая получается из программы $\pi(X)$ введением

- новой входной вершины v_θ, которой приписывается 0-местный атом P_0;
- двух дуг, исходящих из v_θ и ведущих в вершину v_{in}; одна из этих дуг помечается символом 0, а другая – символом 1, и каждой из дуг приписывается подстановка θ.

Программу $\pi(X)\theta$ можно истолковывать как вызов процедуры с телом $\pi(X)$, в котором инициализация переменных осуществляется подстановкой θ.

Пусть конечное множество переменных X разбито на два непересекающихся множества X' и X''. Тогда подстановка $\theta, \theta \in Subst(X, F, X)$, называется *л-т унификатором программ* $\pi'(X')$ и $\pi''(X'')$, если программы $\pi'(X')\theta$ и $\pi''(X'')\theta$ являются л-т эквивалентными. Унифицируемость программ означает, что при некоторой инициализации переменных эти программы вычисляют одну и ту же функцию. Л-т унификатор θ программ $\pi'(X')$ и $\pi''(X'')$ называется *наиболее общим унификатором* (обозначается записью $НОУ(\pi'(X'), \pi''(X''))$), если для любого л-т унификатора θ' этих программ выполняется соотношение $\theta \prec \theta'$. Задача л-т унификации программ состоит в вычислении $НОУ(\pi'(X'), \pi''(X''))$.

4. Алгоритм логико-термальной унификации программ

Для решения задачи л-т унификации программ $\pi'(X')$ и $\pi''(X'')$, так же как и для решения задачи проверки л-т эквивалентности программ, мы воспользуемся графом логически совместных трасс $\Gamma[\pi',\pi'']$. Его устройство таково. Вершинами графа $\Gamma[\pi',\pi'']$ служат всевозможные пары (u',u''), где u' - вершина программы π', а u'' - вершина программы π''. Каждой вершине (u',u'') приписывается пара атомов $(A[u'],A[u''])$, которыми были помечены вершины u' и u'' в программах π' и π''. Если в программе π' из вершины u' в вершину v' ведет дуга, помеченная символом $\sigma, \sigma \in \{0,1\}$, и этой дуге приписана подстановка $\theta' = \{x'_1 / t'_1, x'_2 / t'_2, \ldots, x'_n / t'_n\}$, а в программе π'' из вершины u'' в вершину v'' ведет дуга, помеченная тем же символом σ, и этой дуге приписана подстановка $\theta'' = \{x''_1 / t''_1, x''_2 / t''_2, \ldots, x''_m / t''_m\}$, то в графе $\Gamma[\pi',\pi'']$ из вершины (u',u'') в вершину (v',v'') ведет дуга с пометкой $\theta' \cup \theta'' = \{x'_1 / t'_1, x'_2 / t'_2, \ldots, x'_n / t'_n, x''_1 / t''_1, x''_2 / t''_2, \ldots, x''_m / t''_m\}$.

Для каждого пути

$$path = (u'_{\text{in}}, u''_{\text{in}}) \xrightarrow{\theta_1} (u'_1, u''_1) \xrightarrow{\theta_2} (u'_2, u''_2) \xrightarrow{\theta_3} \cdots (u'_{k-1}, u''_{k-1}) \xrightarrow{\theta_k} (u'_k, u''_k),$$

в графе логически совместных трасс $\Gamma[\pi',\pi'']$ запись $\theta[path]$ будет использоваться для обозначения композиции подстановок $\theta_k \cdots \theta_2 \theta_1$, приписанных дугам этого пути.

Полиномиальный по времени алгоритм проверки л-т эквивалентности программ, предложенный в статьях [15,16], основывается на следующем характеристическом свойстве графов логически совместных трасс программ.

Теорема 1 [15]. Программы $\pi'(X)$ и $\pi''(X)$ л-т эквивалентны в том и только том случае, когда для любой вершины каждого пути (u,v) графа логически совместных трасс $\Gamma[\pi',\pi'']$ и для любого пути $path$, ведущего из начальной вершины $(u'_{\text{in}}, u''_{\text{in}})$ в вершину (u,v), выполняется равенство $A[u]\theta[path] = A[v]\theta[path]$.

Из этой теоремы два важных для решения задачи л-т унификации программ следствия.

Следствие 1. Подстановка η является л-т унификатором программ $\pi'(X')$ и $\pi''(X'')$ тогда и только тогда, когда для любой вершины (u,v) графа логически совместных трасс $\Gamma[\pi',\pi'']$ и для любого пути $path$, ведущего из

начальной вершины $(u'_{\text{in}}, u''_{\text{in}})$ в вершину (u, v), выполняется равенство $A[u]\theta[path]\eta = A[v]\theta[path]\eta$.

Следствие 2. Пусть H - это множество пар $(A[u]\theta[path], A[v]\theta[path])$ для всех вершин (u, v) графа логически совместных трасс $\Gamma[\pi', \pi'']$ и всех путей $path$, ведущих в эти вершины. Тогда $НОУ(\pi'(X'), \pi''(X'')) = НОУ(H)$.

На основании этих следствий мы адаптируем алгоритм проверки л-т эквивалентности, предложенный в статье [16], для вычисления $НОУ(\pi'(X'), \pi''(X''))$.

Вначале опишем этот алгоритм в упрощенном виде, удобном для доказательства его корректности, а затем покажем, каким образом его можно модифицировать, чтобы достичь полиномиальной по времени трудоемкости.

Алгоритм вычисляет монотонно возрастающую последовательность подстановок $\rho_0 \prec \rho \prec \ldots \prec \rho_n$ из множества $Subst(X, F, X)$, начинающуюся тождественной подстановкой ρ_0. На каждом n-ом этапе его работы начальная вершина (u'_{in}, u''_{in}) графа $\Gamma[\pi', \pi'']$ помечается подстановкой ρ_n, и вслед за тем запускается процедура вычисления стационарной разметки вершин графа $\Gamma[\pi', \pi'']$ подстановками. По окончании работы процедуры очередная подстановка ρ_{n+1} указанной последовательности вычисляется путем композиции ρ_n и наиболее общего унификатора некоторого конечного множества пар атомов. Алгоритм завершает работу, когда $\rho_{n+1} \approx \rho_n$.

Процедура вычисления стационарной разметки вершин графа $\Gamma[\pi', \pi'']$ сопоставляет каждой вершине (u', u'') графа

1) подстановку $\eta[u'u'']$ из множества $Subst(X, F, X \cup Y)$, где $Y = \{y_1, y_2, \ldots\}$ – бесконечное множество переменных, отличных от переменных множества X, и
2) некоторое множество подстановок $S[u'u'']$ из класса $Subst(X, F, X)$.

В начале каждого n-го этапа работы алгоритма, $n \geq 0$, вершина (u'_{in}, u''_{in}) графа $\Gamma[\pi', \pi'']$, помечается подстановкой ρ_n, а все остальные вершины

графа $\Gamma[\pi',\pi'']$ помечаются максимальным в квазирешетке подстановок $(Subst(X'\cup X'',F,Y),\prec)$ элементом τ. Кроме того, вершине (u'_{in},u''_{in}) приписывается множество подстановок $S[u'_{in},u''_{in}]=\{\rho_n\}$, а всем остальным вершинам (u',u'') приписывается пустое множество подстановок $S[u',u'']=\varnothing$. Далее выполняется процедура вычисления стационарной разметки вершин графа. Для каждой пары вершин (u',u'') и (v',v''), помеченных подстановками $\eta[u'u'']$ и $\eta[v'v'']$ и соединенных дугой, которой приписана подстановка θ, вычисляется подстановка $\mu=\theta\eta[u'u'']\Downarrow\eta[v'v'']$. Если не выполняется эквивалентность $\mu\approx\eta[v'v'']$, то вершине (v',v'') вместо подстановки $\eta[v'v'']$ приписывается подстановка μ, а вместо множества подстановок $S[v'v'']$ приписывается множество подстановок $S[v'v'']\cup\{\theta\lambda:\lambda\in S[u'u'']\}$. Выполнение процедуры продолжается до тех пор, пока в графе $\Gamma[\pi',\pi'']$ не будет построена такая разметка, что для каждой пары вершин (u',u'') и (v',v'') выполняется эквивалентность $\eta[v'v'']\approx\theta\eta[u'u'']\Downarrow\eta[v'v'']$.

По завершении процедуры разметки для каждой вершины (u,v) графа $\Gamma[\pi',\pi'']$ проверяется равенство $A[u]\eta[uv]=A[v]\eta[uv]$. Если каждое из указанных равенств выполняется, то подстановка ρ_n объявляется наиболее общим унификатором программ $\pi'(X')$ и $\pi''(X'')$. В ином случае вычисляется наиболее общий унификатор

$$\tau=НОУ(\bigcup_{(u,v)\in\Gamma(\pi',\pi'')}\bigcup_{\lambda\in S[uv]}\{(A[u]\lambda,A[v]\lambda)\}).$$

Если указанное множество пар атомов неунифицируемо, то объявляется, что программы $\pi'(X')$ и $\pi''(X'')$ также не имеют унификатора. В противном случае вычисляется подстановка $\rho_{n+1}=\rho_n\tau$, и алгоритм л-т унификации программ переходит к следующему (n+1)-ому этапу вычисления.

Как было показано в статье [16], процедура вычисления стационарной разметки графа $\Gamma[\pi',\pi'']$ всегда завершает работу. Покажем, что последовательность подстановок $\rho_0,\rho_1,\ldots,\rho_n,\ldots$ является конечной.

Лемма 1. Предположим, что на n-ом этапе работы алгоритма вершине (v',v'') приписано множество подстановок $S[v'v'']$. Тогда для любой подстановки

$\lambda, \lambda \in S[v'v'']$, существует такой путь $path$, ведущий в графе логически совместных трасс из начальной вершины $(u'_{\text{in}}, u''_{\text{in}})$ в вершину (v', v''), для которого выполняется равенство $\lambda = \theta[path]\rho_n$.

Доказательство. Проводится индукцией по числу шагов работы процедуры вычисления стационарной разметки на n-ом этапе работы алгоритма. □

Лемма 2. Предположим, что на n-ом этапе работы алгоритма вершине (v', v'') приписана подстановка $\eta[v'v'']$ и множество подстановок $S[v'v'']$. Тогда $\eta[v'v''] = \Downarrow\limits_{\lambda \in S[v'v'']} \lambda$.

Доказательство. Проводится индукцией по числу шагов работы процедуры вычисления стационарной разметки. Покажем справедливость индуктивного перехода. Предположим, что $\eta[v'v''] = \Downarrow\limits_{\lambda \in S[v'v'']} \lambda$, $\eta[u'u''] = \Downarrow\limits_{\lambda \in S[u'u'']} \lambda$, и пусть из вершины (u', u'') в вершину (v', v'') ведет дуга, которой приписана подстановка θ. Тогда на основании закона левой дистрибутивности операции композиции подстановок относительно операции редуцированной антиунификации (см. [16]), справедливо равенство $\theta\eta[u'u''] = \Downarrow\limits_{\lambda \in S[u'u'']} \theta\lambda$. Поэтому верна цепочка равенств $\mu = \theta\eta[u'u''] \Downarrow \eta[v'v''] = (\Downarrow\limits_{\lambda \in S[u'u'']} \theta\lambda) \Downarrow (\Downarrow\limits_{\lambda \in S[v'v'']} \lambda) = \Downarrow\limits_{\lambda \in S[v'v''] \cup \{\theta\lambda : \lambda \in S[u'u'']\}} \lambda$.

□

Лемма 3. Пусть $\rho_0, \rho_1, \ldots, \rho_n, \rho_{n+1}, \ldots$ – последовательность подстановок, которую вычисляет описанный алгоритм л-т унификации. Тогда для любого $n, n \geq 0,$ справедливо соотношение $Var(\rho_{n+1}) \subset Var(\rho_n)$.

Доказательство. Рассмотрим n-ый этап работы алгоритма. Как утверждает лемма 1, для любой вершины (v', v'') каждая подстановка $\lambda, \lambda \in S[v'v'']$, представима в виде композиции $\lambda = \theta[path]\rho_n$ для некоторого пути $path$ в графе $\Gamma[\pi', \pi'']$. Это означает, что $Var_\lambda \subseteq Var_{\rho_n}$. Если алгоритм л-т унификации не завершил работу на n-ом этапе, то для некоторой вершины (u, v) графа $\Gamma[\pi', \pi'']$ имеет место соотношение $A[u]\eta[uv] \neq A[v]\eta[uv]$. Как следует из леммы 2, верно равенство $\eta[v'v''] = \Downarrow\limits_{\lambda \in S[v'v'']} \lambda$. В статье [16] было установлено, что для любой пары атомов $A, B \in Atom(P, F, X)$ и любой пары подстановок $\eta_1, \eta_2 \in Subst(X, F, Y)$ справедливо соотношение $A\eta_1 = B\eta_1 \wedge A\eta_2 = B\eta_2 \quad \Leftrightarrow \quad A(\eta_1 \Downarrow \eta_2) = B(\eta_1 \Downarrow \eta_2)$. Отсюда

следует, что для некоторой подстановки $\lambda,\ \lambda \in S[v'v'']$, выполняется соотношение $A[u]\lambda \neq A[v]\lambda$. Поэтому наиболее общий унификатор $\tau = НОУ(\bigcup_{(u,v)\in\Gamma(\pi',\pi'')} \bigcup_{\lambda\in S[uv]} \{(A[u]\lambda, A[v]\lambda)\})$ отличен от тождественной подстановки. Тогда, как видно из описания любого алгоритма л-т унификации (см., например, [4]), множество переменных Var_τ является собственным подмножеством множества переменных Var_λ хотя бы для одной подстановки $\lambda,\ \lambda \in S[v'v'']$. Следовательно, Var_τ является собственным подмножеством множества переменных Var_{ρ_n}, и справедливо строгое включение $Var(\rho_{n+1}) = Var(\rho_n\tau) \subset Var(\rho_n)$. □

Теорема 2. Алгоритм л-т унификации программ завершает работу для любой пары программ $\pi'(X)$ и $\pi''(X)$.

Доказательство. Из леммы 3 следует, что для последовательности подстановок $\rho_0, \rho_1, \ldots, \rho_n, \rho_{n+1}, \ldots$, вычисляемой алгоритмом л-т унификации, выполняется цепочка строгих включений $X = Var_{\rho_0} \supset Var_{\rho_1} \supset \ldots \supset Var_{\rho_n} \supset Var_{\rho_{n+1}} \supset \ldots$. Поскольку множество переменных X конечно, указанная последовательность подстановок также является конечной. □

Лемма 4. Если алгоритм л-т унификации программ завершает работу, вычислив подстановку ρ_n, то эта подстановка является л-т унификатором программ $\pi'(X)$ и $\pi''(X)$.

Доказательство. Как показано в статье [16], если вычислена стационарная разметка графа, и для некоторой вершины (u,v) выполняется равенство $A[u]\eta[uv] = A[v]\eta[uv]$, то для любого пути $path$, ведущего из начальной вершины $(u'_{\text{in}}, u''_{\text{in}})$ в вершину (u,v), выполняется равенство $A[u]\theta[path]\rho_n = A[v]\theta[path]\rho_n$. Тогда, согласно следствию 1 из теоремы 1, успешное завершение работы алгоритма л-т унификации с результатом ρ_n означает, что подстановка ρ_n является л-т унификатором программ $\pi'(X)$ и $\pi''(X)$. □

Лемма 5. Если программы $\pi'(X)$ и $\pi''(X)$ л-т унифицируемы, то алгоритм л-т унификации вычисляет подстановку ρ_n, и при этом $\rho_n \prec НОУ(\pi'(X'), \pi''(X''))$.

Доказательство. Нетрудно заметить, что наиболее общие унификаторы произвольных множеств пар атомов H_1 и H_2 обладают следующими двумя свойствами.

1. Если H_2 – унифицируемое множество пар атомов, и $H_1 \subseteq H_2$, то $НОУ(H_1) \prec НОУ(H_2)$;
2. Если $НОУ(H_1) = \tau$, то $НОУ(H_1 \cup H_2) \approx НОУ(H_1\tau \cup H_2\tau)$.

Таким образом, на основании следствия 2 теоремы 1, а также леммы 1 можно легко показать, применяя математическую индукцию по числу этапов работы алгоритма л-т унификации, что для л-т унифицируемых программ $\pi'(X)$ и $\pi''(X)$ каждая подстановка в последовательности $\rho_0, \rho_1, \ldots, \rho_n, \rho_{n+1}, \ldots$, которая вычисляется алгоритмом л-т унификации, является прототипом $НОУ(\pi'(X'), \pi''(X''))$. □

Из лемм 3, 4 и 5 вытекает теорема корректности предложенного нами алгоритма л-т унификации программ.

Теорема 3. Алгоритм л-т унификации программ объявляет о том, что подстановка ρ_n является наиболее общим л-т унификатором программ $\pi'(X)$ и $\pi''(X)$ тогда и только тогда, когда $\rho_n \in НОУ(\pi'(X'), \pi''(X''))$.

5. Сложность логико-термальной унификации программ

Описанный в предшествующем разделе алгоритм л-т унификации программ не является оптимальным, поскольку требует построения в явном виде множеств подстановок $S[uv]$, ассоциированных с вершинами. Как видно из описания алгоритма, размер этих множеств может оказаться величиной, экспоненциально зависящей от размеров программ $\pi'(X)$ и $\pi''(X)$. В этом разделе мы покажем, что подстановки из множеств $S[uv]$ можно представлять в неявном виде, используя лишь полиномиальный относительно размеров программ объем памяти.

Обозначим записью N суммарный размер программ $\pi'(X)$ и $\pi''(X)$. Тогда граф $\Gamma[\pi', \pi'']$ имеет не более N^2 вершин, и максимальный размер каждой подстановки, приписанной вершине этого графа, не превосходит величины $O(N^2)$.

На каждом этапе работы алгоритма л-т унификации программ выполняется процедура вычисления стационарной разметки вершин графа логически совместных трасс $\Gamma[\pi',\pi'']$ при помощи операции редуцированной антиунифкации: для каждой пары вершин (u',u'') и (v',v''), помеченных подстановками $\eta[u'u'']$ и $\eta[v'v'']$ и соединенных дугой, которой приписана подстановка θ, вычисляется подстановка $\mu = \theta\eta[u'u''] \Downarrow \eta[v'v'']$, которая (в случае неэквивалентности подстановок μ и $\eta[v'v'']$) приписывается вершине (v',v'') вместо прежней подстановки $\eta[u'u'']$. Обозначим записью $list[v'v'']$ список подстановок, которые были последовательно приписаны вершине (v',v'') по ходу работы алгоритма. Как следует из утверждения 10, приведенного в статье [16], длина такого списка (равная количеству изменений пометки вершины графа $\Gamma[\pi',\pi'']$) не превосходит величины $O(N^2)$.

При вычислении подстановки $\mu = \theta\eta[u'u''] \Downarrow \eta[v'v'']$ также вычисляются две подстановки-пополнения $\xi' = \dfrac{\mu}{\theta\eta[u'u'']} = \{y_1 / t_1, \ldots, y_k / t_k\}$ и $\xi'' = \dfrac{\mu}{\eta[v'v'']} = \{y_1 / s_1, \ldots, y_k / s_k\}$. Эти подстановки применяются к одному и тому же множеству вспомогательных переменных $\{y_1, \ldots, y_k\}$, которые вводятся в термах подстановки μ при выполнении редуцированной антиунификации (подробности см. в [15,16,20]). Будем далее говорить, что подстановки ξ' и ξ'' *определяют множество переменные* $y_1, \ldots, y_k$. Если подстановка μ приписывается вершине (v',v''), то между подстановками μ, ξ' и ξ'' установим отношение предшествования: подстановки ξ' и ξ'' объявляются предшественниками подстановки μ (для обозначения отношения предшествования будем использовать записи $\mu \to \xi'$ и $\mu \to \xi''$). В свою очередь, предшественниками подстановки ξ' объявляются все предшественники (если таковые есть) подстановки $\eta[u'u'']$, а предшественниками подстановки ξ'' объявляются все предшественники подстановки $\eta[v'v'']$. Обозначим транзитивное замыкание отношения предшествования записью $\xrightarrow{*}$. Общее число подстановок-пополнений, порождаемых по ходу работы процедуры вычисления стационарной разметки

графа $\Gamma[\pi',\pi'']$, ограничено величиной $O(N^4)$, и суммарный размер их оценивается величиной $O(N^6)$.

Лемма 6. Предположим, что на некотором шаге работы процедуры вычисления стационарной разметки графа логически совместных трасс $\Gamma[\pi',\pi'']$ вершине (v',v'') приписана подстановка $\eta[v'v'']$ и множество подстановок $S[v'v'']$. Тогда:

1. для любой подстановки λ из множества $S[v'v'']$ существует такая последовательность подстановок-пополнений, связанных отношением предшествования $\eta[v'v''] \to \xi_1 \to \xi_2 \to \cdots \to \xi_m$, для которой справедливо равенство $\lambda = \eta[v'v'']\xi_1\xi_2\ldots\xi_m$;
2. для любой последовательность подстановок-пополнений, связанных отношением предшествования $\eta[v'v''] \to \xi_1 \to \xi_2 \to \cdots \to \xi_m$, существует такая подстановка λ из множества $S[v'v'']$, для которой справедливо равенство $\lambda = \eta[v'v'']\xi_1\xi_2\ldots\xi_m$.

Доказательство. Проводится индукцией по числу шагов работы процедуры вычисления стационарной разметки графа $\Gamma[\pi',\pi'']$. При обосновании индуктивного перехода ограничимся рассмотрением первого утверждения леммы. Обоснование второго утверждения проводится сходным образом.

Предположим, что в вершине (v',v'') произошло изменение разметки: подстановка $\eta[v'v'']$ была заменена подстановкой $\mu = \theta\eta[u'u''] \Downarrow \eta[v'v'']$, а множество подстановок $S[v'v'']$ было заменено множеством подстановок $S[v'v''] \cup \{\theta\lambda' : \lambda' \in S[u'u'']\}$. В этом случае вводятся новые подстановки дополнения ξ' и ξ'', удовлетворяющие равенствам $\theta\eta[u'u''] = \mu\xi'$ и $\mu = \eta[v'v'']\xi''$. Тогда по индуктивному предположению для любой подстановки λ из множества $S[v'v'']$ верны равенства $\lambda = \eta[v'v'']\xi_1\xi_2\ldots\xi_m = \eta[v'v'']\xi''\xi_1\xi_2\ldots\xi_m$, а для любой подстановки λ из множества $\{\theta\lambda' : \lambda' \in S[u'u'']\}$ верны равенства $\lambda = \theta\lambda' = \theta\eta[u'u'']\xi_1\xi_2\ldots\xi_m = \mu\xi'\xi_1\xi_2\ldots\xi_m$, подтверждающие первое утверждение леммы для новой разметки вершины (v',v''). □

Таким образом, вся информация о подстановках из множеств, ассоциированных с вершинами графа $\Gamma[\pi',\pi'']$, содержится в подстановках-пополнениях, связанных отношением предшествования. Покажем теперь, как

можно быстро вычислять «на лету» наиболее общий унификатор $\tau = НОУ(\bigcup_{(u,v)\in\Gamma(\pi',\pi'')} \bigcup_{\lambda\in S[uv]} \{(A[u]\lambda, A[v]\lambda)\})$, используя введенное неявное описание подстановок.

После того, как построена стационарная разметка $\eta[uv]$, для вычисления подстановки τ можно воспользоваться следующей процедурой $UNIF$, которая перемежает выполнение алгоритмом унификации Мартелли-Монтанари (см. [4,15]) и применение подстановок-пополнений. Вначале этот алгоритм применяется для решения системы уравнений $\{A[u]\eta[uv] = A[v]\eta[uv] : (u,v) \in \Gamma[\pi', \pi'']\}$ и конструирует равносильную приведенную систему уравнений. Если такой системы получить не удается, то унификация исходных пар атомов невозможна. Если такая приведенная система уравнений E может быть построена, то рассматривается множество вспомогательных переменных $\{y_1, y_2, \ldots, y_m\}$ и среди подстановок-пополнений выбирается максимальная по отношению $\xrightarrow{*}$ пара подстановок ξ' и ξ'', определяющих переменную одну из переменных y_i рассматриваемого множества $\{y_1, y_2, \ldots, y_m\}$. После этого система уравнений E преобразуется в систему уравнений $E\xi' \cup E\xi''$, в которой переменная y_i уже будет отсутствовать. Далее процедура $UNIF$ вновь применяется к полученной системе алгоритм Мартелли-Монтанари. Поскольку отношение $\xrightarrow{*}$ является отношением строгого частичного порядка на множестве подстановок-пополнений, и каждая переменная определяется лишь в паре таких подстановок ξ' и ξ'', удаленная вспомогательная переменная y_i уже не появится в системах уравнений на последующих этапах выполнения процедуры $UNIF$. Выполнение описанного процесса унификации завершается, когда в очередной приведенной системе уравнений E не остается вспомогательных переменных, т.е. $E = \{x = t : x \in \hat{X}, \hat{X} \subseteq X' \cup X''\}$. В этом случае подстановка $\{x/t : x \in \hat{X}\}$ объявляется результатом его работы.

На основании леммы 6 и свойств корректности и полноты алгоритма унификации Мартелли-Монтанари справедлива.

Лемма 7. Описанный процесс последовательного применения процедуры $UNIF$ вычисляет подстановку $\{x/t : x \in \hat{X}\}$ тогда и только тогда, когда

наиболее общий унификатор $НОУ(\bigcup_{(u,v)\in\Gamma(\pi',\pi'')}\bigcup_{\lambda\in S[uv]}\{(A[u]\lambda, A[v]\lambda)\})$ существует. При этом $\tau = \{x/t : x \in \hat{X}\}$.

Нетрудно видеть, что процедура $UNIF$ $O(N^4)$ раз применяет алгоритм унификации Мартелли-Монтанари к системам, содержащим $O(N^4)$ уравнений, и размер термов в каждом из уравнений оценивается величиной $O(N^2)$. Таким образом, вычисление $НОУ(\bigcup_{(u,v)\in\Gamma(\pi',\pi'')}\bigcup_{\lambda\in S[uv]}\{(A[u]\lambda, A[v]\lambda)\})$ можно осуществить за время $O(N^{10})$.

Теорема 4. Алгоритм л-т унификации программ $\pi'(X)$ и $\pi''(X)$ завершает работу за время, полиномиальное относительно суммарного размера этих программ.

Доказательство. Пусть N – суммарный размер программ $\pi'(X)$ и $\pi''(X)$.Как следует из леммы 3, алгоритм л-т унификации программ $\pi'(X)$ и $\pi''(X)$ завершает работу за $O(N)$ этапов. На каждом этапе выполняется процедура вычисления стационарной разметки графа логически совместных трасс $\Gamma[\pi',\pi'']$, которая завершает работу за $O(N^4)$ шагов. На каждом шаге выполняется операция редуцированной антиунификации, которая применяется к подстановкам размера $O(N^2)$. По завершении построения стационарной разметки наиболее общий унификатор $НОУ(\bigcup_{(u,v)\in\Gamma(\pi',\pi'')}\bigcup_{\lambda\in S[uv]}\{(A[u]\lambda, A[v]\lambda)\})$ вычисляется за время $O(N^{10})$. Таким образом, общее время работы описанного в данной статье алгоритма л-т унификации оценивается величиной $O(N^{11})$. □

6. Заключение

Успешное решение задачи л-т унификации программ в простой постановке, когда в качестве интерфейсов-инициализаторов (π'_{in}, π''_{in}) используются лишь цепочки операторов присваивания, создает предпосылки для изучения более общего случая, когда в качестве интерфейсов (π'_{in}, π''_{in}) и $(\pi'_{out}, \pi''_{out})$ можно использовать произвольные ациклические программы. Другое направление исследований – это изучение возможности понижения сложности алгоритма л-т унификации программ.

Список литературы

[1] Robinson J.A. A machine-oriented logic based on the resolution principle // Journal of the ACM. 1965 — v. 12, N 1. — p. 23-41.

[2] Baxter L.D. An efficient unification algorithm // Technical Report CS-73-23, Dep. of Analysis and Comp. Sci., University of Waterloo, Ontario, Canada, 1973.

[3] Paterson M.S., Wegman M.N. Linear unification // The Journal of Computer and System Science. — 1978. — v. 16, N 2 — p. 158-167.

[4] Martelli A., Montanari U. An efficient unification algorithm //ACM Transactions on Program, Languages and Systems. — 1982. — v. 4, N 2 — p. 258-282.

[5] Stickel E.M. A unification algorithm for associative-commutative functions // Journal of the association for Computing Machinary. — 1981. — v. 28, N 5 — p. 423-434.

[6] Herold A., Sieckmann J. Unification in Abelean semigroups // Jornal of Automated Reasoning. — 1983. — p. 247-283.

[7] Lincoln P., Christian J. Adventures in associative-commutative unification //Journal of Symbolic Computation. — 1989. — v.8. — p. 393 — 416.

[8] Baader F., Snyder W. Unification theory // In J.A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning. — 2001. — v. 1 — p. 447-533.

[9] Goldfarb W.D. The undecidability of the second-order unification problem // Theoretical Computer Science. — 1981. — v. 13, N 2. — p. 225-230.

[10] Фаулер М. Рефакторинг. Улучшение существующего кода. — Символ-Плюс, 2008. — 432 с.

[11] Roy C. K., Cordy J. R. A survey on software clone detection research // Technical report TR 2007-541, School of Computing, Queen's University. — 2007. — v. 115.

[12] Komondoor R., Horwitz S. Using slicing to identify duplication in source code // Proceedings of the 8th International Symposium on Static Analysis. — Springer-Verlag, 2001. — p. 40–56.

[13] Иткин В.Э. Логико-термальная эквивалентность схем программ // Кибернетика. — 1972. — N 1. — с. 5-27.

[14] Сабельфельд В.К. Полиномиальная оценка сложности распознавания логико-термальной эквивалентности // ДАН СССР. — 1979. — т. 249, N 4. — с. 793-796.

[15] Захаров В.А., Новикова Т.А.. Применение алгебры подстановок для унификации программ. Труды Института системного программирования РАН, том 21, 2011 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print).

[16] Захаров В.А., Новикова Т.А.. Полиномиальный по времени алгоритм проверки логико-термальной эквивалентности программ. Труды Института системного программирования РАН, том 22, 2012 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print).

[17] Eder E. Properties of substitutions and unifications // Journal of Symbolic Computations. — v. 1. — 1985. — p. 31-46.

[18] Palamidessi C. Algebraic properties of idempotent substitutions // Lecture Notes in Computer Science — v. 443 — 1990. — p. 386-399.

[19] В.Е., Сабельфельд В.К. Теория схем программ. — М.:Наука, 1991. — 348 с.

[20] Захаров В.А., Костылев Е.В. О сложности задачи антиунификации // Дискретная математика. — 2008. — т. 20, N 1. — с. 131-144.

[21] Luckham D.C., Park D.M., Paterson M.S., On formalized computer programs // Journal of Computer and System Science — 1970. — v.4, N 3. — p. 220-249.

[22] Котов В.Е., Сабельфельд В.К. Теория схем программ. — М.:Наука, 1991. — 348 с.

On Program Unification

V.A.Zakharov (ISP RAS, Moscow Russia),
T.A. Novikova (Faculty of Computational Mathematics and Cybernetics,
Lomonosov Moscow State University, Moscow, Russia)
{zakh@cs.msu.su} {taniaelf@mail.ru}

Annotation. To unify a pair of algebraic expressions t_1 and t_2 is to find out such a substitution θ that both terms $t_1\theta$ and $t_2\theta$ have the same meaning. Unification problem can be extended to computational programs. To unify a pair of programs π_1 and π_2 is to build two sequences of assignment statements $\rho_1 : x_1 := t_1; x_2 := t_2; \ldots x_n := t_n$ and $\rho_2 : y_1 := s_1; x_2 := s_2; \ldots x_m := s_m$, such that both compositions of programs $\rho_1;\pi_1$ и $\rho_2;\pi_2$ are equivalent (i.e. they compute the same function). In this paper we deal with logic-and-term equivalence introduced in 1972 by V. Itkin. This is the most weak decidable equivalence on programs that approximates the functional equivalence. Based on the polynomial-time equivalence checking algorithm for logic-and-term equivalence in the first-order model of imperative programs we introduce a polynomial-time unification algorithm which computes the most general unifier (ρ_1, ρ_2) for every pair of sequential imperative programs (π_1, π_2) w.r.t. logic-and-term equivalence. After discussing the importance of unification proble for program refactoring and optimization we briefly recall the basic concepts of term substitution theory and theory of program schemata. Then we introduce a formal model of sequential programs we deal with, and finally we present our unification algorithm, prove its correctness and show that its complexity is polynomial on the size of programs to be unified

Keywords: program, logic-and-term equivalence, substitution, unification, complexity.

References

[1]. Robinson J.A. A machine-oriented logic based on the resolution principle. Journal of the ACM. 1965, v. 12, N 1, p. 23-41.
[2]. Baxter L.D. An efficient unification algorithm. Technical Report CS-73-23, Dep. of Analysis and Comp. Sci., University of Waterloo, Ontario, Canada, 1973.
[3]. Paterson M.S., Wegman M.N. Linear unification. The Journal of Computer and System Science. 1978, v. 16, N 2, p. 158-167.
[4]. Martelli A., Montanari U. An efficient unification algorithm. ACM Transactions on Program, Languages and Systems. 1982, v. 4, N 2, p. 258-282.
[5]. Stickel E.M. A unification algorithm for associative-commutative functions. Journal of the association for Computing Machinary. 1981, v. 28, N 5, p. 423-434.

[6]. Herold A., Sieckmann J. Unification in Abelean semigroups. Journal of Automated Reasoning. 1983, p. 247-283.
[7]. Lincoln P., Christian J. Adventures in associative-commutative unification. Journal of Symbolic Computation. 1989, v. 8, p. 393-416.
[8]. Baader F., Snyder W. Unification theory. In J.A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning. 2001, v. 1, p. 447-533.
[9]. Goldfarb W.D. The undecidability of the second-order unification problem. Theoretical Computer Science. 1981, v. 13, N 2, p. 225-230.
[10]. Fowler M. Refactoring. Improving the Design of Existing Code. Addison-Wesley, 1999.
[11]. Roy C. K., Cordy J. R. A survey on software clone detection research. Technical report TR 2007-541, School of Computing, Queen's University. 2007, v. 115.
[12]. Komondoor R., Horwitz S. Using slicing to identify duplication in source code. Proceedings of the 8th International Symposium on Static Analysis. Springer-Verlag, 2001, p. 40–56.
[13]. Itkin V.E.: Local- termal equivalence of program schemata. Proceedings of the International Sympoisum on Theoretical Programming 1972: p. 127-143.
[14]. Sabelfeld V.K.: The Logic-Termal Equivalence is Polynomial-Time Decidable. Information Processing Letters. 1980, v.10, N 2, p. 57-62.
[15]. Zakharov V.A., Novikova T.A. Primineniye algebry podstanovok dly unifikatcii program [On the application of substitution algebra to program unification]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, v. 21, p. 141-166.
[16]. Zakharov V.A., Novikova T.A.. Polynomialny po vremeni algoritm proverki logiko-termalnoy ekviviletnosti program [Polynomial time algorithm for checking strong equivalence of programs]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, v. 22, p. 435-455
[17]. Eder E. Properties of substitutions and unifications. Journal of Symbolic Computations. 1985. v. 1, p. 31-46.
[18]. Palamidessi C. Algebraic properties of idempotent substitutions. Lecture Notes in Computer Science. 1990, v. 443, p. 386-399.
[19]. Luckham D.C., Park D.M., Paterson M.S., On formalized computer programs // Journal of Computer and System Science. 1970, v.4, N 3, p. 220-249.