ИСП Институт Системного Программирования
Российской Академии наук

Труды Института Системного Программирования

Том 21

Москва 2011

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

Труды Института Системного Программирования

Том 21

Под редакцией
академика РАН В.П. Иванникова

Москва 2011

УДК51

Труды Института системного программирования: Том 21.
/Под ред. В.П. Иванникова/ – М.: ИСП РАН, 2011. – 370 с.

В двадцать первом томе Трудов Института системного программирования публикуются статьи, содержащие результаты работ, проводимых в Институте в 2011 году.

Содержание

Автоматическая генерация OpenCL-кода из гнезд циклов с помощью полиэдральной модели
Андрей Белеванцев, Алексей Кравец, Александр Монаков. 5

Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ
Арутюн Аветисян, Андрей Белеванцев, Алексей Бородин, Владимир Несов.....23

Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок
Арутюн Аветисян, Алексей Бородин 39

Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах, использующих сетевые сокеты
И.К. Исаев, Д.В. Сидоров, А.Ю. Герасимов, М.К. Ермаков.....55

Динамическое профилирование программы для системы LLVM
А.И. Аветисян, К. Ю. Долгорукова; Ш. Ф. Курмангалеев 71

Методы точного измерения времени выполнения гнезд циклов при анализе JavaMPI-программ в среде ParJava
А.И. Аветисян, М.С. Акопян, С.С. Гайсарян 83

Поддержка команд с условным выполнением в селективном планировщике команд
Дмитрий Мельник, Александр Монаков, Арутюн Аветисян 103

Метод автоматического восстановления переменных из трассы исполнения программы
М.А. Климушенкова, В.А. Макаров.....119

Детерминированное воспроизведение процесса выполнения программ в виртуальной машине
Павел Довгалюк.....123

Оценка производительности программного обеспечения в виртуализованном окружении на основе атомарных тестов
П.А. Клеменков.С. Камкин 133

Применение алгебры подстановок для унификации программ
В.А. Захаров, Т.А. Новикова 141

Риски проектирования и производства мобильных программных продуктов
В.В. Липаев 167

Решение проблемы NULL в запросах к реляционной базе данных на основе использования операторов реляционной алгебры А
И.В. Блудов 183

Обзор моделей данных объектно-ориентированных СУБД
А. Эльдарханов 205

Оценка производительности протокола Snapshot Isolation
Д.Н. Василик. 227

Экспериментальное исследование параллельного исполнения SQL запросов
К.К. Смирнов, Г.А. Чернышев 245

Интеграция алгоритма кластеризации Fuzzy c-Means в PostgreSQL
Р.М. Миниахметов 263

Обнаружение поискового спама в Вебе на основе анализа разнообразия текстов
А.С. Павлов, Б.В. Добров. 277

Извлечение объектов и их атрибутов из таблиц текстовых документов
Никита Астраханцев. 297

WikifyMe: создание модели сравнения для викификаторов
С.О. Бартунов, А.А. Болдаков, Д.Ю. Турдаков. 311

Извлечение предметно-ориентированных подмножеств словаря Википедии с использованием структуры категорий
А.В. Коршунов, Д.Ю. Турдаков, Чингук Чонг, Минхо Ли, Чансунг Мун. 323

Эвристическое моделирование данных в информационных системах
Мартин Давтян 349

Разработка параллельной СУБД на основе PostgreSQL
К.С. Пан. 357

Автоматическая генерация OpenCL-кода из гнезд циклов с помощью полиэдральной модели

Андрей Белеванцев <abel@ispras.ru>, Алексей Кравец <kayrick@ispras.ru>, Александр Монаков <amonakov@ispras.ru>

Аннотация. В данной работе предлагается способ автоматической генерации кода для стандарта OpenCL из гнезд циклов без зависимостей по данным между итерациями для программ на языках Си, Си++ и Фортран. Для генерации используется инфраструктура GRAPHITE компилятора GCC, использующая полиэдральную модель для анализа пространства итераций и пространства данных цикла. Описывается выполненная реализация и проведенные эксперименты, показывающие наилучшие результаты на вычислительных программах, основную часть которых составляют гнезда циклов.

Ключевые слова: оптимизации программы, полиэдральная модель, GCC, OpenCL**.**

1. Введение

Одной из проблем параллельных вычислений являлось отсутствие единого стандарта, охватывающего как многоядерные процессоры, так и графические ускорители. Для многопоточных приложений использовалось OpenMP или POSIX Threads, для графических ускорителей – nVidia CUDA[10-12].

Для решения данной проблемы была разработана программная модель OpenCL [1, 4, 6], которая позволяет программисту писать параллельный код для многоядерных процессоров и графических ускорителей (в дальнейшем будем называть такие устройства акселераторами). Несмотря на то, что данная модель становится все более популярной, в GCC на данный момент отсутствует возможность использования OpenCL в качестве целевого языка для генерации параллельного кода.

Целью данной работы является реализация преобразования гнезд циклов без зависимостей по данным в OpenCL код с использованием инфраструктуры GRAPHITE (GIMPLE Represented as Polyhedra with Interchangeable Envelopes) [7]. Преобразование гнезда циклов заключается в замене кода гнезда, выполняющегося на процессоре, на вызов функции-ядра OpenCL, выполняющей соответствующие вычисления на некотором акселераторе. Так же необходима вставка вызовов всех требуемых для работы OpenCL функций (создание контекста выполнения, создание и компиляция OpenCL-ядер, обмен

данными между центральным процессором и акселератором). Полученная в результате программа может быть выполнена на любом акселераторе, поддерживающем стандарт OpenCL, при наличии соответствующих библиотек времени выполнения.

2. Программная модель OpenCL

Программная модель OpenCL позволяет программисту описывать функции, которые будут параллельно выполнены на некотором акселераторе или наборе акселераторов, доступных на данной машине. Для задания конкретного акселератора служит понятие контекста, который задает класс требуемого устройства (многоядерный процессор, графический процессор) и понятие очереди команд, которая соответствует определенному акселератору.

В основе программной модели OpenCL лежит понятия ядра (kernel). Ядро — это функция, которая будет выполнена параллельно на акселераторе определенным количеством потоков Для того, чтобы отличать ядра-функции и ядра процессора в последующем изложении, понятие ядра всегда используется в смысле функции. Ядра определяются программистом в виде функций на некотором расширении языка Си. Создание ядра разделено на несколько этапов, каждому из которых отвечает вызов соответствующей функции OpenCL на центральном процессоре. В начале требуется преобразовать исходный код одного или нескольких ядер, заданный в виде массива строк, в OpenCL программу, далее полученная программа должна быть скомпилирована для нужного устройства. После компиляции OpenCL программы из нее могут быть получены ядра, отвечающие функциям в ее исходном коде.

Для запуска ядра на акселераторе необходимо указать его аргументы. Аргументами ядра могут быть как скалярные значения, так и адреса буферов в памяти акселератора. Так как в общем случае акселератор не имеет прямого доступа к памяти центрального процессора (например, когда акселератором является графический ускоритель), ядра работают с адресами в памяти акселератора. OpenCL предоставляет механизм выделения буферов памяти на акселераторе и обмена данными между памятью центрального процессора и акселератора.

Запуск нужного ядра с указанием требуемого количества потоков осуществляется с помощью вызова функции библиотеки OpenCL на центральном процессоре. Акселератор, на котором будет выполняться ядро задается очередью команд, которой передается запрос на выполнение ядра. Таким образом, программа состоит из последовательного кода, который выполняется на центральном процессоре (пользовательский код и вызовы библиотеки OpenCL), и параллельного кода, которые выполняется на акселераторе (ядра).

3. Преобразование гнезд циклов в ядра OpenCL

Получение гнезд циклов, над которыми будет производиться преобразование, выполняется в рамках инфраструктуры GRAPHITE компилятора GCC. Результат работы GRAPHITE может быть получен в виде набора SCoP-ов (Static Control Part), каждый из которых представлен в виде CLAST (структуры данных, хранящей SCoP в виде операторов циклов, условных операторов и базовых блоков). Так же в процессе работы GRAPHITE для всех циклов, входящих в каждый SCoP, производится анализ на наличие зависимостей между итерациями. При этом необходимо, чтобы для всех обращений к данным внутри SCoP индексная функция была бы аффинной, а все циклы, входящие в него имели бы аффинные границы и константные шаги. Полученные гнезда циклов могут быть восстановлены как в виде циклов на центральном процессоре, так и в виде OpenCL ядер[13]. Ниже приведено описание алгоритма восстановления GIMPLE из CLAST в процессе которого некоторые гнезда циклов будут заменены на запуск OpenCL ядер на акселераторе.

3.1. Определение целевых гнезд циклов

В первую очередь необходимо определить, какие гнезда циклов будут преобразованы в ядра OpenCL. Данные гнезда должны удовлетворять определенному набору требований, которые можно разделить на две основные группы – требования корректности работы и требования скорости работы. Требования корректности работы включают в себя требование отсутствия зависимостей по данным между ядрами, выполняющимися параллельно, и синхронизацию буферов в памяти акселератора и центрального процессора.

Для проверки наличия зависимостей между итерациями рассматриваются все обращения к данным в гнезде циклов, кроме обращения к тем данным, которые могут быть приватизированы в теле ядра. Наличие зависимостей между итерациями цикла означает, что цикл будет выполняться на центральном процессоре, но в его теле могут содержаться циклы без зависимостей, которые могут быть выполнены на акселераторе. Таким образом, для определения гнезда циклов, которое будет преобразовано в ядро, выполняемое на акселераторе, в первую очередь необходимо выделить идеальные гнезда циклов, между итерациями которых нет зависимостей. Такое гнездо циклов может быть замещено параллельным выполнением его тела на акселераторе без нарушения требований корректности.

Для определения требований к скорости работы воспользуемся формулой Амдаля. При этом необходимо учитывать, что накладные расходы на запуск ядра являются последовательным кодом. Рассмотрим гнездо циклов N, которое может быть преобразовано в ядро, выполняемое на акселераторе (то есть, весь код данного ядра является параллельным). Пусть время, затрачиваемое на выполнение данного цикла на центральном процессоре

равно $T_{base}\ N$, время, затрачиваемое на накладные расходы на запуск ядра, равно $T_{opencl}\ N$. Тогда, по закону Амдаля, ускорение, получаемое при выполнении данного гнезда циклов на акселераторе, не превышает

$$\frac{1}{\frac{T_{opencl}\ N}{T_{base}\ N + T_{opencl}\ N} + \frac{T_{base}\ N}{T_{base}\ N + T_{opencl}\ N * p}}$$

где p - число вычислительных элементов акселератора. Выигрыш от замены гнезда циклов *N* на запуск ядра будет в том случае, когда базовое время работы превышает время, полученное с помощью закона Амдаля.

$$T_{base} > T_{opencl} + \frac{T_{base}}{p} \quad (1)$$

При этом, необходимо заметить, что большую часть накладных расходов составляют расходы на копирование данных из памяти акселератора в память центрального процессора и наоборот. Таким образом, можно сказать, что $T_{opencl}\ N = O\ S_{data}\ N + T_{const}$, где $S_{data}\ N$ является пространством данных гнезда циклов *N*. Так как точная оценка T_{opencl} затруднена, в качестве $T_{opencl}\ N$ будем рассматривать $S_{data}\ N$. Аналогично, точная оценка $T_{base}\ N$ так же затруднена, однако, исходя из того, что *N* является гнездом циклов можно считать, что $T_{base}\ N = O\ S_{iter}\ N$, где $S_{iter}(N)$ является пространством итераций цикла *N*.

Таким образом, неравенство (1) преобразуется в следующее неравенство:

$$S_{iter}\ N > S_{data} + \frac{S_{iter}\ N}{p * C} \quad (2)$$

Где *C* — некоторая константа. Исходя из того, что функции обращения ко всем данным в *N* является аффинной, можно упростить неравенство (2), заменив мощность пространств на их размерности, что приводит к следующей эвристике:

Для выполнения преобразования гнезда циклов *N* в запуск ядра на устройстве требуется, чтобы выполнялось следующее неравенство:

$$dim\ S_{iter} > dim\ S_{data} \quad (3)$$

Таким образом, для выполнения преобразования необходимо, чтобы глубина гнезда циклов была больше, чем максимальная размерность данных,

копирование которых требуется для запуска данного ядра. Однако, данная эвристика не отбрасывает те гнезда циклов, для которых $T_{base} \approx T_{opencl}$, в случае, когда для их запуска не требуется обмен данными между акселератором и устройством. Рассмотрим гнездо циклов *N*. Если все гнездо циклов будет преобразовано в запуск ядра, то неравенство (1) приобретет следующий вид:

$$T_{base} > T_{const} + \frac{T_{base}}{p} \quad (4)$$

Где T_{const} — время, затрачиваемое на служебные операции OpenCL (передача аргументов ядра, запуск ядра и т.д.). Однако, если самый внешний цикл гнезда не может быть преобразован в ядро, для внутреннего гнезда циклов неравенство будет выглядеть так:

$$T_{base} > T_{const} * N_{iter1} + \frac{T_{base}}{p} \quad (5)$$

Где N_{iter1} — число итераций самого внешнего цикла. Таким образом, получаем, что следует избегать распараллеливания циклов, находящихся глубоко в рассматриваемом гнезде циклов.

Это дает следующую эвристику:

Для выполнения преобразования гнезда циклов N_{sub} находящегося на глубине *Depth* в гнезде циклов *N* необходимо, чтобы глубина гнезда циклов N_{sub} была больше, чем *Depth.*

3.2. Восстановление исходного кода ядра

После получения гнезда циклов, которое будет преобразовано в ядро, необходимо восстановить исходный код этого гнезда. Данная задача может быть разбита на три подзадачи – восстановление инструкций, отвечающих телу рассматриваемого гнезда циклов, восстановление объявления переменных и типов, вставка инструкций и переменных, отвечающих за OpenCL часть ядра.

Все восстанавливаемые инструкции можно разделить на два класса – инструкции, восстанавливаемые из структур CLAST и инструкции, восстанавливаемые из базовых блоков представления GIMPLE. К первым относятся операторы `for` и `if`, для которых генерируется соответствующий код на языке Си при обходе дерева CLAST, соответствующего данному гнезду циклов. Ко вторым – все остальные инструкции [2, 7], для которых код генерируется с помощью процедуры синтаксического анализатора языка Си компилятора GCC, модифицированной нами для обработки инструкций GIMPLE.

Все имена, к которым идет обращения в коде ядра, должны быть объявлены в ядре либо как локальные переменные, либо как параметры ядра, переданные

ему при запуске. К первым относятся переменные, объявленные внутри гнезда циклов в исходном коде; итераторы циклов; временные переменные, созданные при преобразовании исходного кода в GIMPLE; служебные переменные и типы, специфичные для OpenCL ядер (например, идентификатор ядра). Остальные переменные считаются параметрами ядра и описываются как параметры генерируемой функции.

```
for (scat_1 = 0; scat_1 <= 99; scat_1++)
  for (scat_3 = 0; scat_3 <= 99; scat_3++)
    some_code (scat_1, scat_3);
```

Рис. 1: Базовое гнездо циклов

```
__kernel void
opencl_auto_function_0 (int ocl_mod_0, int ocl_first_0,
int ocl_mod_1,
int ocl_first_1, int ocl_base_1, int ocl_base_0) {
      size_t opencl_global_id = get_global_id (0);
      int scat_1 = ((opencl_global_id / opencl_base_0)
               % opencl_mod_0) * 1 + opencl_first_0;
      int scat_3 = ((opencl_global_id / opencl_base_1)
               % opencl_mod_1) * 1 + opencl_first_1;
      some_host_code (scat_1, scat_3);
}
```

Рис. 2: Полученный код ядра

Наконец, вставка кода, отвечающего за OpenCL часть ядра, выполняется по двум причинам. Во-первых, при замене гнезда циклов на ядро необходимо в каждой нити, выполняющей данное ядро, вычислить соответствующие значения итераторов исходных циклов (см. пример кода на рисунке 1 и сгенерированный код на рисунке 2, где в ядро выделяется оператор `some_code`), используя уникальный номер потока, получаемый с помощью вызова `get_global_id` [6]. Во-вторых, необходимо обходить ограничения реализаций стандарта OpenCL (в нашем случае – реализации AMD), например, невозможность объявления многомерных массивов [4], что приводит к тому, что в коде ядра необходимо генерировать объявления многомерных массивов с помощью указателей (для примера на рисунке 3 функции, производящей умножение матриц, будет сгенерирован код на рисунке 4).

```
float A[100][100], B[100][100], C[100][100];
int foo5 () {
      int i,j,k;
      for (i = 0; i <= 99; i ++)
```

```
        for (j = 0; j <= 99; j ++) {
          float sum = 0.0;
          for (k = 0; k <= 99; k++)
            sum += A[i][k] * B[k][j];
          C[i][j] = sum;
      }
}
```

Рис. 3. Исходная функция с обращениями к многомерным массивам

```
__kernel void
opencl_auto_function_0 (..., __global float
*oclFTmpArg0,
__global float *oclFTmpArg1, __global float
*oclFTmpArg2) {
      typedef __global float oclFTmpType0[100];
      oclFTmpType0 *C = (oclFTmpType0*)oclFTmpArg0;
      typedef __global float oclFTmpType1[100];
      oclFTmpType1 *A = (oclFTmpType1*)oclFTmpArg1;
      typedef __global float oclFTmpType2[100];
      oclFTmpType2 *B = (oclFTmpType2*)oclFTmpArg2;
      ...;
}
```

Рис. 4. Полученные объявления типов для многомерных массивов

Общая структура кода, получаемая из CLAST, показана на рисунке 5.

3.3. Создание контекста и очереди команд, создание и запуск ядер

Перед вызовом любых функций OpenCL необходимо создать контекст и очередь команд [1]. При этом требуется, чтобы они были созданы один раз за все время работы программы. Значения контекста и очереди команд хранится в глобальных переменных, доступных всей программе. В начале каждой функции, в которой присутствуют вызовы функций OpenCL, проверяется значения переменной, хранящей значение контекста. Если значение равно нулевому указателю, то создается новый контекст и очередь команд, которые присваиваются соответствующим переменным (см. рисунок 6).

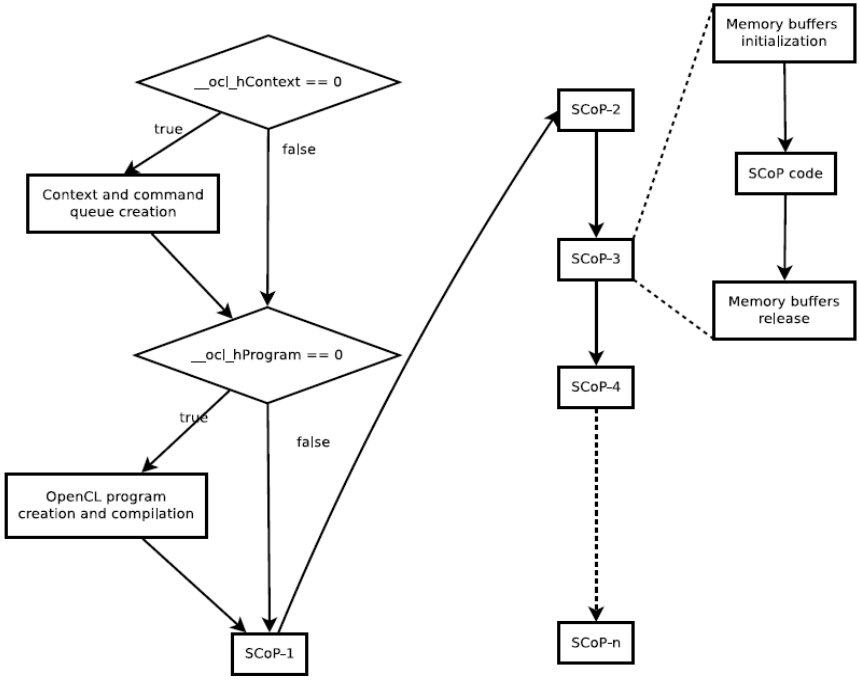


Рис. 5. Структура кода, генерируемого из структур CLAST

```
cl_context __ocl_hContext = 0;
if (__ocl_hContext == 0) {
      __ocl_hContext
      = clCreateContextFromType (0, CL_DEVICE_TYPE_GPU,
      0, 0, 0);
      clGetContextInfo (__ocl_hContext,
CL_CONTEXT_DEVICES, 0, 0,
              &nContextDescriptorSize);
      cl_device_id * aDevices = malloc
      (nContextDescriptorSize);
      clGetContextInfo (__ocl_hContext,
CL_CONTEXT_DEVICES,
              nContextDescriptorSize, aDevices, 0);
      __ocl_hCmdQueue = clCreateCommandQueue
      (__ocl_hContext,
                    aDevices[0], 0, 0);
}
```

Рис. 6. Инициализация контекста в начале функции

Далее, для каждой функции создается статическая переменная, хранящая OpenCL-программу, полученную из SCoP данной функции. Для этого в начало функции вставляется код, проверяющий значение данной переменной. Если значение равно нулевому указателю, то будет выполнено создание и компиляция OpenCL-программы. Нужные ядра создаются из полученной программы уже в нужном SCoP, так как основные затраты идут именно на компиляцию программы, а не на получение нужного ядра из скомпилированной программы.

При запуске ядра количество нитей, в которых оно будет работать, задается границами пространства итераций исходного гнезда циклов. Скалярные параметры передаются напрямую с учетом того, что OpenCL требует передачи указателя на параметр (например, `clSetKernelArg (kernel, index, sizeof (arg_type), &_ocl_scalar_arg)`). Для массивов или указателей передается указатель на соответствующий буфер в памяти акселератора. Этот указатель передается, так же как и скалярный параметр.

3.4. Работа с памятью

Передача данных из памяти центрального процессора в память акселератора и из памяти акселератора в память центрального процессора может занимать существенное время, что приводит к требованию минимизации количества таких передач в полученной программе.

В процессе анализа SCoP для каждого буфера на акселераторе поддерживается информация об актуальности данных, которые содержатся в нем и в соответствующей области памяти центрального процессора, соответствующей этому буферу. В начале работы для каждого массива или указателя, который используется на акселераторе в данной SCoP, в начале SCoP производится выделение буфера памяти на акселераторе. Так как требуемый размер буфера может стать известен только перед началом выполнения SCoP, то выделение буфера на акселераторе необходимо вставлять в начало каждого SCoP. Основной задачей в данном случае является определение базового объекта в памяти центрального процессора и размера нужного буфера на акселераторе. В качестве базового объекта для обращения к массиву или указателю выступает адрес массива или указатель соответственно. Для каждого обращения к соответствующему объекту в текущем SCoP вычисляется максимальное смещение относительно базового объекта в данном обращении, и размер буфера берется как максимальное смещение плюс единица. Все выделенные буферы инициализируются значениями из соответствующих областей памяти центрального процессора.

В процессе работы кода, полученного из SCoP, часть вычислений будет производиться на центральном процессоре и часть на акселераторе. Для корректности полученного кода необходимо, чтобы перед началом вычислений память содержала нужные данные. Алгоритм, определяющий

операции копирования памяти, необходимые для корректности, и места их вставки, описан в следующем подразделе. В конце работы для каждого SCoP происходит очистка памяти. Для всех буферов на акселераторе, данные в которых были изменены, но еще не скопированы в память центрального процессора, производится соответствующее копирование данных. После копирования памяти все буферы на акселераторе удаляются.

3.4.1. Копирование памяти в процессе работы программы

Для каждого объекта в памяти центрального процессора и соответствующего ему буфера на акселераторе на этапе генерации кода поддерживается информация о том, являются ли данные в данной памяти корректными. Корректность данных изменяется по следующему алгоритму:

1. Изначально данные корректны и в памяти центрального процессора и в памяти акселератора.
2. Если производится запись в буфер на акселераторе, то в соответствующей области памяти центрального процессора данные объявляются некорректными.
3. Если производится запись в область памяти центрального процессора, то в соответствующем буфере на акселераторе данные объявляются некорректными.
4. Если производится копирования из памяти центрального процессора в буфер на акселераторе, то данные в этом буфере объявляются корректными.
5. Если производиться копирования из буфера акселератора в область памяти центрального процессора, то данные в этой области памяти объявляются корректными.

Таким образом, получаем следующие правила корректности (их выполнение гарантирует, что не возникнет ситуации, когда и объект в памяти центрального процессора и соответствующий ему буфер на акселераторе содержат некорректные данные):

1. Перед запуском ядра на акселераторе необходимо, чтобы все буферы памяти, используемые этим ядром, содержали корректные данные.
2. Перед выполнением участка кода на центральном процессоре необходимо, чтобы все объекты в памяти, используемые в этом кода, содержали корректные данные.
3. Для копирования из памяти центрального процессора в буфер на акселераторе необходимо, чтобы данные в копируемой области были корректными.
4. Для копирования буфера акселератора в область памяти центрального процессора необходимо, чтобы данные в копируемом буфере были корректными.

Построим алгоритм, который определяет, какие операции копирования для выполнения правил корректности необходимо вставить в программу, и в каких местах должна быть выполнена вставка. Будем рассматривать код в виде дерева циклов, вершинами которого являются циклы, а листьями — последовательные блоки кода (в дальнейшем будем называть блоки кода и вложенные циклы операторами). В данном дереве прямыми потомками вершины `S`, соответствующей циклу `S´`, являются вершины, соответствующие операторам цикла `S´`. При этом порядок потомков `S` отвечает порядку операторов в теле `S´`. Под последовательным блоком кода в данном случае следует понимать участок кода на центральном процессоре или запуск ядра на акселераторе.

В процессе работы с деревом оно анализируется сверху вниз в глубину, причем для каждой вершины дерева ее дети рассматриваются в порядке, отвечающем порядку соответствующих операторов в теле цикла. Для каждого оператора `B` определим множества `last(B)`, `pred(B)` и `modify(B)`, вычисляемые по следующим правилам:

1. Если B является блоком, то last(B)={B}, modify(B) – множество всех объектов в памяти центрального процессора и буферов на акселераторе, модифицируемых в данном блоке.
2. Если B является циклом, то last(B)=last(B´), где B´ является последним блоком или циклом в теле B, modify(B)=∪ modify(Bj), где B_j – операторы, из которых состоит тело цикла B.
3. Если B_i не является первым оператором в теле цикла, то pred(Bi)=last(Bj), где Bj является прямым предшественником Bi.
4. Если же B_i является первым оператором в теле цикла, то pred(Bi)= pred(L) ∪ last(Bj), где L – цикл, в теле которого находится Bi, а B_j – последний оператор этого цикла.

Для каждого блока поддерживается информация о том, в каких буферах акселератора и областях памяти данные являются корректными по окончании работы блока. Перед началом выполнения рассматриваемого гнезда циклов все буферы корректны как на акселераторе, так и соответствующие им области памяти центрального процессора. В дальнейшем корректность изменяется по описанным выше правилам, причем буфер `M` является корректным перед началом работы блока `B` тогда и только тогда, когда он корректен по окончании работы всех блоков из `pred(B)`. Тогда алгоритм определения того, какие операции копирования необходимо вставить перед блоком для выполнения правил корректности, будет иметь следующий вид:

1. Для текущего блока B определяется множество pred(B).
2. Если блок будет выполняться на центральном процессоре, то:
    a. Для всех объектов в памяти, которые используются в данном блоке, проверяется, являются ли данные в них корректными

по окончании работы всех блоков из pred(B).

b. Если после какого-либо блока из pred(B) данные не являются корректными, то для данного объекта памяти требуется вставка копирования соответствующего буфера с акселератора в память центрального процессора.

c. Обновляется информация о корректности буферов памяти по окончании работы блока B.

3. Если блок выполняется на акселераторе, то:

   a. Для всех буферов акселератора, которые используются в данном блоке, проверяется, являются ли данные в них корректными по окончании работы всех блоков из pred(B).

   b. Если после какого-либо блока из pred(B) данные не являются корректными, то для данного объекта памяти требуется вставка копирования соответствующего объекта из памяти центрального процессора на акселератор.

   c. Обновляется информация о корректности буферов памяти по окончании работы блока B.

После определения блоков, перед которыми требуется вставить копирования памяти, осталось определить точное место в программе, куда будет вставлено каждое копирование. Для минимизации числа копирования памяти действует правило, что копирование вставляется в тело наиболее внешнего цикла (при условии сохранения корректности). Для определения цикла, в тело которого следует вставить копирование объекта `m` для блока `B`, ищется наиболее вложенный цикл `L`, содержащий в себе `B`, и такой, что $m \in modify(L)$. Тогда копирование объекта `m` должно быть вставлено в тело цикла `L` прямо перед блоком `B` или циклом, содержащим этот блок.

4. Результаты

Полученный компилятор тестировался на трех наборах тестов – PolyKernels [5], SPEC CPU2000 [3], Polyhedron 2005 Benchmark Suite [8]. Тесты запускались на четырехъядерном процессоре, то есть максимальным возможным ускорением является четырехкратное ускорение по сравнению с той же программой, выполняемой полностью на одном ядре.

В данном случае важно отметить, что, так как центральный процессор и акселератор являлись одним и тем же процессором, передача памяти в процессе работы не требовалась.

Тесты содержат программы на языках Си, Си++ и Фортран.

Тесты PolyKernels представляют собой программы, где основная часть вычислений сконцентрирована в одном гнезде циклов. Такие гнезда при отсутствии в них зависимостей могут быть эффективно преобразованы в

вызов OpenCL-ядра. Результаты запуска тестов на CPU представлены в таблице 1.

Название	Время работы		Ускорение
	GRAPHITE	GRAPHITE-OpenCL	
jac.c	5.87	5.92	x0.99
jac2d.c	86.24	86.28	x0.99
adi.c	0.13	0.13	x1.00
fdtd1d.c	0.93	0.97	x0.96
fdtd-2d.c	3.49	3.5	x0.99
gs.c	1.27	1.32	x0.96
gemver.c	0.28	0.3	x0.91
lud.c	1	0.99	x1.01
mmm.c	22.24	6.41	x3.46
mvt.c	1.36	1.82	x0.74
sor.c	4.43	4.43	x1.00
ssymm.c	14.55	4.22	x3.44
ssyr2k.c	17.86	17.83	x1.00
ssyrk.c	59.29	59.72	x0.99
strmm.c	59.89	59.69	x1.00
strsm.c	59.73	59.93	x0.99
tmm.c	19.06	6.99	x2.72
trisolv_if.c	0.07	0.07	x0.99
trisolv.c	25	26.02	x0.96

Таблица 1. Результаты PolyKernels на CPU

Дополнительно этот же набор тестов был запущен на nVidia GPU (GeForce GTX 260) (таблица 2). В данном случае, так как центральный процессор и акселератор не имеют общей памяти, передача данных между ними является необходимой. Как видно из результатов, для некоторых тестов передача данных уменьшила выигрыш от параллельного выполнения по сравнению с соответствующими результатами на CPU. При этом время выполнения некоторых тестов уменьшилось в несколько раз.

Название	Время работы		Ускорение
	GRAPHITE	GRAPHITE-OpenCL	
jac.c	9.35	9.34	x1.00
jac2d.c	84.58	84.73	x0.99
adi.c	0.09	0.08	x1.05
fdtd1d.c	1.54	1.52	x1.01
fdtd-2d.c	3.65	3.64	x1.00
gs.c	1.5	1.5	x0.99
gemver.c	1.36	1.03	x1.32
lud.c	1.33	1.34	x0.99
mmm.c	34.82	0.91	x37.90
mvt.c	3.51	1.48	x2.36
sor.c	1.06	1.06	x1.00
ssymm.c	31.34	14.94	x2.09
ssyr2k.c	15.62	15.87	x0.98
ssyrk.c	51.82	49.95	x1.03
strmm.c	54.14	54.91	x0.98
strsm.c	52.62	53.05	x0.99
tmm.c	35.25	11.51	x3.06
trisolv_if.c	0.1	0.1	x0.99
trisolv.c	42.95	42.21	x1.01

Таблица 2. Результаты PolyKernels на GPU

Результаты работы набора тестов SPEC CPU 2000 приведены в таблице 3. Тесты SPEC CPU представляют из себя программы среднего объема, в которых тривиально распараллеливаемые циклы достаточно простые и распределены по всему коду программы. Это приводит к некоторому увеличению накладных расходов по сравнению с предыдущим набором тестов, так как количество запускаемых ядер больше. При этом общая доля кода, выполняемого параллельно меньше, чем в случае PolyKernels.

Название	Время работы	Ускорение	
		С эвристиками	Без эвристик
wupwise	76.1	x0.99	x1.07
swim	125	x1	x1.00
mgrid	115	x0.99	x0.99
mesa	59.2	x1.02	x0.99

Название	Время работы	Ускорение	
galgel	69.2	x0.96	x0.91
art	40.2	x1.00	x1.00
equake	53.7	x1	X
facerec	96.5	x1	x0.99
ammp	118	x0.98	X
lucas	83.3	x1.01	x1.07
fma3d	145	x0.99	x0.91
sixtrack	113	x0.99	X
apsi	135	x1	x0.98
gzip	97.3	x1.00	X
vpr	74.7	x0.99	x0.99
gcc	52.5	x0.99	X
mcf	94.7	x1.01	x1.00
parser	130	x1	x0.47
eon	43.9	x0.99	x0.98
perlbmk	67.4	x0.99	X
gap	50.9	x1	x0.99
vortex	80.7	x0.99	x0.99
bzip2	74.6	x1.00	X
twolf	115	x1.01	x0.98

Таблица 3: Результаты SPEC CPU 2000 на CPU

Для тестов SPEC CPU 2000 для демонстрации работы эвристик выбора эффективных для распараллеливания гнезд циклов мы приводим ускорение работы с отключенными и включенными эвристиками соответственно. В случае отключенных эвристик возможен значительный рост времени выполнения программы. Например, время работы теста `parser` выросло более чем в два раза. В этом тесте было выделено 6 гнезд циклов, причем каждое гнездо состояло из одного цикла, и эти гнезда находились в трех различных функциях программы. Это привело к существенным временным затратам на компиляцию OpenCL-программ и выполнение служебных операций, сопутствующих каждому запуску OpenCL ядра. При этом время работы гнезд циклов, соответствующие полученным ядрам, составляло небольшую часть времени работы исходной программы, что привело к тому, что выигрыш по времени, полученный за счет параллельного выполнения этих циклов, незначителен по сравнению с накладными расходами. В случае же включенных эвристик почти все гнезда циклов отбрасываются на основании того, что время выполнения служебных операций, требуемых для запуска

ядра, будет больше, чем время, сэкономленное за счет параллельного выполнения гнезда цикла.

Наконец, результаты работы тестов Polyhedron 2005 представлены в таблице 4. Поведение этих тестов схоже со SPEC CPU 2000.

Название	Время работы		Ускорение
	GRAPHITE	GRAPHITE-OpenCL	
ac	10.91	10.96	x0.99
aermod	35.02	34.9	x1.00
air	0.24	0.23	x1.04
capacita	41.24	46.89	x0.87
channel	2.56	3.13	x0.81
doduc	43.71	43.09	x1.01
fatigue	9.45	9.49	x0.99
gas_dyn	7.93	7.81	x1.01
induct	48.31	48.62	x0.99
linpk	22.52	22.5	x1.00
mdbx	12.92	12.99	x0.99
nf	21.93	21.81	x1.00
protein	37.12	37.75	x0.98
rnflow	38.02	38.09	x0.99
test_fpu	10.81	12.1	x0.89
tfft	2.5	2.63	x0.95

Таблица 4: Результаты Polyhedron 2005 на CPU

Из результатов тестов видно, что накладные расходы при запуске большого числа небольших ядер могут занимать существенное время работы полученной программы. В некоторых случаях время, затрачиваемое на вызов служебных функций OpenCL, превышает выигрыш по времени, полученный за счет параллельного выполнения некоторых циклов программы. Таким образом, наилучший результат достигается на вычислительных программах, основную часть которых составляют гнезда циклов. В таких программах время, затрачиваемое на накладные расходы мало по сравнению с временем, затрачиваемым на сами вычисления.

5. Заключение

В данной работе была реализовано преобразование кода, заменяющее некоторые гнезда циклов на запуск ядер на акселераторе. Было реализовано

восстановление кода на языке Си из структур CLAST, преобразование этого кода в код OpenCL ядра, эвристики, определяющие оправданность замены гнезда циклов на запуск ядра, а так же вставка вызовов всех необходимых для запуска ядра функций. Тестирование показало, что данное преобразование достигает наилучшего результата на больших гнездах циклов, перенося основную часть вычислений на акселератор.

В дальнейшем планируется расширять множество конструкций языка, которые могут быть восстановлены в код ядра, а также уменьшить требования к виду циклов, которые могут быть преобразованы в запуск ядра.

Работа поддержана контрактом с Минобрнауки РФ № 07.514.11.4001

Список литературы

[1] ATI. OpenCL Programming Guide, March 2010. http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques, pp. 7-16, Juan-les-Pins, September 2004.

[3] Standard Performance Evaluation Corporation. Spec CPU 2000. http://www.spec,org/cpu

[4] Khronos OpenCL Working Group. The OpenCL 1.1 Specification, September 2010. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[5] IBM Polykernels. http://groups.google.com/group/gcc-graphite/

[6] NVIDIA OpenCL JumpStart Guide, April 2009. http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf

[7] Jan Sjödin, Sebastian Pop, Harsha Jagasia, Tobias Grosser, and Antoniu Pop. Design of graphite and the polyhedral compilation package. In Proceedings of GCC Summit 2009, Ottawa, Canada.

[8] Polyhedron Software. Polyhedron 2005 benchmark suite. http://www.polyhedron.com/polyhedron_benchmark_suite.html

[9] Альфред Ахо, Рави Сети, Джеффри Ульман. Компиляторы: Принципы, Технологии, Инструменты. Издательский дом Вильямс, 2008.

[10] NVIDIA. CUDA Programming Guide, 2010. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

[11] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. High Performance Embedded Architectures and Compilers, 2010, pp. 111-125.

[12] A. Monakov and A. Avetisyan. Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs. Embedded Computer Systems: Architectures, Modeling, and Simulation, 2009, pp. 289-297.

[13] A. Kravets, A. Monakov, and A. Belevantsev. GRAPHITE-OpenCL: Generate OpenCL Code from Parallel Loops. Proceedings of the GCC Developers' Summit, 2010, pp. 7-6.

Automatically generating OpenCL code from loop nests via a polyhedral model

Andrey Belevantsev <abel@ispras.ru>, Alexey Kravets <kayrick@ispras.ru>,
Alexander Monakov <amonakov@ispras.ru>

Abstract. In this work we suggest automatically generating code for OpenCL standard from loops with no dependencies in C/C++/Fortran programs. We use GCC compiler's GRAPHITE infrastructure that represents loop nests as polyhedra. We describe our implementation and experimental results which are the best for computational programs that spend most of their execution time in loop nests.

Keywords: program optimizations, polyhedral model, GCC, OpenCL.

Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ

Арутюн Аветисян <arut@ispras.ru>, Андрей Белеванцев <abel@ispras.ru>,
Алексей Бородин <alexey.borodin@ispras.ru>,
Владимир Несов <nesov@ispras.ru>

Аннотация. Статический анализ является популярным средством поиска в исходном или двоичном коде программ определенных шаблонов или ситуаций (ошибок стиля кодирования, нарушений проектных соглашений об использовании определенных библиотек или свойств языка программирования, критических ошибок, уязвимостей, закладок). В данной статье предлагается обзор инструмента статического анализа исходного кода программ на языках Си/Си++, разработанного в ИСП РАН для поиска критических ошибок и уязвимостей. Применение межпроцедурного анализа потока данных, не гарантирующего нахождение всех заданных ситуаций, позволяет проводить автоматический анализ с долей истинных предупреждений в 40-80%, что находится на уровне лучших коммерческих инструментов статического анализа.

Ключевые слова: статический анализ, анализ потока данных, интервальный анализ, межпроцедурный анализ, уязвимости.

1. Введение

Высокая сложность программ влечет необходимость механизмов защиты от атак, в том числе поиска уязвимостей и ошибок, через которые могут происходить такие атаки. Одним из средств обнаружения критических ошибок, не выявляемых обычным тестированием, является статический анализ программ, а именно поиск ситуаций в исходном коде программы, которые могут означать наличие уязвимости. Примером простых ситуаций подобного типа являются выдаваемые компилятором предупреждения, например, об использовании указателя одного типа для манипулирования объектом другого типа. В более сложных ситуациях требуется применение глубокого анализа, который не может быть выполнен компилятором из-за практических ограничений (например, по времени анализа). Обычно такой анализ реализуется как отдельный инструмент с широким привлечением компиляторных технологий.

Данная статья предлагает обзор инструмента статического анализа Svace, разработанного в Институте системного программирования РАН для анализа программ на языках Си/Си++ и подробно описанного в статьях [1-9]. Инструмент генерирует список предупреждений с указанием типа предупреждения и данных о том, как именно может произойти ситуация, описанная в предупреждении (например, указатель на объект, созданный в одной функции программы, может принять нулевое значение в другой функции и после использоваться в третьей).

Инструмент разработан с учетом следующих требований:

- анализ производится автоматически, не требуется специально подготавливать исходный код либо вмешиваться в ходе анализа;
- анализ является межпроцедурным, то есть учитывает влияние разных функций программы на ее поведение при поиске заданных ситуаций;
- размер анализируемых программ может достигать миллионов строк кода и сотен тысяч функций (например, ядро ОС Linux, браузер Mozilla Firefox);
- инфраструктура анализа расширяема, т.е. возможно разработать дополнительные алгоритмы для поиска новых видов потенциально опасных ситуаций в коде программы с использованием имеющихся алгоритмов и подготовленных ими данных;
- доступ к полному исходному коду программы и коду используемых программой библиотек не является необходимым – достаточно иметь спецификации лишь некоторых часто используемых стандартных библиотечных функций;
- значительная часть найденных предупреждений должна быть истинной (т.н. true positive), т.е. необходимо находить ситуации, действительно подпадающие под описание указанного типа предупреждения как ошибочные или опасные; это не означает, что такие ситуации обязательно будут уязвимостями или проявятся во время выполнения программы [15].

Для достижения описанных целей (достаточно хорошее качество достаточно глубокого автоматического анализа при сохранении масштабируемости) необходимостью является применение такого анализа, который не гарантирует нахождение всех ситуаций заданных типов в исходном коде (англоязычный термин – unsound analysis), то есть качество анализа повышается за счет пропуска некоторого числа ошибок. Все известные коммерческие инструменты для автоматического статического анализа, не требующего трудоемкой подготовки анализируемых программ или требований к ним (Prevent компании Coverity [12] и K9 компании Klocwork [13]), используют тот же подход. При этом доля истинных предупреждений этих коммерческих инструментов, так же, как и разработанного нами инструмента, составляет обычно от 30% до 80% (в зависимости от типа

обнаруживаемой ситуации в исходном коде; дополнительная информация по этим инструментам доступна в сравнении [16]).

Текущая версия разработанного нами инструмента поддерживает работу в ОС Linux и Windows для анализа программ, написанных для работы на ОС Linux и предназначенных для сборки компиляторами GCC или ARMCC. Пользователю доступен интерфейс командной строки и графический интерфейс, реализованный в среде Eclipse [14].

Далее в статье мы рассмотрим архитектуру и поведение разработанного инструмента, а также характеристики результатов, достигнутых при анализе программ с открытым исходным кодом. Мы будем называть найденную ошибочную ситуацию в исходном коде *предупреждением*, которое является *истинным* в случае действительного наличия описываемой данным типом предупреждения ситуации в коде и *ложным* в обратном случае. *Детектором* (checker) будем называть компонент инструмента анализа, ответственный за поиск предупреждений определенного типа (или схожих типов). Детекторы используют остальные подсистемы анализа (в т.ч. другие детекторы) и обычно малы по сравнению со всем инструментом.

2. Архитектура инструмента статического анализа

Инструмент анализа оперирует файлами с *внутренним представлением* (IR), сгенерированными компилятором из файлов исходного кода программы и содержащими описания всех используемых типов, глобальных и локальных переменных модуля компиляции, а также текст функций в кода для абстрактной регистровой машины. Нами используется компилятор LLVM-GCC на основе открытого компилятора GCC [10], генерирующий внутреннее представление для LLVM [11] – популярной системы для построения компиляторов. Представление LLVM (т.н. биткод) компактно, его можно сохранять на диске как в двоичном, так и в текстовом виде, добавлять к коду программы метаданные, компоновать и т.п. Такое представление хорошо подходит для использования в системах статического анализа.

Для построения файлов с LLVM-биткодом (биткод-файлы) ипользуется специальная утилита *перехвата сборки*, поставляемая вместе с инструментом анализа, которая перехватывает все запуски указанного ей компилятора в ходе исполнения произвольной системы сборки анализируемой программы, разбирает командную строку запуска компилятора, формирует команду для выполнения компилятора LLVM-GCC и использует ее для построения биткод файлов анализируемой программы. Утилита перехвата сборки поддерживает как операционные системы на базе ядра Linux, так и ОС Windows, при этом способы ее реализации различны для этих ОС.

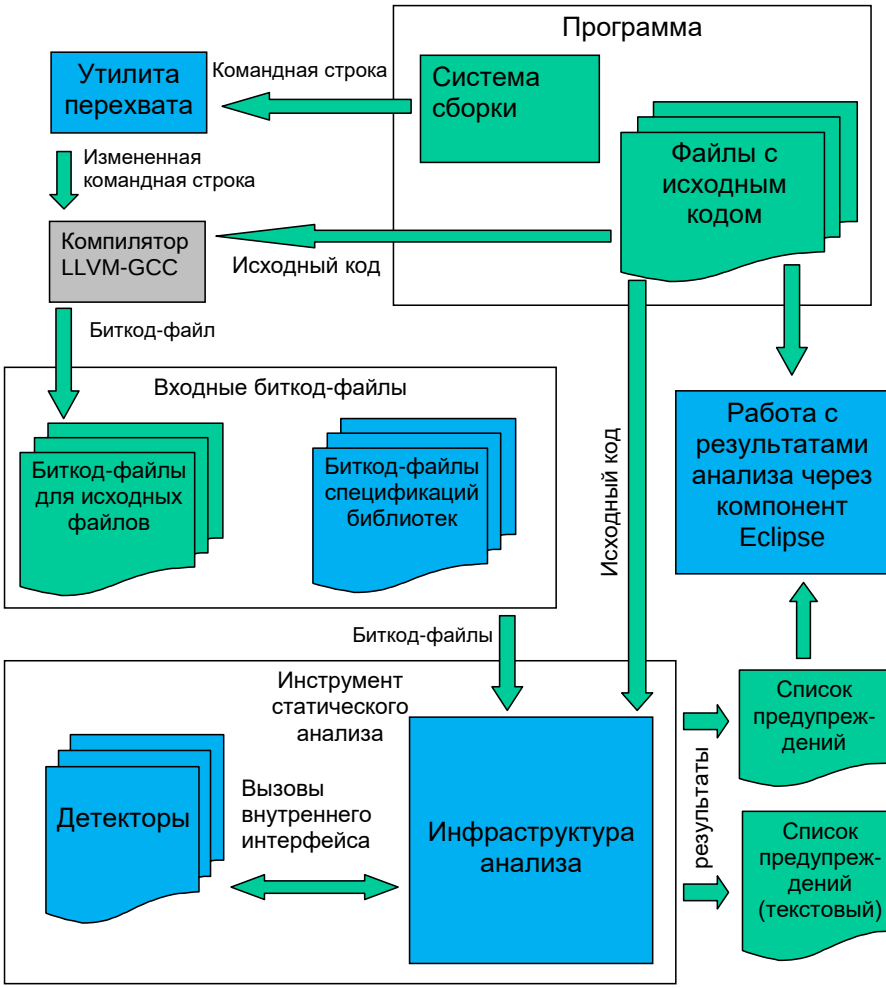


Рис. 1. Схема работы инструмента статического анализа.

Инструмент статического анализа состоит из инфраструктуры анализа, управляющей ходом анализа и реализующей набор общих алгоритмов, и набора детекторов, небольших компонент, осуществляющих поиск конкретных типов предупреждений и реализующих специальные алгоритмы анализа, необходимые для обнаружения ситуаций соответствующего вида. Помимо файлов внутреннего представления, при анализе используются

спецификации некоторых стандартных библиотечных функций, входящие в состав инструмента анализа, и (в некоторых случаях) файлы с исходным кодом анализируемой программы. Результаты анализа выдаются как в текстовом виде, так и во внутреннем формате, который может быть просмотрен в среде Eclipse с помощью поставляемого с инструментом анализа встраиваемого компонента – при этом текст предупреждения автоматически показывается в окружении соответствующего исходного кода.

Схема работы инструменты приведена на рис. 1, где компоненты инструмента выделены синим, а обрабатываемые данные – зеленым цветом. Компилятор LLVM-GCC выделен серым, так как он является сторонним компонентом. В начале работы утилита перехвата запуска компилятора запускает систему сборки программы обычным образом, параллельно создавая биткод-файлы. После окончания сборки над множеством сгенерированных биткод-файлов запускается инструмент анализа, который также подгружает спецификации библиотечных функций, необходимые для анализа и хранящиеся в виде биткод-файлов (далее работа со спецификациями ведется так же, как и с биткод-файлами анализируемой программы). Управление ходом анализа ведет инфраструктура анализа, вызывающая детекторы по мере надобности (детекторы регистрируют обработчики интересующих их событий, возникающих в ходе анализа программы). После окончания анализа сгенерированные предупреждения сохраняются в текстовом виде и внутреннем формате. Компонент Eclipse позволяет просматривать результаты с привязкой к исходному коду программы.

3. Ход анализа

3.1. Межпроцедурный анализ и аннотации

Распространяемые между функциями данные о программе хранятся в ходе анализа в виде аннотаций функций. *Аннотация* является структурой данных, связанной с функцией анализируемой программы, которая абстрактно описывает эффект от выполнения данной функции в произвольном контексте вызова – побочные эффекты, возвращаемые значения, способы использования и изменения параметров и других доступных функции данных. Аннотация создается автоматически как результат анализа функции. Детекторы могут сохранять в аннотации необходимые им данные для последующего анализа и выдачи предупреждений.

Необходимые инструменту анализа спецификации стандартных библиотечных функций (например, языка Си) пишутся также на Си в виде заглушек, в которых указаны только полезные для анализа операции. Исходный код спецификаций транслируется компилятором LLVM-GCC, и результат поставляется в виде биткод-файлов. Эти файлы анализируются до начала анализа остальных файлов, в результате чего создаются аннотации соответствующих библиотечных функций. Например, для функции `printf`

достаточно указать, что ее первый аргумент всегда разыменовывается (это свойство полезно для поиска ошибок разыменования нулевого указателя; для указания этого свойства первый параметр явно разыменовывается в исходном коде спецификации) и является форматной строкой (это свойство важно для поиска уязвимостей форматной строки), а также то, что данная функция является представителем семейства функций обработки форматной строки и не меняет свои аргументы. Для пометки о том, что некоторый указатель есть форматная строка, используется специальная интерфейсная функция `sf_use_format`, предоставляемая детектором уязвимостей форматной строки (аналогично для остальных функций). Использование этой функции перехватывается в ходе анализа. Сама аннотация на языке Си выглядит, как показано на рис. 2.

```
int printf(const char *format, ...) {
    char d1 = *format;
    sf_use_format(format);

    sf_fun_printf_like(0);
    sf_fun_does_not_update_vargs(1);
}
```

Рис. 2. Пример спецификации библиотечной функции.

Статический анализ программы происходит в четыре этапа. На первом этапе все биткод-файлы считываются по-очереди в произвольном порядке, и строится общий граф вызовов программы. На втором (основном) этапе граф вызовов обходится в обратном топологическом порядке («снизу вверх»), так что (насколько позволяет отсутствие циклов в графе вызовов) каждая функция посещается после того, как были посещены все вызываемые из нее функции. Для каждой посещенной в этом порядке обхода функции программы выполняется внутрипроцедурный анализ. На третьем этапе выполняется специфический для Си++ анализ кода, исследующий взаимодействие методов классов. На четвертом этапе принимаются решения о выдаче либо исключении некоторых предупреждений на основании собранных в ходе основного анализа статистических данных и фрагментов исходного кода, соответствующих местам потенциальной выдачи предупреждений.

В ходе анализа конкретной функции инструменту доступно внутреннее представление лишь этой функции и данные о вызываемых ей функциях, присутствующие в виде аннотаций. В результате анализа функции создается ее аннотация, которая может использоваться в дальнейшем. Размер аннотаций ограничен сверху (при превышении лимита из аннотации исключается часть данных о поведении функции, в порядке уменьшения приоритета), поэтому в

ходе анализа каждой функции обрабатывается ее внутреннее представление и ограниченный дополнительный объем информации, не зависящий от полного размера программы (количества составляющих всю программу функций). Таким образом, анализ всей программы масштабируется линейно. Все аннотации функций по возможности хранятся в оперативной памяти, но при ее нехватке сериализуются на диск и считываются при необходимости. Сериализация позволяет снять ограничение на размер анализируемого кода и выполнять анализ программ размером в несколько миллионов строк кода. Тем не менее, обычного современного объема оперативной памяти (1-4 ГБ) хватает для анализа программ из сотен тысяч строк кода без использования сериализации.

3.2. Внутрипроцедурный анализ

При анализе функции строится ее граф потока управления, после чего проводится потоково-чувствительный анализ, аналогичный анализу потока данных (т.е. для разных точек функции вычисляются разные значения атрибутов потока данных), при этом после нескольких проходов сверху вниз анализ завершается проходом снизу вверх. С каждой дугой графа потока управления ассоциируется *контекст* – информация о потоке данных, установленная для путей выполнения, проходящих через данную дугу. Например, после выполнения строки `if (x >= 4 && x <= 7)` в контексте подчиненного условному выражению ребра будет отражена информация о том, что значение переменной x находится в отрезке `[4,7]`. Анализ потока данных происходит путем «продвижения» контекста по дуге, входящей в инструкцию, через эту инструкцию и построения контекста на выходе из инструкции, основываясь на том, как инструкция манипулирует данными и памятью (при этом используются компактные структуры данных, не требующие чрезмерного дублирования данных). Для инструкции вызова продвижение контекста заключается в получении аннотации вызываемой функции (описывающей эффект от вызова вызываемой функции) и использовании ее для построения выходного контекста.

Контекст описывает взаимосвязь между элементами трех видов – абстрактными ячейками памяти, идентификаторами значений и атрибутами. Абстрактные ячейки памяти моделируют ячейки памяти, к которым происходит обращение в программе на различных путях исполнения. Идентификаторы значений обозначают значения, разделяемые различными ячейками памяти без изменения (схожей цели служат поколения переменных в представлении с единственным присваиванием, SSA). Наконец, атрибуты описывают свойства значения, отслеживаемого некоторым идентификатором значения. Например, после обработки следующих инструкций контекст выглядит как показано ниже (в упрощенном виде):

```
s = 6;
*p = s;
```

Ячейка памяти	p	*p	s
Идентификатор значения	vid1	vid2	
Атрибуты	PT-TO: *p NOT-NULL	POSITIVE	

Элементами этого контекста являются три ячейки памяти (p, s и *p, при этом p и s соответствуют явно указанным *переменным*), два идентификатора значений (один соответствует значению 6, другой соответствует адресу ячейки *p), и 3 атрибута. Ячейка p хранит значение с идентификатором vid1, адресом ячейки *p. Это отражено в атрибутах PT-TO (куда указывает p) и NOT-NULL (если бы указатель был нулевым, то присваивание *p=s привело бы к ошибке времени выполнения). Ячейки *p и s хранят одно и то же значение vid2 (этот факт устанавливается после обработки того же присваивания), которое имеет атрибут “положительный”. Кроме этого, например, детектор уязвимости переполнения буфера может отметить в атрибуте значения vid2, что оно равно в точности 6.

3.3. Контекстная чувствительность и аннотации

При анализе функции не используется информация о том, откуда она могла быть вызвана. Такой вид межпроцедурного анализа называется *контекстно-нечувствительным*, так как разные точки вызова функции не различаются, и возможно появление информации о данных на “ложных” путях выполнения, которые входят в функцию в одной точке вызова и выходят в другой (анализ не может отбросить эти данные как невозможные).

Тем не менее, некоторая контекстная чувствительность оказывается возможной при использовании параметризованных аннотаций [17].

На рис. 3 аннотация функции foo отражает тот факт, что ее возвращаемое значение равно аргументу, через использование одного идентификатора значения для обеих этих ячеек памяти, при этом идентификатор значения не привязан к конкретным атрибутам.

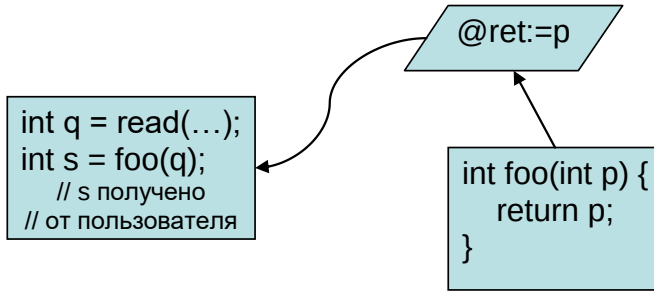


Рис 3. Параметризованная аннотация.

При продвижении контекста через вызов функции `foo` в примере используется фактический параметр `q`, значение которого получено от ввода пользователя (“испорчено”, т.е. не может быть использовано в потенциально опасных операциях без проверки на безопасность ввода). В ходе продвижения идентификатор значения для ячейки `s` устанавливается в идентификатор значения для ячейки `q`, тем самым значение `s` будет также помечено атрибутом “получено от пользователя”. При этом анализ вызова функции `foo` не влияет на ее аннотацию, и таким образом, информация об испорченности возвращаемого значения в данном вызове не будет влиять на анализ других вызовов той же функции. Такой анализ одновременно контекстно-чувствителен и позволяет распространять атрибуты через вызовы функций. Более подробно данная процедура описана в статье [9].

Процесс продвижения контекста через точку вызова функции называется *трансляцией аннотаций*, т.к. при этом элементы аннотации (по своей структуре аннотация очень схожа с контекстом) ставятся в соответствие элементам контекста в точке вызова – побочные эффекты вызываемой функции отражаются в контексте точки вызова, и новая информация, полученная об уже существующих в контексте вызова идентификаторах значений (указывающая, как фактические параметры вызова и доступные через них значения изменялись или использовались в вызываемой функции), должна быть объединена с имеющейся. В ходе трансляции возможно создание новых элементов контекста точки вызова.

На примере, показанном на рис. 4, аннотация функции `foo` содержит ячейки `z` и `*z`, у ячейки `z` есть идентификатор значения `v1`, имеющий атрибут, говорящий о том, что ячейка указывает на `*z`, а у ячейки `*z` есть идентификатор `v2`, имеющий атрибут, показывающий, что значение идентификатора равно четырем. В общем случае ячейки, идентификаторы и атрибуты составляют лес, в котором происходит поиск соответствующих элементов контекста в точке вызова и их создание в случае неудачи поиска. В нашем примере, контекст на ребре, входящем в инструкцию вызова, содержит

лишь ячейку x, которая связывается с z. Остальные элементы должны быть созданы в контексте исходящего из инструкции вызова ребра графа потока управления, где создается ячейка памяти для *x и соответственно устанавливается атрибут для идентификатора значения ячейки x (описывающий тот факт, что x указывает на *x), а также создается идентификатор значения для *x. После того, как все идентификаторы созданы, атрибуты идентификаторов *x и *z должны быть объединены (точнее, атрибуты идентификатора v2 должны быть перенесены с объединением на атрибуты идентификатора значения соответствующей ячейки *x). Так как ранее ячейки *x не было в контексте, то информация из аннотации просто копируется в виде атрибута, показывающего, что значение *x равно четырем. Если бы, например, идентификатор x в точке вызова содержал атрибут “получен от пользователя”, атрибут идентификатора z об указывании z на *z был бы объединен с имеющимся.

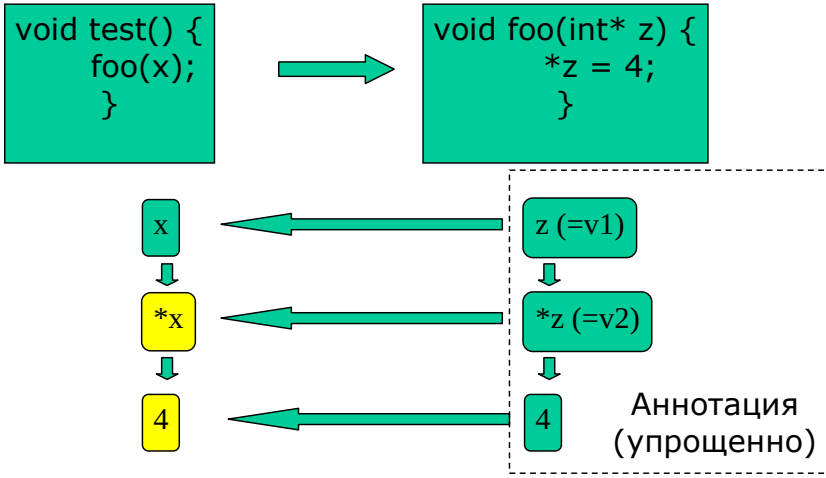


Рис 4. Трансляция аннотаций.

3.4. Использование атрибутов детекторами

Каждый детектор может вводить новые атрибуты для отслеживания необходимых свойств данных программы и выдачи предупреждений. Чтобы определить способы продвижения и слияния новых атрибутов, детекторы описывают обработчики этих событий, которые вызываются инфраструктурой анализа, и специальные функции манипуляции атрибутами, которые используются при написании спецификаций библиотечных функций (как на рис. 2). Обработчики событий имеют доступ к инструкции внутреннего

представления, через которую продвигается контекст, и всем элементам контекста, на которые ссылаются используемые в инструкции переменные и константы.

```
char buf[10];
buf[i]=0; // i∉[0,9] ⇒OVERFLOW
if(i>9) {
        // i ∈[10,+INF); i∉[0,9] ⇒OVERFLOW
        // ⇒ OVERFLOW
        exit(1);
}
```

Рис 5. Продвижение атрибутов.

На рис. 5 описаны шаги анализа, используемые для нахождения ситуации переполнения буфера. В ходе анализа приведенного фрагмента кода вводятся два атрибута. Первый атрибут описывает отрезок, внутри которого должно лежать значение, чтобы в данной точке программы не было переполнения буфера. Второй атрибут описывает отрезок, в котором гарантированно находится данное значение (т.н. *отрезок значения*). Во второй строке рассматриваемого примера первый атрибут устанавливается в отрезок `[0,9]`. Атрибут, связанный с идентификатором значения, распространяется до вызова функции `exit`. В точке вызова этой функции отрезок значения для этого идентификатора будет установлен в `[10,+INF)` как результат продвижения через условное выражение. Так как второй атрибут не содержит значений, находящихся в первом, детектор выдает предупреждение о переполнении буфера (на путях исполнения, проходящих через вызов функции `exit`, неизбежно происходит переполнение буфера).

В редких случаях детекторы могут обращаться к файлам с исходным кодом программы для отсечения ложных предупреждений с помощью информации, которая не сохранена в файлах с внутренним представлением. Например, некоторые предупреждения могут выдаваться в тексте макросов в местах, которые являются мертвым кодом. Знание о том, сгенерирован ли некоторый код с помощью макроса или написан программистом вручную, позволяет отсеять такие случаи.

3.5. Управление результатами анализа

После окончания анализа его результаты могут быть просмотрены пользователем через компонент в среде Eclipse. Для каждого предупреждения показывается сообщение и релевантные места в исходном коде программы (таких мест может быть несколько, например, может быть указана полная межпроцедурная трасса по функции программы, приводящая к ошибке). Для

удобства пользователя реализованы следующие дополнительные возможности:

- возможность запуска анализа некоторого проекта через среду Eclipse (а не только просмотр результатов);
- возможность размечать результаты анализа как истинные или ложные;
- поддержка истории запусков анализа. Эта часть инструмента позволяет сохранять результаты анализа и соответствующие исходные коды в базу данных и просматривать по мере необходимости. Кроме того, история запусков в совокупности с разметкой истинных и ложных предупреждений позволяет не выдавать пользователю повторно предупреждения, которые им были помечены как ложные (в т.ч. в случае, когда исходный код программы в ходе разработки был изменен, и предупреждение было выдано на другой строке).

Пример просмотра результатов анализа в Eclipse приведен на рис. 6.

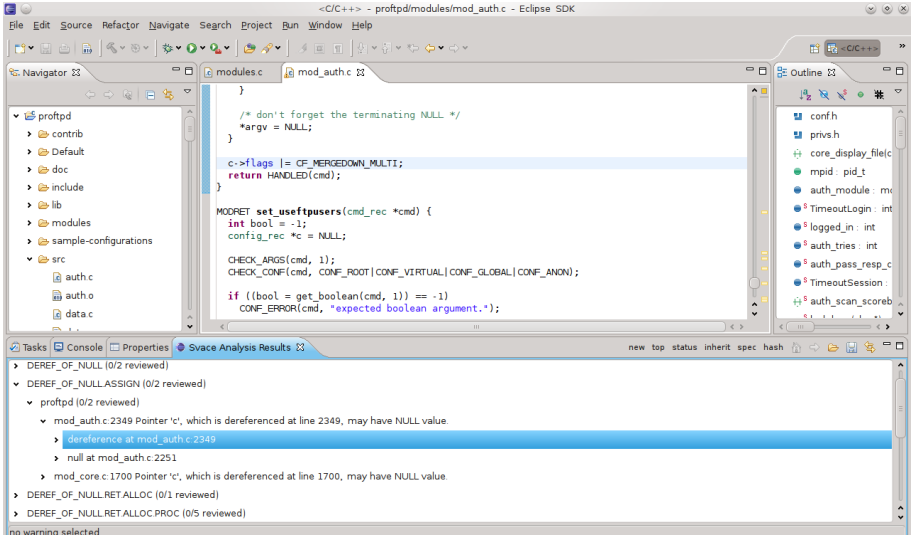


Рис.6. Просмотр результатов анализа в среде Eclipse.

4. Тестирование компонентов инструмента

Детекторы тестируются с помощью вручную написанных на языке Си тестов. Тестирование заключается в запуске анализа на данном файле или наборе файлов. Результат запуска тестов определяется функциями-утверждениями, вызываемыми в ходе теста, обычно о том, что предупреждение определенного

типа было выдано (или не выдано) на определенной строке файла. Все тесты могут быть запущены автоматически.

Например, на рис. 7 приведен тест для предупреждения `DEREF_OF_NULL` (разыменование указателя, имеющего нулевое значение). Этот тест проверяет работу межпроцедурного анализа с побочными эффектами, выражающимися в изменении значений глобальных переменных. Функция `get_var` всегда возвращает значение глобальной переменной `var`. В функции `test` первый условный оператор выполняется, когда функция `get_var` возвращает (а переменная `var` имеет) нулевое значение, и предупреждение `DEREF_OF_NULL` должно быть выдано при разыменовании этого значения. Требование о выдаче выражается вызовом функции-утверждения `sf_assert_thrown`, которой передается имя предупреждения. Выдача ожидается на следующей после вызова функции строке. Аналогично, второй условный оператор выполняется при ненулевом возвращаемом значении, и предупреждение не должно быть выдано, что выражается вызовом функции `sf_assert_not_thrown`.

```
#include "specfunc.h"

int* var;

int* get_var(void) {
    return var;
}

void test(int *p, int x, int y)
{
    if(!get_var()) {
        sf_assert_thrown(DEREF_OF_NULL);
        x = *get_var();
    }
    if(get_var()) {
        sf_assert_not_thrown(DEREF_OF_NULL);
        x = *get_var();
    }
}
```

Рис.7. Пример теста.

5. Тестирование качества анализа

Помимо небольших, написанных вручную тестов, для проверки качества общей инфраструктуры анализа и отдельных детекторов, а также для поиска

часто встречающихся ситуаций и разработки новых эвристик используются наборы пакетов программ с открытым исходным кодом. Рассматриваемые пакеты были автоматически собраны для поддерживаемых целевых платформ и содержат в общей сложности около 6 миллионов строк кода на Си и Си++.

Нужно заметить, что большая часть работы над инструментом заключается как раз в обработке результатов анализа больших пакетов, поиске ошибок и идентификации возможных улучшений алгоритмов их поиска. Без этой работы ценность основной инфраструктуры анализа невысока, так как большинство деталей анализа, достигающих конечного пользователя, раскрываются отдельными детекторами при выдаче предупреждений, а они в свою очередь тем точнее и надежнее, чем больше примеров исходного кода было проработано при их создании.

В ходе оценки качества анализа необходимо выработать критерии выбора предупреждений для анализа. На большом объеме исходного кода, как правило, обнаруживается большое количество предупреждений, и просмотреть все выданные предупреждения и для каждого из них вынести суждение об его истинности или ложности является чрезмерно трудоемкой задачей (на полном наборе тестовых пакетов программ может быть выдано до нескольких тысяч предупреждений одного типа). Мы используем псевдослучайную выборку фиксированного количества предупреждений каждого типа (или группы схожих типов), стабильную между различными запусками анализа. Таким образом, количество работы, необходимой для оценки результатов, ограничено, независимо от количества проанализированного исходного кода. Случайность выборки делает результаты оценки репрезентативными для проанализированных пакетов программ, а большой объем анализируемого кода делает результаты оценки репрезентативными среди возможных (не входящих в тестовый набор) пакетов программ.

При анализе каждое предупреждение обычно классифицируется как истинное либо ложное, но дополнительно возникает два случая: а) предупреждение истинное, так как инструмент нашел ровно ту ситуацию в исходном коде, которая была задана при поиске, но из кода видно, что такая ситуация не представляет ошибки и допущена намеренно; б) неясно, является ли предупреждение истинным или ложным, либо из-за нечеткости формулировки предупреждения, либо из-за сложности анализируемого исходного кода.

Полные результаты оценки разработанного инструмента статического анализа не могут быть раскрыты как полученные в ходе работы по коммерческим контрактам. В целом можно сказать, что из 15 групп оцениваемых предупреждений средняя доля истинных предупреждений составила 67%, от 25% до 100%. Более высокая доля истинных предупреждений (50-85%, в зависимости от типа) наблюдается у групп предупреждений о возможном разыменовании нулевого указателя и небезопасном использовании испорченных (контролируемых внешним вводом) данных. Более низкая доля

(40-60%, в зависимости от типа) у предупреждений об утечках памяти, использовании неинициализированных данных или освобожденной памяти, и переполнении буфера.

6. Заключение

Инструменты автоматического статического анализа являются полезными для задачи обеспечения безопасности программного обеспечения – с помощью глубокого межпроцедурного анализа можно достичь хорошей доли истинных предупреждений (40-80%) при приемлемом времени анализа (несколько часов для миллионов строк кода). Институтом системного программирования РАН разработан и успешно опробован в коммерческих проектах такой инструмент, качество которого не уступает лучшим коммерческим продуктам в данной области. Залогом успеха является кропотливая работа по просмотру реальных промышленных программ с открытым исходным кодом и доработке инструмента анализа на основе его результатов на этих программах, для достижения приемлемого компромисса между качеством выдаваемых результатов и их количеством. В планах работы над инструментом – поддержка других языков программирования (в первую очередь Java), языков Web-программирования (Javascript), развитие интерфейса пользователя (поддержка удаленного и распределенного анализа, поддержка командной работы), а также предоставление доступа к инструменту как к сервису в рамках систем облачных вычислений.

Список литературы

[1] С.С. Гайсарян, А.В. Чернов, А.А. Белеванцев, О.Р. Маликов, Д.М. Мельник, А.В. Меньшикова. О некоторых задачах анализа и трансформации программ. Труды ИСП РАН, №5, с.7-41, 2004.

[2] О.Р. Маликов, А.А. Белеванцев. Автоматическое обнаружение уязвимостей в программах. Материалы конференции «Технологии Майкрософт в теории и практике программирования», Москва, 2004.

[3] О.Р. Маликов. Автоматическое обнаружение уязвимостей в исходном коде программ. Известия ТРТУ, №4, с. 48-53, 2005.

[4] В.С. Несов, О.Р. Маликов. Использование информации о линейных зависимостях для обнаружения уязвимостей в исходном коде программ. Труды ИСП РАН, №9, с. 51-57, 2006.

[5] О.Р. Маликов, В.С. Несов. Автоматический поиск уязвимостей в больших программах. Известия ТРТУ, Тематический выпуск «Информационная безопасность», №7 (62), с. 114-120, 2006.

[6] В.С. Несов. Использование побочных эффектов функций для ускорения автоматического поиска уязвимостей в программах. Известия ЮФУ. Технические науки. Тематический выпуск «Информационная безопасность». Таганрог: Изд-во ТТИ ЮФУ, 2007. № 1(76), с. 134-139.

[7] В.С. Несов, С.С. Гайсарян. Автоматическое обнаружение дефектов в исходном коде программ. Методы и технические средства обеспечения безопасности

информации: Материалы XVII Общероссийской научно-технической конференции. СПб.: Изд-во Политехн. ун-та, 2008, с.107.
[8] В.С. Несов. Автоматическое обнаружение дефектов при помощи межпроцедурного статического анализа исходного кода. Материалы XI Международной конференции «РусКрипто'2009».
[9] Vladimir Nesov. Automatically Finding Bugs in Open Source Programs. Electronic Communications of the EASST 20. ISSN 1863-2122, 2009.
[10] Компилятор GCC. http://gcc.gnu.org
[11] Компиляторная инфраструктура LLVM. http://llvm.org
[12] Инструмент Coverity Prevent. http://www.coverity.com/library/pdf/coverity_prevent.pdf
[13] Инструмент статического анализа компании Klocwork. http://www.klocwork.com/products/insight/klocwork-truepath
[14] Среда Eclipse. http://www.eclipse.org/
[15] P. Godefroid. The Soundness of Bugs is What Matters (position statement). In BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools), 2005.
[16] P. Emanuelsson & U. Nilsson. A Comparative Study of Industrial Static Analysis Tools (extended version). Tech. rep., Linköping University, 2008.
[17] D. Liang & M. J. Harrold. Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses. In SAS '01: Proceedings of the 8th International Symposium on Static Analysis, pp. 279-298, London, UK, Springer-Verlag, 2001.

Using static analysis for finding security vulnerabilities and critical errors in source code

Arutyun Avetisyan <arut@ispras.ru>
Andrey Belevantsev <abel@ispras.ru>
Alexey Borodin <alexey.borodin@ispras.ru>
Vladimir Nesov <nesov@ispras.ru>

Abstract. Static analysis is a popular way of finding given patterns in source or binary code (e.g., coding style errors, violations of project guidelines of using specific libraries or language features, critical errors, security vulnerabilities, malicious code). In this paper we review the static analysis tool developed in ISP RAS for finding critical errors and security vulnerabilities in C/C++ source code. The tool uses interprocedural unsound dataflow analysis and allows performing fully automatic analysis resulting in 40-80% true positive rate which is on par with the best commercial tools in this area.

Keywords: static analysis, dataflow analysis, range analysis, interprocedural analysis, security vulnerability.

Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок

Арутюн Аветисян <arut@ispras.ru>, Алексей Бородин <alexey.borodin@ispras.ru>

Аннотация. В ИСП РАН разрабатывается инструмент статического анализа Svace для поиска ошибок в исходном коде программ на языках Си и Си++. Цель Svace - найти как можно больше ошибок при низком количестве ложных срабатываний и разумном использовании имеющихся ресурсов. Важными требованиями, предъявляемыми к системам статического анализа являются масштабируемость и расширяемость. В статье описывается встроенный механизм, поддерживающий включение в систему Svace детекторов новых видов ошибок, сохраняющий ее масштабируемость. Использование механизма иллюстрируется на примере четырёх разработанных детекторов ошибок.

Ключевые слова: статический анализ; анализ потока данных; расширяемость; ошибки разыменования нулевого указателя; ошибки работы с динамической памятью.

1. Введение

В ИСП РАН разрабатывается инструмент статического анализа Svace для поиска ошибок в исходном коде программ. Текущая версия инструмента позволяет анализировать программы, написанные на языках Си и Си++. Цель Svace - найти как можно больше ошибок при низком количестве ложных срабатываний и разумном использовании имеющихся ресурсов.

Важными требованиями, предъявляемыми к Svace являются масштабируемость и расширяемость. Масштабируемость необходима для возможности анализа больших по размеру программ с пропорциональным увеличением потребляемых ресурсов по сравнению с небольшими программами. Расширяемость требуется для возможности обработки новых видов ошибок. Расширяемость достигается с помощью системы расширяемых модулей и спецификаций. Основное внимание данной статьи будет посвящено проблеме расширяемости.

2. Краткое описание Svace

Входная программа на языке Си (или Си++) преобразуется с помощью компилятора gcc в набор модулей с промежуточным представлением на языке LLVM, этот набор подаётся на вход анализатору Svace.

На первом этапе Svace просматривает все файлы и строит граф вызовов функций.

На втором этапе этот граф обходится в обратном топологическом порядке, таким образом, что каждая функция посещается после того, как были посещены вызываемые из неё функции. При этом рекурсивные вызовы игнорируются, что может вносить некоторые неточности в анализ. На практике существенных проблем, связанных с рекурсивными функциями, обнаружено не было.

Для каждой посещаемой функции производится её анализ. Результатом анализа функции будет выдача найденных предупреждений, а также создание аннотации функции. Аннотация описывает эффект от вызова данной функции в произвольном месте программы. После этого функция уже не будет переанализироваться, а при её вызове для вычисления побочных эффектов используется аннотация.

Более подробно узнать о Svace можно в статьях [1-9].

3. Внутри процедурный анализ

При анализе функции строится ее граф потока управления, после чего проводится потоково-чувствительный анализ, аналогичный анализу потока данных. С каждой дугой графа потока управления ассоциируется контекст – информация о потоке данных, установленная для путей выполнения, проходящих через данную дугу. Большинство инструкций имеют один входной контекст, который затем преобразуется в выходной контекст в соответствии с семантикой инструкции. Инструкции слияния потока управления принимают несколько входных контекстов и создают на их основе один выходной. А инструкции ветвления могут иметь несколько выходных контекстов.

Контекст описывает взаимосвязь между следующими элементами: абстрактными ячейками памяти, идентификаторами значений и атрибутами. Абстрактные ячейки памяти моделируют ячейки памяти, к которым происходит обращение в программе на различных путях исполнения.

Идентификаторы значений обозначают группу возможных значений, которые может иметь ячейка памяти. Несколько ячеек памяти могут разделять одни и теже идентификаторы значений. Например, подобное происходит после

обработки инструкции присваивания – обе ячейки памяти, участвующие в инструкции присваивания будут иметь одно и тоже значение после её выполнения.

```
//a → val1, b →val2
a = b;
//a →val2, b →val2.
```

Рис. 1. Присваивание.

На рис. 1 показано, что 2 ячейки памяти a и b после выполнения инструкции будут иметь одни и те же значения val2. При этом val2 – это не одно какое-либо конкретное значение, а целый класс значений, которые ячейка памяти может иметь в результате работы программы на различных путях выполнения.

Атрибуты позволяют описывать интересующие свойства. Атрибутами можно помечать идентификаторы значений переменных, области памяти и точки графа потока управления. Большинство видов анализов в Svace оперируют с атрибутами идентификаторов значений, т.к. интересующие свойства являются свойствами значений. Например, при поиске ошибок разыменования нулевого указателя необходимо проверить, что значение переменной, которую разыменовывают, не равно нулю. Для этого можно создать специальный атрибут, который будет обозначать свойство, что значение равно нулю (противоположное свойство - “значение не обязательно равно нулю”), и помечать им идентификаторы значений переменных, которые точно равны нулю. После этого в момент разыменования можно проверить значение указателя и значение атрибута.

Иногда для анализа свойств только значений не достаточно. Примерами могут служить функции завершения программы, подобные exit. В этом случае необходимо пометить путь выполнения программы после вызова функции exit как недостижимый, для чего граф потока управления помечается специальным атрибутом. Контекст на этой части графа не участвует в операциях слияния, а код, после вызова функции exit, может быть помечен как недостижимый.

Для примера атрибутов, которые ассоциируются с абстрактными ячейками, можно рассмотреть анализ утечек памяти. Утечки ищутся только для таких ячеек, которые выделены в динамической памяти. Чтобы правильно производить анализ необходимо специальным атрибутом пометить сами абстрактные ячейки памяти.

```
int* p = new[10];

p[2] = 7;
```

Рис 2. Атрибуты ячеек памяти.

На рис. 2 область памяти, на которую указывает указатель p, выделена в динамической памяти. Вторая инструкция присваивает константу 7 ячейке

памяти p[2]. При этом меняется значение ячейки, но сама ячейка по прежнему остаётся в динамической памяти. Поэтому атрибут, обозначающий динамическую память, некорректно ассоциировать со значением ячейки (т.к. после операции присваивания ячейке будет соответствовать новое значение без нужного атрибута).

4. Описание спецификаций функций

Для многих библиотечных функций Svace имеет спецификации, описывающие побочные эффекты от вызова функции. Наличие таких спецификаций продиктовано 2-мя обстоятельствами:

1. Исходный код библиотечных функций как правило отсутствует.

2. Часто сложно вывести семантику работы функции, анализируя её код.

Спецификация представляет собой функцию на языке Си, включающую вызов специальных предопределённых функций Svace, которые обрабатываются анализатором особым образом. Результатом анализа спецификации также является аннотация, хранящая все указанные эффекты от вызова функции.

Сокращённая спецификация для функции malloc приведена на рис. 3.

```
void *malloc(size_t size) {
      void* ptr;
      sf_overwrite(&ptr);
      sf_overwrite(ptr);
      sf_set_alloc_possible_null(ptr);
      sf_new(ptr, MALLOC_CATEGORY);
      return ptr;
}
```

Рис 3. Спецификация для malloc.

sf_overwrite сообщает, что изменяется значение указываемой ячейки памяти. Таким образом, с помощью двух вызовов sf_overwrite сообщается, что malloc инициализирует значение указателя ptr, а также указываемой памяти. Далее с помощью функции sf_set_alloc_possible_null сообщается, что указатель может иметь нулевое значение (но не обязательно имеет). И затем вызывается функция sf_new, чтобы показать, что ptr указывает на вновь выделенную в динамической области память. Константа MALLOC_CATEGORY указывает способ выделения новой памяти. Это требуется для возможности в дальнейшем проверить, что память была корректно освобождена с помощью нужной функции освобождения памяти.

Спецификация для free приведена на рис. 4.

```
void free(void *ptr) {
      sf_overwrite(ptr);
```

```
        sf_delete(ptr, MALLOC_CATEGORY);
}
```

Рис. 4. Спецификация для free.

В спецификации для free показывается, что изменяется состояние памяти, на которую указывает входной аргумент. А также то, что память освобождается. При этом освобождение происходит для памяти, выделенной с помощью функции malloc.

Пользователь может создавать свои собственные спецификации. Например, можно легко создать детектор, который будет проверять, что вся память, созданная пользовательской функцией xmalloc, освобождена с помощью xfree. Для этого достаточно создать аналогичные спецификации для этих функций и изменить константу MALLOC_CATEGORY на некую другую, например XMALLOC_CATEGORY.

Не смотря на то, что спецификации обрабатываются несколько иначе, чем обычные функции, момент вызова функции обрабатывается одинаково, при этом используется только аннотация функции, которая имеет единый формат.

В настоящее время созданы спецификации для большинства функций стандартной библиотеки языка Си. Для поддержки другой библиотеки необходимо создать спецификации для её основных функций, корректно описывающие побочные эффекты вызова.

5. Описание расширяемых модулей

Детектор - это компонент инструмента анализа, ответственный за поиск предупреждений определенного типа (или схожих типов). Детектор считается межпроцедурным, если он распространяет атрибуты за пределы одной функции и при выдаче предупреждений учитывает побочные эффекты от вызова функций.

Большинство детекторов реализовано в виде расширяемых модулей, которые регистрируются в диспетчере расширений и затем получают уведомления об интересующих событиях. Расширяемые модули расширения представляют из себя классы языка Java, реализующие интерфейс расширения Svace. Некоторые модули являются вспомогательными и не участвуют в выдаче предупреждений, а только распространяют атрибуты.

Примеры событий, которые доступны модулям расширения: add (сложение), sub (вычитание), deref (разыменование), bufferAccess (доступ к элементу массива), apply (трансляция атрибутов из контекста вызываемой функции в контекст вызывающей), annotate (создание аннотации). Все события

позволяют работать с атрибутами: проверять другие атрибуты или устанавливать свои, а также выдавать предупреждения.

Для межпроцедурных детекторов необходимо реализовать обработчики annotate и apply. Annotate позволяет внутренние атрибуты перевести в аннотацию функции, а apply применить аннотацию к контексту вызывающей функции в момент вызова.

Система расширяемых модулей позволяет не только улучшить расширяемость программы, но и в некоторых случаях уменьшить время анализа. Все общие действия выполняются 1 раз, при необходимости вызываются обработчики, на которые подписаны расширяемые модули, при этом работа всех модулей осуществляется одновременно. Модули не видят результаты друг друга до завершения операции, но при этом используют общие структуры данных. Благодаря чему после реализации нового детектора общее время работы программы увеличивается незначительно. Верно и обратное – если отключить половину всех детекторов, то скорость анализа не увеличится вдвое, т.к. Svace по прежнему должен тратить время на вычисление общих данных.

Каждый расширяемый модуль работает со своим набором атрибутов. Изменения этих атрибутов не должны влиять на другие модули. В некоторых случаях атрибут необходимо сделать видимым для других модулей, чтобы они могли переиспользовать её. Для этого необходимо зарегистрировать атрибут как видимый.

6. Описание типов атрибутов

Атрибуты группируются по типам атрибутов, которые позволяют разделять разные свойства и задавать возможное поведение самих атрибутов. Разные типы атрибутов могут иметь разный формат значений. Svace имеет набор предопределённых типов атрибутов, которые могут использоваться в качестве основы для пользовательских атрибутов. Наиболее важные будут перечислены в этой главе.

Часто во время анализа надо отследить какое-то свойство. Например, что значение указателя равно нулю. В данном случае не важно какое конкретное значение имеет указатель. При этом не интересуют случаи, когда указатель может быть равен нулю. Необходимо определить ситуации, когда указатель равен нулю на всех путях выполнения программы, и только в этом случае выдать предупреждение в случае его разыменования. В противном случае получится слишком много ложных срабатываний. Для того, чтобы отследить подобные свойства имеется двоичный атрибут BooleanFlag, который имеет 2 возможных значения – истина или ложь. Во время слияния графа потока управления есть 2 основные стратегии объединения двоичных атрибутов – для

описания отношений “может” и “должен”. Отношение “может” означает некоторое свойство, которое выполняется хотя бы на одном пути выполнения программы, отношения “должен” обозначает свойство, которое истинно на всех путях выполнения программы.

Соответственно имеются 2 подтипа OrBooleanFlag и AndBooleanFlag, которые ведут себя по разному в момент слияния. В примере с разыменованием описывается отношение “должен”, поэтому лучше использовать атрибут AndBooleanFlag, который будет сопоставлен результату слияния только в том случае, если описываемое свойство выполнялось на всех ветвях слияния.

В некоторых случаях интересны сразу оба отношения “может” и “должен”. Можно создать два двоичных атрибута для этого случая, но удобнее оперировать одним. Для этого используется троичный атрибут TernaryFlag с тремя значениями: “истина”, “ложь” и “возможно истина” (или “может быть”). Значение “истина” обозначает, что некоторое свойство всегда выполняется, значение “может быть” - истинно только на некоторых путях выполнения программы. Т.е. “истина” реализует отношение “должен”, а “может быть” отношение “может”.

Существуют и множество других предопределённых атрибутов. Интервальный атрибут позволяет оценить диапазон возможных значений для целых чисел. Параметризованный атрибут позволяет связать вместе две переменные. Подобное бывает необходимым при доступе к массиву. При этом в зависимости от детектора, либо массив имеет атрибут с ссылкой на индекс, либо наоборот.

7. Детекторы

7.1 FREE_OF_ARITHM

Наиболее простой из описываемых детекторов. Находит подозрительное использование функций освобождения памяти, заключающееся в удалении памяти по смещённому указателю. Позволяет найти ошибки, допущенные по невнимательности. В хорошо отлаженных проектах как правило подобное использование является намеренным.

```
static void
limited_free (gpointer mem)
{
	gpointer real = ((char*)mem) - HEADER_SPACE;
	//...
	free (real);
```

```
}
```

Рис. 5. Освобождение памяти по смещённому указателю (gtk+2.0).

В примере выше память, на которую ссылается указатель real, получена по смещению HEADER_SPACE. Такое использование является подозрительным, поэтому Svace сообщает об ошибке. В данном случае такое использование не является ошибкой, и пакет gtk+2.0 соответствующим образом переопределяет другие операции работы с динамической памятью.

Для реализации детектора, позволяющего находить подобные ситуации необходимо запомнить, что указатель был получен смещением другого указателя, а затем проверить, что значение такого указателя, не передаётся в функцию освобождения памяти.

Всю информацию о переменных и их значениях в Svace необходимо указывать с помощью атрибутов. Для того, чтобы пометить указатель как смещённый, был создан атрибут resultOfArithmeticOpFlag (результат арифметической операции). Так как предупреждение необходимо выдавать когда указатель сдвинут на всех путях, то в качестве типа атрибута был выбран AndBooleanFlag. Детектор подписывается на события add и sub, и проверяет что первый операнд имеет тип указателя, а второй является целым ненулевым числом. Операции add (сложение) и sub (вычитания), таким образом, применённые к указателям обозначают операцию смещения. После чего можно установить атрибут resultOfArithmeticOpFlag на значение результата операции. Больше не требуется никаких действий с этим атрибутом. В момент слияния путей флаг результата будет установлен автоматически, если этот флаг был установлен для всех сливаемых значений. В случае операции присваивания переменная получит новое значение, поэтому не требуется никаких дополнительных действий, чтобы снять данный флаг.

```
static void
limited_free (gpointer mem, int mode)
{
      gpointer real;
      if(mode==1) real = ((char*)mem) - HEADER_SPACE;
      else real = ((char*)mem) - HEADER_SPACE_EX;
      //...
      free (real);
}
```

Рис. 6. Слияние.

Если немного модифицировать экземпляр кода, как на рис. 6., то предупреждение также будет выдано, т.к. в момент слияния путей на разных ветках условного оператора, Svace сопоставит что переменная real имеет абстрактное значение val1 (равное ((char*)mem) – HEADER_SPACE) на одном

пути, и абстрактное значение val2 (равное ((char*)mem) – HEADER_SPACE_EX) на другом пути. После слияния переменной будет присвоено новое объединённое абстрактное значение val3, которое будет включать общие свойства значений на всех путях. После чего будут вызваны обработчики слияния для всех атрибутов этих значений. Эти действия производятся инфраструктурой Svace и нет необходимости заботиться о них при реализации детекторов.

На этом обработка смещения завершена, но требуется также правильно обработать вызов функции free для освобождения памяти. Для этого достаточно создать обработчик функции free, который будет вызван в нужный момент. Для очень многих детекторов такого решения достаточно, но в данном случае оно имеет некоторые недостатки. Во-первых, появляется жёсткая привязка к функции free. Если будут вызваны другие функции освобождения памяти, то придётся менять исходный код анализатора. Во-вторых, не будут обработаны межпроцедурные ситуации, когда вызывается некоторая функция, которая в свою очередь вызывает функцию free.

Чтобы решить первую проблему необходимо описание функции сделать спецификацией. Для обработки межпроцедурных ситуаций, надо создать атрибут, который будет реагировать на операции apply и annotate (см. Описание расширяемых модулей). Но в данном случае этих действий тоже не требуется, т.к. Svace уже имеет спецификацию для функции free (рис. 4), и модуль, который отслеживает, что память будет освобождена; при этом эта область памяти будет помечена специальным атрибутом deleteAttr. При такой реализации наш детектор будет находить также ситуации, в которых память освобождается оператором delete языка Си++, либо нестандартной функцией, для которой была создана аналогичная спецификация.

Осталось только среагировать на подозрительную ситуацию и в случае необходимости выдать предупреждение. Для этого надо подписаться на событие apply, которое будет вызвано в момент трансляции атрибутов из контекста вызываемой функции в контекст вызывающей. Это событие имеет два параметра: dst и src – оба параметра обозначают одну и ту же область памяти, но разные атрибуты (в случае, если область именованная, то они обозначают переменные в вызывающей функции и вызываемой функции). Параметр dst обозначает ячейку памяти на стороне вызывающей функции до инструкции вызова, параметр src обозначает эту же ячейку на стороне вызываемой функции после инструкции вызова. Все межпроцедурные атрибуты должны быть перенесены с параметра src на параметр dst, при этом каждый модуль переносит только те атрибуты, за которые он отвечает. В момент распространения атрибутов необходимо решить какие атрибуты из вызываемой функции перейдут в вызывающую. Для данного детектора достаточно только проверить, что dst имеет значение, помеченное флагом resultOfArithmeticOpFlag,а область памяти src помечена атрибутом deleteAttr.

Фактически это будет означать, что смещённый указатель был передан в функцию, которая освободит память, на которую он указывает.

7.2 FREE_INCOMPATIBLE и DELETE_INCOMPATIBLE

Предупреждение FREE_INCOMPATIBLE выдаётся в случае нахождения ситуаций несоответствия функций выделения памяти и функций освобождения. Например, при смешивании Си++ операторов new/delete с функциями Си библиотеки malloc/free. На практике встречается не часто.

Интересен подвид предупреждения (DELETE_INCOMPATIBLE), описывающий часто-встречающуюся ошибку, связанную с непониманием правильной работы с операторами выделения/освобождения памяти языка Си++ операторов new[]/delete. Если память была выделена с помощью оператора new[], то необходимо освободить её оператором delete[] для массивов. Частой ошибкой является освобождение памяти с помощью оператора delete для одиночных элементов. Нарушение этого правила ведёт к неопределённому поведению. Даже если в текущей версии программы не было выявлено ошибок, они могут возникнуть при смене компилятора, или изменении опций компилирования. Поэтому такие ошибки сложно находить в ходе ручного тестирования, но статические анализаторы легко справляются с такой задачей.

```
size_t SndFile::write(int srcChannels, float** src,
size_t n) {
      // ...
      float *buffer = new float[n * dstChannels];
      if (srcChannels == dstChannels) {
            // ...
      else {
            printf("SndFile:write channel mismatch %d ->
%d\n",
                  srcChannels, dstChannels);
            delete buffer; //необходимо использовать
delete[] buffer.
            return 0;
       }
      int nbr = sf_writef_float(sf, buffer, n) ;
      delete buffer; //необходимо использовать delete[]
buffer.
      return nbr;
}
```

Рис. 7. Неправильное освобождение памяти (проект muse).

Далее будет приведено описание работы детектора DELETE_INCOMPATIBLE, т.к. он немного проще, но при этом принципиальных различий в реализации с FREE_INCOMPATIBLE нет.

Для реализации необходимо создать один атрибут createPtrAttr для того, чтобы пометить результат new[], и один атрибут deletePtrAttr, чтобы пометить, что оператор delete освобождает свой аргумент несовместимым образом. Эти атрибуты устанавливаются на значения указателей, чтобы понять почему именно на значения, а не на сами указатели, стоит посмотреть следующий пример:

```
char* array = new char[MAX_PATH];

char* ptr = array;

delete ptr;
```

Рис. 8. Вновь присваивание.

В момент присваивания ptr = array свойства области памяти, в которой находится указатель ptr, не меняются, поэтому область ptr не будет помечена атрибутом createPtrAttr, и ошибка не будет найдена. Если же пометить значение указателя этим атрибутом, то в момент присваивания указатель ptr получит значение указателя array, которое уже помечено этим атрибутом, и ошибка будет найдена.

Т.к. в Си++ часто встречаются маленькие функции, то анализ надо делать межпроцедурным. Это в том числе позволит найти ошибки, когда выделение массива происходит в конструкторе некоторого класса, а некорректное освобождение в деструкторе. Чтобы сделать детектор межпроцедурным, достаточно реализовать обработчики событий apply и annotate. В данном случае реализация тривиальная – надо просто скопировать атрибут из источника в приёмник.

Детектирование ошибки так же не представляет сложности и делается аналогично как для детектора FREE_OF_ARITHM. В обработчике apply необходимо проверить, что переменная в контексте вызова имеет значение, помеченное атрибутом createPtrAttr, а значение переменной в контексте вызываемой функции помечено атрибутом deletePtrAttr.

При описании этого детектора заметно, что очень многие детали похожи, либо идентичны другим детекторам. Аналогично при реализации детектора не возникает множества новых проблем, практически всё необходимое уже встречалось в других детекторах и вынесено в общую для всех модулей часть, работа с который осуществляется анализатором Svace.

7.3 FREE_INCONSISTENT

Как правило, функция освобождения памяти должна обрабатывать входной аргумент в любом случае, независимо ни от каких условий. Если же освобождение происходит только в некоторых случаях, то это может привести к трудноуловимым утечкам памяти. Детектор FREE_INCONSISTENT проверяет, что входной параметр некоторой функции освобождается на некоторых путях, и не освобождается на других.

Как и FREE_INCOMPATIBLE анализ также выполняется над значениями указателей. Но в данном случае двоичного атрибута уже будет недостаточно. Если функция освобождения вызывается на всех путях, то ошибки нет, если же она не вызывается вообще, то ошибке тоже нет. Фактически необходимо найти ситуации, когда функция освобождения вызывается на некоторых путях, для этой цели прекрасно подходит троичный атрибут TernaryFlag.

При реализации нет необходимости создавать отдельную спецификацию для функции освобождения памяти, т.к. она уже создана. Необходимо только добавить действия на обработчик sf_delete, который будет устанавливать атрибут deletePtrTernaryAttr.

Троичный атрибут уже имеет обработчик, вызываемый в момент слияния путей. Если на обоих путях значение было равно “истина”, то и результат будет “истина”, если же на обоих атрибут имел значение “ложь”, то и результат будет “ложь”. В остальных случаях будет установлено значение “может быть”.

Для детектора FREE_INCONSISTENT значение “ложь” будет означать, что функция освобождения ни разу не была вызвана для данного указателя. Значение “истина” означает, что функция была вызвана на всех путях. А значение “может быть”, что функция была вызвана на некоторых путях, фактически это и есть искомая ситуация. Когда завершится анализ функции, Svace начнёт создавать аннотацию, для этого он вызовет обработчик annotate для необходимых атрибутов. В этот момент и можно проверять, что флаг deletePtrTernaryAttr имеет значение либо “ложь”, либо “истина”.

Важно подчеркнуть, что анализ будет производиться для каждого отдельного значения.

```
void freeSomething(char* ptr1, char* ptr2, int index) {
      if(index==1)
            free(ptr1);
      else
            free(ptr2);
}
```

Рис. 9. Слияние путей.

В этом надуманном примере функция free вызывается на всех путях, но в первом случае она вызывается для ptr1, а во втором для ptr2. Таким образом

для входного параметра ptr1 функция free вызывается только на одном из путей выполнения, поэтому будет выдано предупреждение.

Если бы в начале функции существовало присваивание ptr1 = ptr2, тогда оба указателя имели бы одно и тоже значение и ошибки выдано не было бы. Т.к. присваивания нет, то Svace считает параметры разными, если по каким-либо причинам параметры являются псевдонимами друг друга, то будет выдано ложное предупреждение.

В ближайших планах является создание дополнительного анализа указателей, который во многих случаях позволит понять, что указатели являются псевдонимами.

7.4 DEREF_OF_NULL.RET

Часто, когда функция возвращает указатель, нулевое значение используется в случае возникновения какой-либо ошибки. В этом случае необходимо проверить, что указатель не нулевой, перед его разыменованием. Для случая отсутствия подобной проверки Svace имеет предупреждение DEREF_OF_NULL.RET.

Все такие функции можно разделить на 2 категории: библиотечные и пользовательские.

В документации к библиотечной функции как правило явно задано, что она может вернуть нуль, при этом тело функции может быть недоступно. Поэтому имеет смысл делать явные спецификации для подобных функций.

```
char *getenv(const char* key) {

      char *str;
      sf_overwrite(&str);
      sf_set_possible_null(str);
      return str;
}
```

Рис. 10. Спецификация для getenv.

Обработчик специальной функции sf_set_possible_null должен установить атрибут PossibleNull на значение аргумента. Этот атрибут означает, что значение некоторого указателя может быть равно нулю, поэтому указатель желательно проверить на нуль перед разыменованием. Атрибут имеет подтип OrBooleanFlag – в случае слияния путей, результат будет иметь этот атрибут, если его имеет хотя бы один из аргументов. Очевидно, что реализуется отношение “может”, то есть ищутся ситуации, когда переменная равна нулю хотя бы на одном пути выполнения программы.

Необходимо заметить, что создавать спецификации нужно только для таких функций, которые возвращают нуль в случае неконтролируемой ошибки, которая напрямую не зависит от входных аргументов, например в случае отсутствия файла. Иначе будет множество ложных срабатываний, т.к. результат функции может быть детерминирован и зависеть от потока управления. На рис. 11 на первой строке у массива buf последнему элементу присваивается символ новой строки. Таким образом массив будет гарантированно иметь этот символ хотя бы 1 раз. Далее с помощью функции strchr ищется первое вхождение символа новой строки и вместо него устанавливается символ окончания строки, таким образом обрубая строку в этом месте. Функция strchr возвращает нуль, если ничего не было найдено. Таким образом для предотвращения ошибки разыменования нулевого указателя необходимо проверить, что результат strchr не нулевой. Но в данном примере, очевидно, что функция всегда вернёт ненулевое значение.

```
buf[sizeof(buf) - 1] = '\n';
*strchr(buf, '\n') = '\0';
```

Рис. 11. Вызов детерминированной функции (проект busybox).

Для реализации в самом простейшем случае необходимо проверить, что во время разыменования значение указателя имеет атрибут PossibleNull, и в этом случае выдать предупреждение (подтипа DEREF_OF_NULL.RET.LIB). Также необходимо реагировать на сравнение указателя с нулём, чтобы понять, что указатель проверен на нуль. Для простой реализации достаточно снимать этот атрибут, если указатель сравнивается с нулевой константой.

Более полная реализация будет отслеживать, что другие указатели могут иметь нулевое значение, и соответственно сравнение с ними может быть эквивалентно сравнению с нулевой константой.

Довольно интересно использование функции malloc стандартной библиотеки. Эта функция выделяет память заданного размера и в случае неудачи возвращает нулевой указатель. На современных системах, где при нехватке оперативной памяти, её часть может сгружаться на диск, такая ситуация довольно редка. Кроме этого, часто в случае ошибки сделать уже ничего нельзя, поэтому многие программисты не проверяют результат malloc. По этой причине для функции malloc выделено специальное предупреждение DEREF_OF_NULL.RET.ALLOC. Если в проекте сознательно игнорируют проверку результата функции malloc, то это предупреждение можно легко отключить.

Поиск ошибок для пользовательских функций немного сложнее. Необходимо по коду определить, что функция иногда возвращает нулевое значение. Механизм определения не будет подробно описываться. Основная идея – использовать вспомогательный тернарный атрибут, у которого значение “истина” означает, что указатель равен нулю, а значение “ложь”, что не равен. В случае слияния путей, если один из атрибутов истинен, а другой ложен, значение результата будет “возможно истинен”, что фактически означает, что

указатель может иметь нулевое значение. А это почти эквивалентно семантике атрибута possibleNull, разница лишь в том, что атрибут possibleNull устанавливается на возвращаемое значение.

Детектирование ошибок с пользовательскими функциями абсолютно аналогично, с той лишь разницей, что выдаётся предупреждение DEREF_OF_NULL.USER. И также как и в случае с библиотечными функциями возможны ложные срабатывания, когда код, возвращающий нуль, каким-либо образом зависит от входных аргументов. В этом случае при использовании функции можно заранее знать, что она не вернёт нулевое значение, т.к. её аргументы не случайны. Поэтому можно не проверять возвращённый указатель. Отличить подобные ситуации с помощью статического анализа очень сложно и первое время предупреждение имело очень высокий уровень ложных срабатываний. Какие-либо попытки улучшить результат за счёт усложения модели анализа не давали существенных результатов. Svace выдавал огромное количество предупреждений, которые на первый взгляд казались истинными, но при более тщательной проверке выяснялось, что в данном конкретном случае функция не может вернуть нулевое значение.

Улучшить ситуацию удалось с помощью технологии Z-ranking. Подробно она описана в [10]. Эта технология позволяет откинуть часть предупреждений на основе статистической информации. Идея заключается в том, что если результат функции никогда не проверяется в коде, следовательно, существует негласное правило, что она не может вернуть нулевое значение. Если же результат, в большинстве случаев проверяется, а иногда нет, то в последнем случае с большой долей вероятности удалось найти истинную ошибку. Z-ranking позволяет дать количественную оценку вероятности, что предупреждение будет ложным. Таким образом, можно не только отсеить предупреждения, которые могут быть ложными, но и отсортировать все предупреждения в порядке уменьшения вероятности, что они будут истинными.

Список литературы

[1] С.С. Гайсарян, А.В. Чернов, А.А. Белеванцев, О.Р. Маликов, Д.М. Мельник, А.В. Меньшикова. О некоторых задачах анализа и трансформации программ. Труды ИСП РАН, No5, с.7-41, 2004.

[2] О.Р. Маликов, А.А. Белеванцев. Автоматическое обнаружение уязвимостей в программах. Материалы конференции «Технологии Майкрософт в теории и практике программирования», Москва, 2004.

[3] О.Р. Маликов. Автоматическое обнаружение уязвимостей в исходном коде программ. Известия ТРТУ, No4, с. 48-53, 2005.

[4] В.С. Несов, О.Р. Маликов. Использование информации о линейных зависимостях для обнаружения уязвимостей в исходном коде программ. Труды ИСП РАН, No9, с. 51-57, 2006.

[5] О.Р. Маликов, В.С. Несов. Автоматический поиск уязвимостей в больших программах. Известия ТРТУ, Тематический выпуск «Информационная безопасность», No7 (62), с.114-120, 2006.
[6] В.С. Несов. Использование побочных эффектов функций для ускорения автоматического поиска уязвимостей в программах. Известия ЮФУ. Технические науки. Тематический выпуск «Информационная безопасность». Таганрог: Изд-во ТТИ ЮФУ, 2007. No 1(76), с. 134-139.
[7] В.С. Несов, С.С. Гайсарян. Автоматическое обнаружение дефектов в исходном коде программ. Методы и технические средства обеспечения безопасности информации: Материалы XVII Общероссийской научно-технической конференции. СПб.: Изд-во Политехн. ун-та, 2008, с.107.
[8] В.С. Несов. Автоматическое обнаружение дефектов при помощи межпроцедурного статического анализа исходного кода. Материалы XI Международной конференции «РусКрипто'2009».
[9] Vladimir Nesov. Automatically Finding Bugs in Open Source Programs. Electronic Communications of the EASST 20. ISSN 1863-2122, 2009.
[10] T. Kremenek, D. Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. Static Analysis, pp. 295-315, 2003 — Springer.

Mechanisms for extending the system of static analysis Svace by new types of detectors of vulnerabilities and critical errors

Arutyun Avetisyan <arut@ispras.ru>
Alexey Borodin <alexey.borodin@ispras.ru>

Abstract. A static analysis tool Svace finding vulnerabilities and critical errors in the source code of C/C++ programs is developed in the ISP RAS. The purpose of Svace is to find as many errors as possible with low level of false positives and suitable use of available resources. Important requirements for this kind of systems are scalability and extensibility. The article presents the mechanism supporting the addition to the Svace system detectors of new kinds of errors that preserves the scalability. Using the mechanism illustrated by the four detectors developed errors.

Keywords: static analysis; dataflow analysis; extensibility; null pointer dereference; improper use of dynamic memory.

Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах использующих сетевые сокеты[1]

Исаев И.К., Сидоров Д.В., Герасимов А.Ю., Ермаков М.К.,
iisaev@ispras.ru, sidorov@ispras.ru, agerasimov@ispras.ru, mermakov@ispras.ru

Аннотация. В данной статье рассматривается модификация и применение инструмента Avalanche для проведения динамического анализа и тестирования приложений, получающих входные данные через сокеты. Вводится концепция замены получаемых данных, описывается реализация этой концепции при помощи средств Valgrind. Разбирается перехват и обработка системных вызовов, используемых при работе с сокетами. Рассматривается применение модифицированной версии инструмента для анализа сетевых приложений с открытым исходным кодом, перечисляются обнаруженные во время анализа дефекты.

Ключевые слова: динамический анализ; обнаружение ошибок; тестирование.

1. Введение.

Известно, что проблемам надежности программ, работающих с данными, получаемыми через сеть, уделяется традиционно повышенное внимание. Наличие уязвимости в такой программе может отразиться на всех её пользователях, поскольку данные, вызывающие возникновение ошибки, могут быть быстро переданы по сети большому числу пользователей. Значительные усилия тратятся на борьбу с уязвимостями безопасности в Web-приложениях.

Инструмент Avalanche в своем исходном виде использует динамический анализ для создания входных данных, демонстрирующих ошибки в анализируемой программе. При этом в качестве источника входных данных для программы поддерживается лишь единственный файл, с которым работает программа. Расширение списка возможных источников входных данных для программы так, чтобы он включал чтение данных из сокета, - основная цель данной работы. При этом результат анализа, как и в немодифицированной

[1] Работа поддержана грантом РФФИ №11-07-00466-а и ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы» ГК №07.514.11.4040

версии инструмента Avalanche, должен являться не только список обнаруженных ошибок, но и набор входных данных для программы, приводящих к её возникновению.

Инструмент Avalanche изначально представлен в работе [1]. Напомним кратко его структуру и основные свойства. Для описания Avalanche вводится понятие символических или помеченных (tainted) данных - данных, полученных программой из внешнего источника (стандартный поток ввода, файлы, переменные окружения и т. д.)

Avalanche состоит из 4 основных компонент (см. Рисунок 1): двух модулей расширения (плагинов) Valgrind - Tracegrind и Covgrind, инструмента проверки выполнимости ограничений STP и управляющего модуля. Tracegrind динамически отслеживает поток помеченных данных в анализируемой программе и собирает условия для обхода её непройденных частей и для срабатывания опасных операций. Управляющий модуль передаёт собранные ограничения STP для проверки их выполнимости. Если какие-то из условий выполнимы, то STP определяет те значения всех входящих в условия переменных (в том числе и значения байтов входного файла), которые обращают условие в истину.

- Если какое-то из ограничений для выполнения опасной операции выполнимо, управляющий модуль запускает программу ещё раз (на этот раз безо всякой инструментации) с соответствующим входным файлом для того, чтобы подтвердить обнаруженную ошибку.
- Выполнимые ограничений для обхода ранее не пройденных частей программы определяют набор входных данных для новых запусков программы. Таким образом, после проведения одной итерации анализа STP автоматически вычисляет набор новых входных данных для последующих итераций. Из этого набора необходимо выбрать определённое значение для запуска следующей итерации. При этом в первую очередь должны обрабатываться входные данные на которых наиболее вероятно возникновение ошибки. Для решения этой задачи используется эвристическая метрика - количество ранее не обойденных базовых блоков в программе. Для измерения значения эвристики используется компонент Covgrind, в функции которого входит также фиксация возможных ошибок выполнения. Covgrind - гораздо более легковесный модуль, нежели Tracegrind, поэтому возможно сравнительно быстро измерить значения звристики для всех полученных ранее входных файлов и выбрать входной файла с наибольшим её значением.
- Covgrind может обнаруживать только те ошибки, которые приводят к аварийному завершению программы. Для того, чтобы обнаруживать иные типы ошибок, может быть использован один из наиболее популярных инструментов Valgrind – Memcheck.

Более подробное описание исходной версии Avalanche можно найти в статье [1]. Данная статья является её непосредственным логическим продолжением. Здесь используется та же терминология, что и в [1], опускается существенная часть вопросов, ранее рассмотренных в [1] и т. п. Работа имеет следующую структуру. В разделе 2 вводится концепция замены получаемых входных данных, описывается общая схема работы. В разделе 3 описывается специальный формат файла, который Avalanche использует для запуска анализируемого приложения с различными входными данными. В разделе 4 описывается перехват системных вызовов, используемых при взаимодействии клиента с сервером. В разделе 5 приводится алгоритм замены получаемых входных данных. В шестом разделе перечисляются изменения в компонентах Tracegrind и Covgrind, необходимые для поддержки сокетов. В разделе 7 рассматриваются результаты работы модифицированной версии Avalanche, перечисляются обнаруженные дефекты. Раздел 8 описывает имеющиеся ограничения предложенного метода, а также намечает направления для дальнейшего исследования.

2. Организация анализа

2.1. TCP и UDP сокеты

Большинство сетевых приложений используют архитектуру клиент-сервер. Для организации взаимодействия между удаленными процессами на транспортном уровне используются протоколы TCP (ориентированный на установку соединения) и UDP (ориентированный на отправку сообщений). Типичная последовательность системных вызовов на стороне клиента при использовании протокола TCP имеет вид:

- socket
- connect
- read/write или send/recv

Сервер же будет использовать иную последовательность:

- socket
- bind
- listen
- accept
- read/write или send/recv

В случае протокола UDP последовательности системных вызовов также будут различными. Поскольку Avalanche использует возможности Valgrind[2] для динамической инструментации и перехвата системных вызовов, то различные последовательности системных вызовов требуют, вообще говоря, различных

моделей работы. В данной работе рассматривается анализ клиентских приложений, использующих сокеты, ориентированные на установку соединения (TCP-сокеты).

2.2. Взаимодействие с сервером

Если в случае простого чтения из файла для запуска программы на новых входных данных достаточно лишь изменить содержимое этого файла, то в случае клиент-серверного взаимодействия запуск программы на новых входных данных не так прост. Очевидно, что сокеты не могут быть источником потенциально опасных данных в случае изолированной работы анализируемого приложения-клиента. С точки зрения логики работы сетевого приложения, клиент составляет лишь часть единой программы – связки клиент-сервер. Поэтому решение вопроса об организации взаимодействия анализируемого приложения с сервером является одним из ключевых для успешного проведения анализа.

Допустим, что во время анализа приложения-клиента доступно соответствующее приложение-сервер. Таким образом, клиент может осуществлять обмен данными с сервером в обычном режиме. Тогда для того, чтобы заставить приложение-клиент работать с измененными входными данными, достаточно подменить данные приходящие от сервера, на данные, полученные в результате проверки выполнимости условий, собранных при предыдущих запусках программы. Такую подмену нужно производить непосредственно в местах появления этих данных в программе, т. е. сразу после выполнения вызовов read или recv, читающих данные из сокетов, соответствующих соединению с сервером.

С использованием такого подхода общая схема работы инструмента Avalanche претерпевает следующие изменения. При первом запуске компонента Tracegrind, помимо обычных файлов с запросами для проверки выполнимости условий, формируется файл специального формата, содержащий все данные, полученные анализируемым приложением от сервера. После проверки выполнимости собранных условий содержимое этого файла изменяется управляющим модулем: в него поочередно подставляются те значения, которые делают эти условия выполнимыми (проверка выполнимости и подбор соответствующих значений, как и при работе с файлами, осуществляются компонентом STP [3]). При всех последующих запусках компонентов Covgrind и Tracegrind они заменяют приходящие от сервера данные в соответствии с содержимым модифицированных файлов с данными. Кроме того, если во время работы компонента Tracegrind приложение получает от сервера порцию данных большего размера, чем есть в файле с данными для замены, то “остаток” полученных от сервера данных приписывается к файлу с данными для замены в неизменном виде. Далее рассмотрим общую схему более детально.

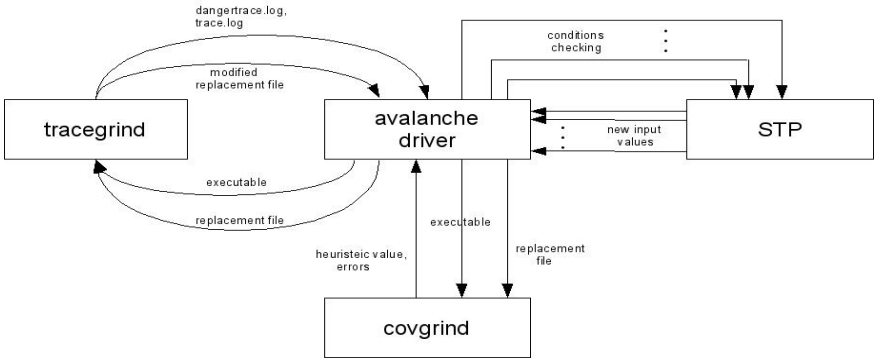


Рис. 1. Общая схема

3. Формат файла с данными для замены

Управляющий модуль по результатам проверки выполнимости формулы компонентом STP формирует файл, содержащий такие данные, что при их получении клиентом со стороны сервера выполнение должно пойти по новому пути. Этот файл имеет следующий вид:

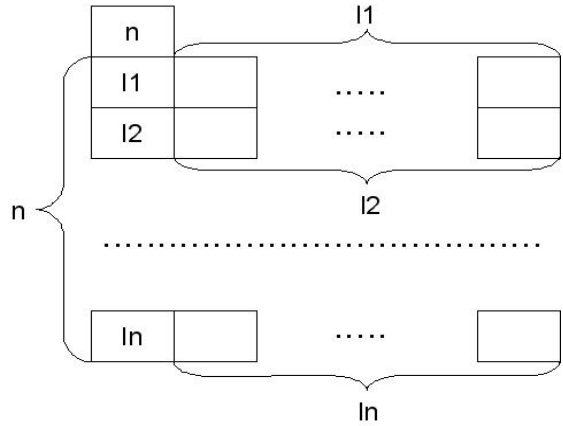


Рис. 2. Формат файла с данными для замены.

Первые четыре байта содержат целое число n – количество различных соединений клиента и сервера (оно равно количеству вызовов connect со

стороны клиента в анализируемой части трассы клиента при предыдущем запуске). Далее следует n групп. В i-ой группе первые четыре байта – это длина группы li (т. е. число байт, прочитанных клиентом по i-ому соединению), а следующие li байт – это собственно полученные клиентом данные.

В таком же формате сохраняются и данные, приводящие к возникновению ошибки в исследуемом приложении.

4. Перехват системных вызовов

Поток потенциально опасных данных в анализируемом приложении отслеживается аналогично тому, как это делалось в программах, работающих с обычными файлами. Однако есть одно существенное различие – иной источник потенциально опасных данных. Для того, чтобы зафиксировать факт получения данных через сокет, приходится перехватывать следующие системные вызовы, ответственные за работу с сокетами:

- connect
- read
- recv
- close

Эти системные вызовы необходимо перехватывать как в компоненте Tracegrind, так и в компоненте Covgrind, поскольку, хоть Covgrind и не отслеживает поток потенциально опасных данных в программе, но оба компонента должны осуществлять замену приходящих от сервера данных при их получении клиентом.

В общем случае, клиент может устанавливать за время работы несколько соединений с сервером. Каждое установленное соединение идентифицируется при помощи сквозной нумерации – все соединения нумеруются согласно порядку их установления, начиная с нуля. Оба компонента поддерживают во время работы внутреннюю структуру данных, позволяющую устанавливать соответствие между номерами дескрипторов сокетов и номером соединения, а также подсчитывать объем данных, полученных по каждому из соединений.

Если в случае чтения данных из файла для его идентификации используется имя, то сетевое соединения идентифицируется парой IP-адрес плюс номер порта. При перехвате системного вызова connect анализируется второй параметр вызова – адрес сервера, с которым устанавливается соединение. IP-адрес и номер порта, по которому сервер ждёт подключения от клиента, должны быть переданы Avalanche в качестве параметров запуска (которые управляющий модуль затем передаёт Tracegrind при каждом его запуске). Если IP-адрес и номер порта, содержащиеся во втором параметре успешно завершившегося вызова connect совпадают с IP-адресом и номером порта,

указанными в параметрах запуска, то такой вызов connect устанавливает соединение клиента с сервером, и все данные переданные по этому сокету помечаются как потенциально опасные. При этом запоминается, что первый параметр вызова connect (дескриптор сокета) соответствует очередному соединению. Счетчик полученных данных для нового соединения устанавливается равным нулю.

5. Замена данных при работе компонентов Tracegrind и Covgrind

При перехвате системных вызовов read или recv анализируется их первый параметр – дескриптор, по которому осуществляется чтение данных. Если этот дескриптор соответствует какому-то из установленных и не закрытых соединений, то такой вызов является источником потенциально опасных данных в программе.

При перехвате этих вызовов и осуществляется непосредственная замена данных. При этом если количество байт уже полученных по тому соединению, из которого осуществляется чтение, не превосходит количества байтов в соответствующем файле с данными для замены, то прочитанные данные заменяются на данные из файла. В противном случае замены не происходит, и анализируемое клиентское приложение получает от сервера именно те данные, которые и были переданы.

Поясним данное правило на примере. Пусть перехвачен вызов read, читающий ri байт из сокета, соответствующего i-ому соединению. Пусть ранее из этого соединения было прочитано bi байт, а длина соответствующей порции данных в файле с данными для замены равна li. Тогда возможны следующие варианты:

a) bi + ri <= li

В этом случае все ri прочитанных байт заменяются на байты с номерами с bi + 1 по bi + ri в i-ой группе байтов из файла с данными для замены.

b) bi < li, bi + ri > li

Тогда первые li - bi из прочитанных ri байт заменяются на байты с номерами с bi + 1 по li в i-ой группе байтов из файла с данными для замены, а последующие ri - (li - bi) байтов передаются программе без изменений. Кроме того, в случае работы компонента Tracegrind, эти ri - (li - bi) байтов дописываются в конец i-ой группы.

c) bi >= li

Тогда все прочитанные данные передаются программе без изменений и, кроме того, в случае работы компонента Tracegrind, эти данные дописываются в конец i-ой группы.

После этого счетчик прочитанных данных увеличивается на ri: bi += ri.

Системный вызов recv обрабатывается аналогично за одним исключением. Если в четвёртом параметре вызова recv установлен флаг MSG_PEEK, то это означает, что прочитанные данные не удаляются из очереди полученных данных, и последующие обращения к вызовам read или recv прочитают те же данные. В таком случае счетчик прочитанных данных после завершения обработки увеличивать не надо.

6. Изменения в компонентах Tracegrind и Covgrind

6.1. Моделирование сокетов

Сокеты представляются аналогично тому, как это делается с входным файлом, адресным пространством программы и набором регистров в обычной версии Avalanche. Каждое установленное соединение представляется в виде массива с 32-битными индексами. При перехвате каждого успешного вызова connect, устанавливающего соединение с сервером, в файлы с STP-декларациями добавляется декларация вида

```
socket_2 : ARRAY BITVECTOR(32) OF BITVECTOR(8);
```

Здесь 2 – это порядковый номер установленного соединения (начиная с 0).

6.2. Декларации для системных вызовов

При перехвате системных вызовов read и recv создаётся набор деклараций, отражающих факты равенства значений, прочитанных из сокета, содержимому ячеек памяти, по которым осуществлялась запись:

```
memory_1 : ARRAY BITVECTOR(32) OF BITVECTOR(8) =
memory_0 WITH [0hex08090218] := socket_0[0hex00000000];
```

Данная декларация моделирует чтение первого байта, полученного по первому соединению, установленному анализируемым приложением, в ячейку памяти с адресом 0x08090218. Аналогично чтению из файла, каждый из вызовов read и recv приводит к появлению цепочки таких деклараций – по числу прочитанных байт.

6.3. Тайм-аут для предотвращения блокировки анализа

Как уже упоминалось, для поддержки запуска программы на различных значениях входных данных, компонент Covgrind вынужден перехватывать системные вызовы, работающие с сокетами и осуществлять подмену данных. Кроме того, изменилась семантика таймера, используемого компонентом Covgrind. Если в случае анализа приложения, осуществляющего чтение из файла, значение устанавливаемого при старте приложения таймера означало максимальное допустимое время работы приложения, и его истечение интерпретировалось как показатель наличия в программе бесконечного цикла

(т. е. ошибки), то в случае анализа сетевых приложений продолжительную работу клиента можно трактовать как ошибку с гораздо меньшей степенью уверенности. Длительная работа клиента может быть связана, например, со спецификой конкретного сетевого приложения (например, проигрывание потокового аудио) или с ожиданием отклика сервера. Поэтому при анализе сетевых клиентских приложений таймер используется для борьбы с блокировкой анализа – Covgrind измеряет покрытие анализируемой программы и фиксирует наличие ошибок только за отведённое таймером время. По истечении таймера работа Covgrind прерывается, но это не считается признаком наличия ошибки в анализируемой программе.

6.4. Воспроизведение обнаруженного дефекта

Помимо этого, Covgrind используется для воспроизведения найденной ошибки. Если в случае простого чтения данных из файла для воспроизведения ранее обнаруженной ошибки достаточно запустить программу с предварительно сохранённым файлом, то в случае сетевого приложения продемонстрировать уязвимость сложнее, даже имея соответствующие входные данные. Можно было бы для каждой обнаруженной ошибки создать сервер, отсылающий соответствующие данные клиенту, но это, в общем случае, требует хорошего знания протокола, по которому сервер общается с клиентом, и не позволяет автоматически (т. е. без вмешательства эксперта) воспроизводить ошибочную ситуацию. Однако, можно использовать всё ту же технику замены приходящих от сервера данных на данные, сгенерированные Avalanche и демонстрирующие уязвимость программы. Таким образом, для воспроизведения ошибки достаточно отдельно (вне окружения Avalanche) запустить Covgrind с ранее сохранённым файлом в соответствующем формате (см пункт 2).

7. Результаты работы Avalanche на реальных приложениях, получающих данные через сокеты

Эффективность поиска ошибок при помощи Avalanche была исследована на большом числе проектов с открытым исходным кодом. В данном разделе будут рассмотрены результаты работы на следующих 6 проектах:

- fetchmail-6.3.14
  Fetchmail – утилита, используемая для сбора почты с удалённых POP3, IMAP, ETRN или ODMR почтовых серверов и доставки локальным пользователям.
- mencoder (версия из репозитория проекта)
  MEncoder – инструмент кодирования/декодирования и фильтрации данных в аудио и видео форматах. Является частью проекта MPlayer, может использоваться для конвертации файлов во всех форматах, поддерживаемых MPlayer. Поддерживает работу с потоковым аудио, получаемым от удаленного сервера.

- nmap-5.21
  Nmap — утилита, предназначенная для настраиваемого сканирования IP-сетей и определения состояния объектов сканируемой сети (портов и соответствующих им служб). Поддерживает в том числе и TCP-сканирование.
- wget-1.12
  Wget – неинтерактивная консольная программа для загрузки файлов по сети. Поддерживает протоколы HTTP, FTP и HTTPS.
- ogg123 (vorbis-tools-1.2.0)
  Ogg123 – проигрыватель для воспроизведения медиаданных в форматах Ogg Vorbis, Ogg Speex, Ogg FLAC. Как и MEncoder, поддерживает работу с потоком данных, получаемых от удаленного сервера.

В соответствии с концепцией замены приходящих от сервера данных все тестируемые приложения во время анализа взаимодействовали с “ожидаемым” приложением сервером (т. е. с сервером, поддерживающим тот же протокол, что и анализируемое приложение). Для анализа mencoder и ogg123 использовался сервер потокового аудио Icecast. Fetchmail работал с почтовым сервером Dovecot. Nmap осуществлял TCP-сканирование порта, на котором также работал почтовый сервер Dovecot. Wget загружал файл с HTTP-сервера nginx.

При анализе fetchmail, ogg123 и wget устанавливался шестисекундный тайм-аут для работы компонента Covgrind. Глубина просмотра во всех случаях указывалась равной 100 условиям. На работу Avalanche отводилось 3600 секунд (час), затем анализ принудительно завершался. Результаты работы приведены в таблице 1.

Перечислим обнаруженные дефекты:

- mencoder
  Дефект связан с разыменованием нулевого указателя. Возникал при ошибке разбора пакета, полученного от сервера. Был исправлен разработчиком в течение суток после момента сообщения.
- wget
  Аналогично предыдущему случаю, разыменование нулевого указателя при разборе полученного пакета. Разработчик сообщил, что дефект уже был ранее исправлен в текущей разрабатываемой версии приложения.
- nmap, ogg123, fetchmail
  Дефектов в результате анализа не обнаружено.

По-видимому, все обнаруженные дефекты реализуются в ситуации, когда клиент получает от сервера “неожиданные” данные, то есть данные, получение которых не предусмотрено протоколом взаимодействия. Таким образом, при полностью корректной работе сервера подобные ошибки в приложении-клиенте никак себя проявлять не будут. Однако в случае, если при реализации протокола на стороне сервера допущена ошибка, или в качестве сервера потенциально может выступать «ненадежное» приложение

(т. е. приложение, намерено пытающееся спровоцировать ошибку на стороне клиента), такие ошибки обычно приводят к аварийному завершению приложения-клиента, и могут быть классифицированы как серьезные уязвимости.

	fetchmail	mencoder	nmap	wget	ogg123
общее число тестов	396	163	163	613	36
начальное покрытие	8199	7751	17131	6833	9154
Прирост покрытия	41 (0.5%)	226 (3%)	380 (2.2%)	39 (0.5%)	103 (1.1%)
точность предсказания	79%	100%	100%	93%	100%
время tracegrind	35%	0.3%	38.4%	10.5%	1%
время STP	8%	91%	<1%	1%	94%
время covgrind	57%	8.7%	61%	88.5%	5%
опасные операции	0	0	0	0	0
число дефектов	0	1	0	1	0
тестовые данные	0	1	0	9	0
tmin	-	98	-	95	-
tmax	-	98	-	95	-

Таблица 1.Результаты и статистика работы Avalanche при анализе сетевых клиентских приложений

8. Ограничения и направления дальнейшего развития

Стоит ещё раз подчеркнуть, что анализ приложения требует наличия сервера, работающего по понятному приложению протоколу. Изолированный анализ приложения требовал бы восстановления протокола лишь по его клиентской части, что представляется крайне затруднительным. При работе над поддержкой сокетов в Avalanche была первоначально предпринята попытка

организовать анализ таким образом, чтобы серверная часть приложения эмулировалась управляющим модулем инструмента Avalanche. При этом подход к эмуляции сервера состоял в следующем: при достижении анализируемым клиентским приложением (во время запуска с инструментирующим модулем Tracegrind или Covgrind) одного из системных вызовов для работы с сокетами (connect, select, poll, read, recv), Tracegrind или Covgrind сообщают об этом вызове управляющему модулю, и тот выполняет ответные действия по обработке соответствующего вызова. В частности, в случае выполнения вызова connect управляющий модуль должен принять новое соединения (выполнить системный вызов accept), а в случае вызовов read или recv записать ранее определённый при помощи компонента STP порцию данных (см. Рисунок 3).

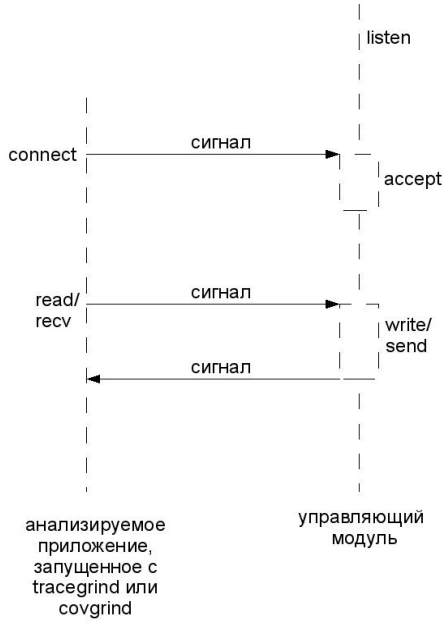


Рис. 3. Схема работы с эмуляцией сервера

Это решало бы проблему запуска программы с новыми входными данными. Однако попытка реализации данного подхода привела к обнаружению ряда сложностей:

1) Необходимость синхронизации между управляющим модулем и анализируемым при помощи Tracegrind или Covgrind приложением. Например, для обработки системного вызова read инструментирующий модуль должен каким-то образом сообщить об этом вызове управляющему модулю и приостановить работу программы, а управляющий модуль

должен, получив соответствующее сообщение, записать необходимую порцию данных, и затем выслать инструментирующему модулю подтверждение о завершении обработки, чтобы тот мог продолжить свое выполнение. В сочетании со сложностью межпроцессного взаимодействия в рамках фреймворка Valgrind это приводило к существенному усложнению компонентов Tracegrind и Covgrind.

2) Проблема обратной инженерии протокола взаимодействия клиента и сервера. Поскольку в общем случае про анализируемое приложение заранее не известно ничего, то возникает необходимость учёта большого числа различных вариантов при взаимодействии клиента и сервера – использование неблокирующих сокетов, установка ненулевых тайм-аутов в системных вызовах poll и select и т. д. Приходится либо вводить жесткие ограничения на анализируемые приложения (вида “анализируемое приложение должно использовать только блокирующие сокеты” и т. п.), либо пытаться организовать эмуляцию сервера управляющим модулем с поддержкой всех возможных вариантов, что крайне трудоёмко, если вообще возможно.

Поэтому поиск дефектов с эмуляцией сервера управляющим модулем успешно работал только на простых, искусственно созданных примерах уязвимых приложений. В итоге успешного анализа удалось добиться, только используя замену данных, приходящих от независимо работающего сервера.

Кроме того, как уже было сказано, на данный момент поддержка сокетов в качестве источника потенциально опасных данный реализована в инструменте Avalanche только для клиентских приложений и только для TCP-сокетов. Очевидно, встает вопрос о возможности поддержи UDP-сокетов, а также о возможности анализа серверной части приложений и о применимости концепции замены данных к этим случаям. Исследование этих вопросов привело к следующим выводам.

По-видимому, поддержка UDP-сокетов представляет чисто техническую задачу. Напротив, поддержка возможности анализа серверных приложений представляется более сложной задачей. Основные причины этого таковы:

- Приложение-сервер часто представляет собой несколько параллельно работающих процессов.
  Наличие нескольких процессов, одновременно работающих с потенциально опасными данными, усложняет сбор трассы и генерацию запросов для получения новой порции тестовых данных. Придётся собирать отдельные трассы для каждого из таких процессов, что усложнит как компонент Tracegrind, так и управляющий модуль, осуществляющий разбор собранных трасс.
- Необходимость многократного запуска приложения-клиента.
  Как уже было сказано, для замены приходящих данных во время анализа приложения-клиента, необходим работающий сервер. Для этого достаточно запустить такой сервер единственный раз непосредственно

перед началом анализа. При этом многократные запуски клиента по ходу анализа не влияют (по крайней мере, существенно) на работоспособность сервера.

Но многократные запуски и завершения сервера (во время его анализа) в общем случае потребуют и многократного запуска вспомогательного приложения-клиента, поскольку при неработающем сервере многие клиенты прекращают свою работу. Это также усложнит управляющий модуль.

9. Схожие работы

Для автоматического поиска известных уязвимостей могут применяться инструменты – сканеры уязвимостей (Nikto[4], Nessus[5]). Такие инструменты прежде всего сканируют порты и определяют использующие их сервисы. Затем проводится проверка обнаруженных сервисов при помощи ранее составленной базы известных уязвимостей. Такие инструменты весьма полезны при проверке безопасности тестируемых приложений. Однако, в отличие от Avalanche, с их помощью нельзя обнаружить ранее не известную ошибку. По сути, эти инструменты осуществляют тестирование, но не поиск непосредственных ошибок.

Как статический, так и динамический анализ широко применяется для поиска уязвимостей безопасности. В работе [6] используется динамический анализ для тестирования Web-приложений, написанных на языке Python. При этом так же, как и в Avalanche, собирается трасса в виде набора инструкций байт кода. Однако, отличие работы [6] заключается в том, что динамический анализ используется не для расширения покрытия тестируемого приложения, а для более точного составления тестовых данных (например, если какой-то из параметров HTTP-запроса используется только при составлении запроса к базе данных, то это признак того, что данный параметр бесполезно тестировать на внедрения скрипта и т. п.). Кроме того, основное внимание уделяется уязвимостям безопасности, что ставит Avalanche и работу [6] в несколько различные категории.

В работе [7] используется инструментация байт кода Java для предотвращения уязвимостей безопасности в Java Web приложениях. При этом список методов, которые могут быть источниками потенциально опасных данных, является одним из параметров конфигурации, предоставляемых пользователем. Аналогично предоставляется список методов, использование потенциально опасных данных в качестве параметров которых приводит к возникновению ошибки. Инструментирующий код отслеживает распространение потенциально опасных данных в анализируемой программе и вызывает исключение в том случае, если программа пытается использовать потенциально опасные данные в качестве параметров одного из методов списка. Этот подход, хоть и схож в плане инструментации и отслеживанию распространения потенциально опасных данных в программе, не

осуществляет целенаправленный поиск ошибок в программе, а лишь защищает от их возможных последствий.

10. Заключение

Мы рассмотрели поддержку получения потенциально опасных данных через сокеты для анализа сетевых приложений при помощи Avalanche. Основной идеей, реализация которой позволила этого добиться, является концепция замены получаемых приложением “нормальных” входных данных на специально подобранные значения. Несмотря на то, что поддержка анализа реализована пока только для клиентских приложений, есть основания рассчитывать на то, что концепция замены данных является универсальной, т. е. пригодной как для анализа клиентских, так и для анализа серверных приложений. Если удастся её реализовать для анализа серверных приложений, это, вероятно, позволит обнаруживать ещё более ценные ошибки, поскольку, в общем случае, ошибки в программе-сервере являются более критичными, чем ошибки в программе-клиенте.

Список литературы

[1] И. Исаев, Д. Сидоров. "Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах". Программирование, №4 2010.

[2] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In PLDI, 2007.

[3] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In CAV 2007, LNCS 4590, pages 519–531.

[4] Nikto Web Scanner. http://cirt.net/nikto2

[5] Nessus, The Network Vulnerability Scanner. http://www.nessus.org/nessus/

[6] D. Kozlov, A. Petukhov, "Detecting Security Vulnerabilities in Web Applications Using Dynamic Analysis with Penetration Testing". OWASP Application Security Conference 19-22 May 2008, Ghent, Belgium, 2008.

[7] V. Haldar, D. Chandra, M. Franz. "Dynamic Taint Propagation for Java". In: Proceedings of the 21st Annual Computer Security Applications Conference (2005).

Avalanche: Using dynamic analysis for automatic defect detection in programs based on network sockets

Ildar Isaev, Denis Sidorov, Alexander Gerasimov, Mikhail Ermakov
iisaev@ispras.ru, sidorov@ispras.ru, agerasimov@ispras.ru, mermakov@ispras.ru

Annotation. This article describes an attempt to modify and use Avalanche tool for dynamic analysis and testing of programs reading input data from network sockets. The technique of received data substitution is introduced, and it's Valgrind based implementation is described. An overview of interception and handling of network system calls is provided. The results of analysis of open-source network applications are included, as well as a list of newly discovered defects.

Keywords: dynamic analysis; defect detection; testing.

Динамическое профилирование программы для системы LLVM

А.И. Аветисян, К. Ю. Долгорукова; Ш. Ф. Курмангалеев,
arut@ispras.ru, unerkannt@ispras.ru; kursh@ispras.ru

Аннотация. При построении системы компиляции для языков общего назначения, учитывающей специфические особенности целевой аппаратуры и наиболее вероятный сценарий использования, необходимо применять методы динамической и адаптивной оптимизации. Исследование таких методов удобно проводить в компиляторной инфраструктуре LLVM. Тем не менее, в настоящий момент LLVM не поддерживает динамический сбор профиля и перекомпиляцию, а также содержит лишь одно преобразование, использующее данные профиля. В рамках данной работы, для LLVM была предложена и реализована система сбора профиля аппаратных прерываний и алгоритм, корректирующий переоценку профиля, а также несколько оптимизирующих преобразований с учетом профиля. Выполнена интеграция сбора профиля и динамического компилятора LLVM, что позволило сохранять качество программ при их переносе на другую архитектуру.

Ключевые слова: профилирование; динамическая компиляция;

1. Введение

В современных компиляторах для языков общего назначения (Си/Си++) основными видами выполняемой оптимизации является статическая оптимизация и оптимизация с использованием профиля программы. При статической оптимизации генерируется одна версия объектного кода программы, которая оптимизирована для некоторой “средней” машины данной процессорной архитектуры, а для учета специфических особенностей конкретной аппаратуры необходимо создавать и поддерживать несколько бинарных версий. Например, использование данных профиля позволит повысить качество планирования.

При оптимизации с учетом профиля программы производится сбор профилей на заданном множестве наборов входных данных и учет полученной статистики, при этом ответственность за подбор входных данных ложится на программиста. Отметим, что статистика на пользовательских наборах данных может значительно отличаться, что в некоторых случаях приводит к замедлению программы. Кроме того, этот подход связан со значительными накладными расходами на сбор профилей и подбор параметров компилятора.

Предлагаемым решением является применение методов динамической (JIT-компиляции) и адаптивной оптимизации в компиляторах для Си/Си++. Динамическая оптимизация во время работы программы имеет то преимущество, что программа оптимизируется на конкретном наборе входных данных для данного конкретного запуска. Собранная статистика используется только для оптимизации данного запуска, а разные запуски программы могут приводить к различным оптимизациям.

Для реализации динамических оптимизаций в компиляторе необходимо распространение программы в объектных файлах, содержащих внутреннее представление, сохраняющее информацию высокого уровня и позволяющее проводить динамический мониторинг, профилирование и оптимизацию программы. Это позволяет сократить затраты на распространение и поддержку программы (достаточно поддерживать одну версию программы, при сборке которой применялись лишь машинно-независимые оптимизации). Окончательная специализация будет происходить автоматически на машине пользователя. При этом значительно упрощается процесс разработки без потерь в производительности получаемой программы, так как не требуется выполнять подбор «хорошего» набора входных данных для сбора статистики.

В качестве основы для построения системы динамической оптимизации программ на языках общего назначения удобно выбрать LLVM [1] – компиляторную инфраструктуру с открытыми исходными кодами на языке Си++. В рамках этого проекта представлены: статический компилятор, компоновщик, виртуальная машина, JIT-компилятор. Функционирование системы обеспечивается единым внутренним представлением, которое может быть представлено в текстовом виде, в виде структур данных в оперативной памяти, а также в двоичном виде как бит-код. Этот бит-код может быть сохранен в промежуточных объектных файлах для дальнейшей оптимизации, в том числе динамической. При этом возможно использовать все предоставляемые LLVM возможности по обработке внутреннего представления (включая различные анализы, трансформации и т.п.). Поэтому инфраструктура LLVM предоставляет удобную базу для исследований по динамической оптимизации программ. Данная работа является развитием идей, предложенных в [2].

К сожалению, в LLVM возможности работы с профилем реализованы в минимальном объеме: из существующих оптимизаций профиль используется только при переупорядочивании базовых блоков, а работа с данными профиля ведется лишь при статической компиляции в виде обычной двухпроходной схемы компиляции. Поэтому необходимой задачей является разработка такой схемы профилирования, которая, во-первых, может применяться динамически, а во-вторых, достаточно эффективна. Мы предложили схему профилирования на основе частичного профилирования, или сэмплинга (sampling). Идея этого метода состоит в записи лишь части данных (проб), выбрав соотношение между записываемыми и пропускаемыми данными

таким образом, чтобы, с одной стороны, профилирование выполнялось с малыми затратами (замедление программы не превосходило бы нескольких процентов), а с другой стороны, полученные данные достаточно полно отражали бы реальную картину поведения программы. Предложенная схема профилирования реализована на основе утилиты Oprofile, кроме того, реализован алгоритм «подгонки» полученного динамического профиля таким образом, чтобы в результате данные профилирования могли быть использованы напрямую оптимизациями в компиляторе LLVM. Мы применили полученные данные в оптимизации перемещения базовых блоков и смогли получить выигрыш по сравнению с программой, собранной без учета профиля.

2. Динамическое профилирование программы для системы LLVM

2.1. Изменения, внесенные в систему сборки LLVM

Так как в LLVM не предусмотрены средства прозрачного, автоматического получения бит-кода с учетом зависимостей между модулями. А также отсутствует поддержка динамического связывания модулей с бит-кодом. Для программ система сборки которых основана на использовании утилит configure и make, была предложена следующая схема двухэтапной компиляции (см.рис 1).

На первом этапе происходит генерация установочного пакета, содержащего в себе модули бит-кода и скрипты автоматического развертывания. Вместо использования оригинальной утилиты configure предлагается использовать специальную обертку configure-proxy, осуществляющей необходимые подстановки и вызывающую оригинальную утилиту, результатом работы является скрипт make.sh, принцип работы которого аналогичен. Для компиляторов реализованы обертки, основанные на том же подходе. Помимо этого необходимые изменения внесены в компиляторы переднего плана, и линкер, эти изменения позволяют отследить зависимости между отдельными модулями программы. После окончания компиляции программы с помощью скриптов пост-обработки происходит, создание инсталляционного пакета на основе сгенерированных ранее зависимостей. Инсталляционный пакет содержит файлы с бит-кодом. Файлы, помеченные как зависимости на этапе постобработки, и скрипты компиляции и установки. На втором этапе во время установки программы существует две альтернативы. Первый вариант установки, это статическая компиляция, второй использование динамической компиляции.

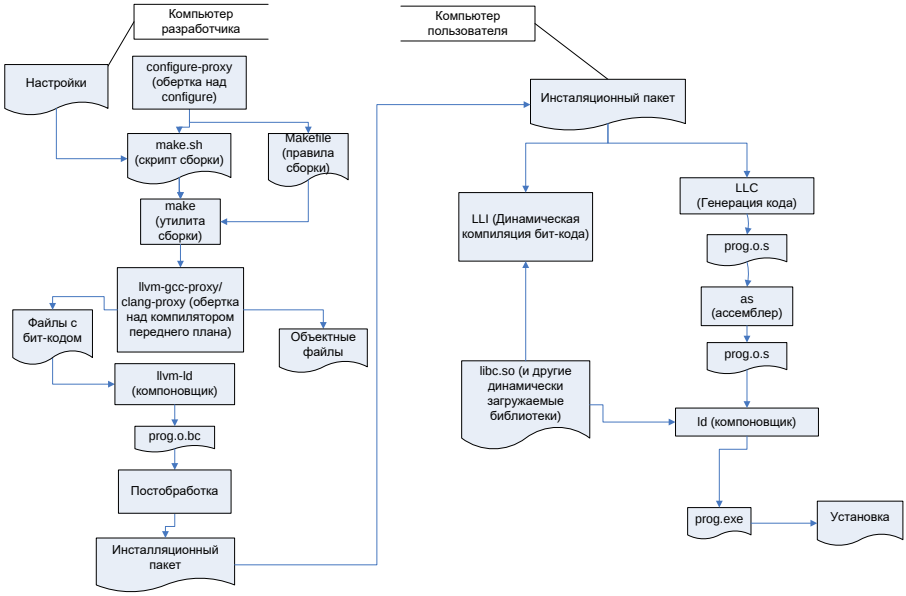


Рис. 1. Схема двухэтапной компиляции.

В обоих случаях все происходит прозрачно для пользователя. Описанный подход позволяет при компиляции учитывать специфику конкретной архитектуры и генерировать более оптимальный код. А так же позволяет использовать динамическую компиляцию и полностью автоматизировать процесс получения бит-кода и определения как статических, так и динамических зависимостей между модулями программы, как содержащими бит-код, так и бинарный код, для программ состоящих более чем из одного модуля, и написанных на языках общего назначения c/c++.

2.2. Существующие подходы к сбору профиля

Существует два способа получить статистику во время исполнения программы: инструментировать ее, - то есть вставлять счетчики в код при генерации или еще в промежуточное представление программы, - либо собирать выборочным профилировщиком аппаратных прерываний. В первом случае код увеличивается в размерах и сильно замедляется работа программы. Как показали результаты тестирования на наборе тестов aburto, при инструментировании ребер между базовыми блоками, в худшем случае происходит замедление в 2-4 раза. Эвристики инструментирования помогают опустить эту разницу лишь до 15-30 процентов [3]. Во втором же случае программа замедляется всего на 2-8 процентов [4], поэтому в этой работе будет рассмотрен именно этот метод - профилирование сэмплингом.

При профилировании компилируемых на лету программ возникает следующая проблема: профилировщик видит только запускаемые программы и таблицу символов этих программ, а символы генерируемого кода ему не видны, и информация о генерированном коде помещается в анонимное пространство имен. Для того, чтобы знать, какая функция исполняется JIT-компилятором, нужно задать способ отображения символов на виртуальные адреса. В виртуальных машинах Java к сэмплингу применяется следующий подход: компилятор записывает в файл виртуальные адреса начала и конца динамически загружаемых методов или участков кода (как правило, это циклы) при их загрузке или выгрузке. Для записи информации о символах используется библиотечный интерфейс JIT-агентов, как правило, включающий в себя функции оповещении о загрузке и выгрузке методов [5].

После получения сэмплов профилировщик сопоставляет полученные в виде пар (вирт. адрес, значение счетчика) выборки с файлами, записанными агентами, и распределяет статистику по символам, записывая ее в файл, который потом передается на считывание JIT-компилятору. Но данный подход не слишком удобен для языков общего назначения, где логика динамически загружаемых классов неприемлема. Но, тем не менее, LLVM предусматривает возможность раздельной компиляции и исполнения функций, поэтому механизм работы с агентом может быть использован. Но в этой работе был использован другой подход. За основу профилировщика был взят Oprofile 0.9.6, в котором механизм записи сэмплов в файл был заменен общением с JIT-компилятором через FIFO файл.

2.3. Описание предлагаемого подхода к сбору профиля

Поскольку предлагаемая система использует в качестве профилировщика утилиту OProfile, опишем принцип ее работы. Oprofile состоит из двух частей – сборщика информации, реализованного как модуль к ядру Linux, и клиентской части, обрабатывающей полученную информацию. Сборщик информации считает количество раз срабатывания некоторого счетчика (например, таймера или события промаха кэша), при переполнении этого счетчика происходит сброс информации в буфер сборщика и далее на жесткий диск. Клиентская часть обрабатывает записанную информацию, строит необходимые графы вызовов и т.п.

Схема работы OProfile была применена для реализации динамического профилирования в LLVM см. рис.2. Система состоит из следующих частей: JIT-компилятор считывает из файла бит-код, компилирует, запускает его и инициирует передачу статистики; модуль ядра Linux собирает статистику о счетчиках на машине и сбрасывает ее в буфер; демон-профилировщик считывает данные из буфера, распознает и записывает их в FIFO-каналы.

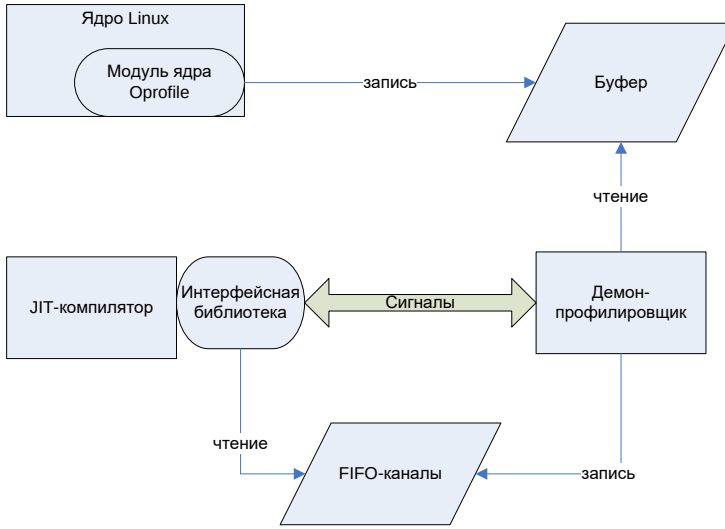


Рис. 2. Взаимодействие компонентов системы.

Демон является отдельной программой, считывающей данные из буфера, созданного модулем OProfile. Основанный на «родном» демоне Oprofile, он обрабатывает только данные, необходимые для Just-In-Time компилятора. Прежде всего, во входных параметрах, помимо тех, что есть у демона OProfile, он принимает имя образа запущенного JIT-компилятора и идентификатор или несколько идентификаторов процессов, в которых будет работать профилируемый код. Так как код генерируется и исполняется самим компилятором, он не виден системе и не может быть идентифицирован, поэтому помещается в анонимное пространство имен. Демон выбирает помеченные как анонимные данные для конкретных процессов; когда к нему приходит сигнал с запросом сбросить собранную информацию, он записывает ее в канал и продолжает работу до получения сигнала о завершении работы. Компилятор, производящий оптимизации во время исполнения (или Just-In-time compiler), представляет собой оптимизирующий компилятор бит-кода LLVM, который был сгенерирован компилятором переднего плана LLVM-GCC или Clang с языка Си или Си++. JIT-компилятор может, как интерпретировать код, так и компилировать с различными уровнями оптимизации. Нами была выполнена реализация библиотеки, осуществляющей интерфейс между демоном-профилировщиком и компилятором. В отличие от виртуальных машин Java, библиотека не пишет информацию о сгенерированных символах в файлы и не перекладывает ответственность за отображение символов на адреса утилитам Oprofile, а общается с демоном посредством сигналов и получает данные от него из

FIFO-каналов, формируя базу данных профиля. Отображение на виртуальные адреса производится в процессе работы JIT-компилятора.

3. Переоценка профиля

У полученной выборки погрешность будет довольно велика ввиду задержки срабатывания сигнала и записи в счетчик. Чтобы получить наиболее корректный профиль программы, мы должны убедиться, что, представив граф потока управления сетью, получим ее такой, что на ней будет выполняться теорема о сохранении потока. Для этого мы применили к профилю программы алгоритм переоценки профиля [6] [7]:

1. По имеющемуся графу потока управления и профилю строится расчетная сеть.
2. Далее на этой сети ищется максимальный поток минимальной стоимости.
3. В соответствии с функцией максимального потока распределяются новые веса в профиле.

Алгоритм принимает на вход структурированный по базовым блокам профиль, а также информацию статического анализа циклов: она будет необходима для предварительной оценки весов ребер графа потока управления функции. Так как профиль собирался для инструкций, суммарное количество сэмплов на базовом блоке не есть частота исполнения блока, поэтому оцениваем последнюю как среднее по инструкциям:

$$BB_{count} = \frac{\sum_{i=1}^{N_{statements}} IR.count_i}{N_{statements}}$$

где $N_{statements}$ $N_{statements}$ - число инструкций в базовом блоке, $IR.count_i$ $IR.count_i$ - число сэмплов для i-й инструкции. Для облегчения работы алгоритму также предварительно удаляются все недостижимые пути графа.

3.1. Построение расчетной сети

Необходимость построения дополнительной расчетной сети обусловлена особенностями применяемого к ней впоследствии алгоритма поиска максимального потока минимальной стоимости:

1. в сети не должно быть циклов длины, меньшей 3,
2. должны присутствовать исток и сток.

3.2. Построение функции максимального потока минимальной стоимости

Для построения функции максимального потока был использован алгоритм Эдмондса-Карпа - одна из реализаций алгоритма Форда-Фалкерсона, уточняющая, что при построении увеличивающего пути используется алгоритм поиска в ширину. Поэтому он является конечным для всех действительных значений пропускной способности ребер, в отличие от самого алгоритма Форда-Фалкерсона, который может работать бесконечно при произвольном выборе пути на каждом шаге [8]. После построения максимального потока строится остаточная сеть, в которой по алгоритму Клейна удаляются циклы отрицательной стоимости [9].

3.3. Пересчет профиля

По сети максимального потока минимальной стоимости однозначно восстанавливается исходный граф потока управления: по ребрам, построенным в результате преобразования вершин, мы восстанавливаем профиль вершин. Ребра, соответствующие исходному графу, не удаляются алгоритмом, т.к. им была назначена бесконечная максимальная пропускная способность, значит, тоже могут быть восстановлены. Теперь профиль удовлетворяет условию сохранения потока, т.к. лишний поток был «слит» в источник или в сток с помощью балансировки в процессе работы алгоритма Эдмондса-Карпа. Таким образом, новыми значениями счетчиков сэмплов станут значения потоков на ребрах сети максимального потока.

3.4. Сохранение собранного профиля в формате LLVM

После пересчета, у нас есть нормализованный профиль, его можно сохранить на диск, но нельзя применить непосредственно к имеющемуся бит-коду. Несоответствие обусловлено тем, что перед запуском кодогенерации LLVM запускает несколько проходов изменяющих машинно-независимое внутреннее представление. Таким образом, для обеспечения возможности использования профиля собранного сэмплингом, следует выполнить все оптимизации, запускаемые над машинно-независимым представлением, запустить требуемые оптимизации использующие профиль и сохранить оптимизированный бит-код на диск. Описанный подход был реализован нами в текущей работе. Но в данный момент имеет некоторые ограничения, связанные тем, что если применяется динамическая компоновка нескольких модулей с бит-кодом, то профиль сохраняется для объединенного представления модулей в памяти, в будущем мы будем сохранять профиль отдельно для каждого модуля.

4. Оптимизации, использующие профиль

Имея профиль уже после работы одной или нескольких функций, мы можем точно определить не только степень «горячести» самой функции - то есть, частоту исполнения по сравнению с другими, - но и также относительную «горячесть» базовых блоков, узкие места внутри функции, которые можно оптимизировать прямо во время работы программы. К сожалению в настоящее время LLVM 2.9 имеет поддержку только одной оптимизации, использующей информацию о собранном профиле (Basic Block Placement). Далее описываются некоторые оптимизации применение которых с учетом собранного профиля может привести увеличению быстродействия компилируемых программ.

4.1. Перемещение базовых блоков

Основная идея этой оптимизации заключается в том, чтобы расположить код часто исполняемых базовых блоков близко друг к другу, сокращая тем самым время вычисления адресов переходов между блоками, а в большинстве случаев позволяя даже исполнять «горячий код», состоящий из цепочки блоков, последовательно.

4.2. Разбиение блоков

Под разбиением понимается удвоение часто исполняемых базовых блоков графа потока управления, имеющих более одного исходящего ребра и более одного входящего. Суть алгоритма показана на рис. 3.

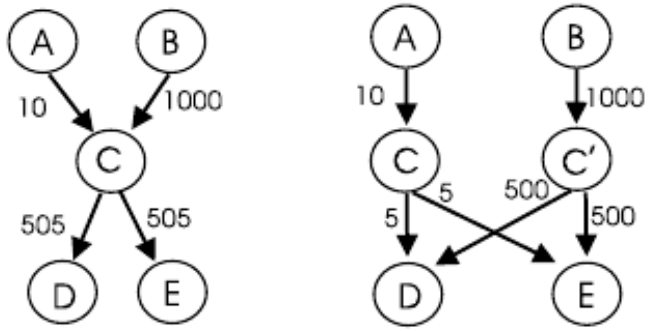


Рис. 3 Разбиение соединенных базовых блоков: слева - до разбиения, справа - после.

Данное преобразование не является оптимизирующим само по себе, но позволяет оптимизациям, основанным на анализе потока данных, - таким, как удаление загрузок, удаление границ массивов, замещение на стеке и пр., - работать более эффективно.

4.3. Встраивание функций

Очень важным преобразованием является встраивание тел функций вместо их вызовов. Встраивание, основанное только на статических эвристиках, на практике не всегда дает оптимальный код. Так, если функция будет встроена в редко исполняемые участки кода, но определенные статическим анализатором как возможно горячие, это не принесет практически никакой пользы. Если же основывать решение о встраивании функции в конкретный блок на фактической частоте его исполнения, возможно увеличение производительности даже в сравнении со статическими компиляторами.

4.4. Развертка циклов

Имея динамический анализ, можно также разворачивать циклы, основываясь на частоте их исполнения. Например, часто исполняемые циклы разворачивать большее число раз для увеличения производительности, редко исполняемые - меньшее или же вообще не разворачивать, чтобы они занимали меньше памяти.

5. Текущие результаты

На данном этапе реализованы: инфраструктура, обеспечивающая прозрачную компиляцию в промежуточное представление LLVM, динамическое связывание модулей с промежуточным представлением во время выполнения программы, демон-профилировщик, библиотека работы с профилем, сохранение профиля в формате LLVM. Так как в LLVM JIT-компиляторе замещение на стеке пока находится в зачаточном состоянии, тестирование проходило в 2 прохода: сначала функции компилировались с уровнем оптимизации -O2 и сборкой профиля, а потом с использованием профиля компилировались заново. Результаты сравнивались с GCC 4.4.3 с опциями -O2 и стандартным интерпретатором lli с опциями -O2, тестирование проводилось на машине Intel Core2 Duo, E8500 3.00GHz, на ОС Linux Ubuntu 10.04 LTS. Результаты тестирования приведены в табл.1.

Тест	**Без сбора профиля**	**сбор профиля**	**с учетом профиля**	**gcc**	**Единица измерения**
Whetstone	3511,6	3219	3308	3594,8	MWIPS
Tfftdp	122,311	120,221	126,719	130,239	VAX_FFTs
Heapsort	4458,57	3585,35	4458,57	2907,72	MIPS

Тест		Без сбора профиля	сбор профиля	с учетом профиля	gcc	Единица измерения
Nsieve		2968	2667,2	3022,9	2996	MIPS
Flops		880,362	854,833	1080,8889	1180,5825	MFLOPS
Fibonacci		0,72	0,97	0,72	0,86	secs
Matrix Multi-plication	normal algorithm	0,48	0,52	0,49	0,48	secs
	temporary variable in loop	0,49	0,5	0,47	0,49	secs
	unrolled inner loop	0,49	0,51	0,49	0,47	secs
	pointers used to access matrices	0,48	0,49	0,48	0,48	secs
	transposed b matrix	0,14	0,14	0,12	0,14	secs
	interchanged inner loops	0,14	0,18	0,13	0,13	secs
Scimark		1042	920,21	1046,22	989,71	Composite Score

Табл. 1. Результаты тестирования.

Помимо этого, были проведены тесты по сбору профиля с выгрузкой его на диск, последующим применением к файлу бит-кода на примере программы sqlite и оптимизацией “Basic Block Placement”, что привело к ускорению работы программы на тестовом наборе данных на ~3%. Для проверки корректности проводилась, рекомпиляция бит-кода, без оптимизации использующей профиль, в этом случае мы наблюдали замедление порядка 0,5%-1%.

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] А. Белеванцев, Д. Журихин, Д. Мельник. Компиляция программ для современных архитектур. Труды Института системного программирования РАН, том 17, 2009 г. стр. 31-50.
[3] Omri Traub, Stuart Schechter, Michael D. Smith. Ephemeral Instrumentation for Lightweight Program Profiling.— 2000. www.eecs.harvard.edu/hube/publications/muck.pdf. Date retrieved: October 7, 2011.
[4] OProfile official website. http://oprofile.sourceforge.net.
[5] Java Virtual Machine Profling Interface documentation. http://download.oracle.com/javase/1.4.2/docs/guide/jvmpi/jvmpi.html. Date retrieved: October 7, 2011.
[6] Vinodha Ramasamy, Robert Hundt, Dehao Chen, Wenguang Chen. Feedback-Directed Optimizations in GCC with Estimated Edge Profiles from Hardware Event Proceedings of GCC Summit 2008. http://research.google.com/pubs/pub36576.html. Date retrieved: August 18, 2011.
[7] R. Levin, I. Newman, G. Haber. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. Proceedings of the 3rd international conference on High performance embedded architectures and compilers.— HiPEAC'08.— Berlin, Heidelberg: Springer-Verlag, 2008.— Pp. 291–304.
[8] Zwick, U. The smallest networks on which the ford-fulkerson maximum flow procedure may fail to terminate. Theoretical Computer Science.— 1995.— Vol. 148.
[9] Levin, R. — Complementing Incomplete Edge Profile by applying Minimum Cost Circulation Algorithms.— Master's thesis, University of Haifa, 2007.—Aug. http://www.cs.technion.ac.il/~royl/MscThesis_Final_Version_Submission.pdf. 38 pages

Dynamic profile collection for LLVM

A.I. Avetisyan, K.U. Dolgorukova; Sh. F. Kurmangaleev
arut@ispras.ru, unerkannt@ispras.ru; kursh@ispras.ru

Abstract. One has to harness dynamic and adaptive recompilation methods when designing the system for general-purpose languages compilation which takes into account the specific factors of target hardware and the most likely way of usage. It is favorable to research those methods in the LLVM infrastructure environment. Unfortunately, LLVM does not support dynamic profile collection and recompilation at the moment, as well as contains only one profile-based optimization. Within the introduced work, we proposed and implemented the system for hardware interruptions profile collection and the algorithm to correct profile estimations, and, apart from that, the number of profile-based optimizations. We integrated dynamic profile collection into the LLVM dynamic compiler resulting in constant program quality regardless of hardware architecture.

Keywords: profiling; just in time;

Методы точного измерения времени выполнения гнезд циклов при анализе *JavaMPI*-программ в среде *ParJava*

А.И. Аветисян, М.С. Акопян, С.С. Гайсарян [1],
arut@ispras.ru, manuk@ispras.ru, ssg@ispras.ru

Аннотация. В работе рассматриваются методы оценки времени выполнения модели параллельной программы на инструментальном компьютере, которые позволяют достаточно точного предсказывать время реального выполнения параллельной программы на заданном параллельном вычислительном комплексе. Модель разработана для параллельных SPMD программ с явным обменом сообщениями, написанных на языке Java с обращениями к библиотеке MPI, и включена в состав среды ParJava. В модели выделяются определенные виды циклов (однородные, редуцируемые) и производится их оценка на узле целевой вычислительной системы (высокопроизводительного кластера). Это позволяет не только уменьшить погрешность предсказания, но и ускорить время интерпретации модели на инструментальном компьютере.

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

1. Введение

Среда *ParJava* [1] предоставляет прикладному программисту набор инструментов, поддерживающих разработку прикладных параллельных программ для вычислительных систем с распределенной памятью (высокопроизводительных кластеров) на языке *Java*, расширенном стандартной библиотекой передачи сообщений *MPI* [2, 3]. Многие инструменты среды *ParJava* могут выполняться на инструментальном компьютере, используя вместо разрабатываемой параллельной программы ее модель [4], которая может интерпретироваться как на целевой вычислительной системе, так и на инструментальном компьютере. Модель параллельной программы адекватно отражает ее поведение на кластере,

1 НИР поддержана грантами РФФИ 09-07-00382-а и ФЦП «Научные и научно-педагогические кадры инновационной России на 2009-2013 годы», ГК №П728 от 12 августа 2009 года

позволяя исследовать динамические характеристики (в частности, время выполнения) разрабатываемой программы, или ее частей.

Модель *SPMD*-программы представляет собой совокупность моделей всех классов, входящих в состав моделируемой программы. Модель каждого класса – это множество моделей всех методов этого класса; кроме того, в модель класса включается модель еще одного дополнительного метода, описывающего поля класса, его статические переменные, а также инициализацию полей и статических переменных. Модель метода (функции) представляется в виде пары ⟨модель потока управления, модель вычислений⟩. Модель потока управления представляет собой модифицированное дерево управления метода, полученное из графа потока управления путем выделения и сворачивания областей. Внутренние вершины модели соответствуют операторам управления моделируемой программы, а листовые представляют собой дескрипторы ее базовых блоков. В качестве базовых блоков рассматриваются не только линейные последовательности вычислений (вычислительные базовые блоки), но также и вызовы библиотечных функций, вызовы пользовательских методов и функций и вызовы коммуникационных функций. Для обеспечения возможности частичной интерпретации модели и интерпретации модели по частям введено понятие редуцированного блока, который представляет значения динамических атрибутов уже проинтерпретированных частей метода. Дескриптор каждого базового блока содержит оценку времени выполнения данного базового блока, определенную до начала интерпретации на одном узле целевой вычислительной системы. Модель вычислений содержит байт-код каждого базового блока моделируемого метода, что позволяет выполнять (интерпретировать) ее на *JavaVM*. По окончании выполнения очередного базового блока модели вычислений вызывается интерпретатор среды *ParJava*, который, с помощью модели потока управления, определяет очередной базовый блок, который следует интерпретировать.

В данной работе рассматриваются проблемы, связанные с прогнозом времени выполнения параллельной прикладной программы и исследованием зависимости времени выполнения программы от числа узлов целевой вычислительной системы (кластера), участвующих в выполнении программы. Оценки времени выполнения программы для различных значений числа улов кластера, позволяют прикладному программисту построить график зависимости оценки времени выполнения его программы от числа задействованных узлов, оценить границы области масштабируемости программы и выбрать оптимальное число узлов кластера. Кроме того, такие оценки позволяют вручную сбалансировать обмены данными между параллельными ветвями программы за счет правильного выбора функций библиотеки *MPI* и определения оптимальных точек вызова этих функций.

Интерпретатор модели параллельной программы, работающий на инструментальном компьютере использует временные профили параллельной программы, полученные заранее на целевой вычислительной системе.

Проблемы, связанные с получением достаточно точных оценок времени выполнения операций обмена по характеристикам коммуникационной сети кластера рассмотрены в работе [5]. В данной работе рассматриваются методы оценки времени выполнения фрагментов *Java*-программы (в частности, базовых блоков), не содержащих операций обмена. Как показано в данной работе, при построении временного профиля не следует ограничиться оценкой времени выполнения базовых блоков, интерпретация может быть существенно ускорена, если оценивать и время выполнения более крупных фрагментов: отдельных итераций циклов, циклов в целом, а в некоторых случаях – и гнезд циклов.

Временной профиль моделируемой программы строится на целевой вычислительной системе и уточняется в процессе интерпретации программы. С помощью операции редукции (свертки), введенной в [4], модель преобразуется к форме, обеспечивающей ее быструю интерпретацию. Методам разбиения модели программы на достаточно крупные фрагменты, для которых возможно получить и использовать достаточно точные оценки их времени выполнения, посвящена данная статья.

В разделе 2 приводится краткое описание модели программы, методов оценки времени выполнения программы и ее фрагментов, реализованных в среде ParJava, и уточняется постановка задачи. В разделе 3 рассматривается схема укрупнения фрагментов программы, для которых возможно построить достаточно точные оценки времени их выполнения. В частности, в данном разделе рассматриваются методы получения оценок для гнезд циклов и разбираются случаи, когда этих оценок достаточно для моделирования поведения программы с требуемой точностью. В разделе 4 обсуждаются методы измерения степени разбалансированности гнезда циклов и методы оценки времени выполнения разбалансированных гнезд циклов в среде ParJava. В разделе 5 рассматриваются методы получения оценок времени выполнения параллельных циклов, требующих синхронизации. В разделе 6 рассмотренные методы применяются для оценки ускорения и границ области масштабируемости некоторых модельных и реальных программ. В частности строятся графики предсказания границ области масштабируемости для программы моделирования интенсивных атмосферных вихрей [6].

2. Вычисление оценки времени выполнения методов.

Как описано в [4], модель каждого метода *Java*-программы рекурсивно строится из моделей его фрагментов[2] – базовых блоков и их комбинаций (последовательностей, ветвлений, переключателей и циклов). Измерив значение времени выполнения каждого базового блока, можно в процессе интерпретации модели вычислять оценки времени выполнения различных комбинаций базовых блоков и более крупных фрагментов, получая в конце интерпретации метода оценку времени его выполнения. При этом время выполнения последовательности фрагментов всегда равно сумме времен выполнения составляющих фрагментов. Но если в состав фрагмента входит ветвление, то время его выполнения зависит от времени выполнения текущей ветви, т.е. определяется значениями параметров метода. Если время выполнения фрагмента не зависит от параметров метода (например, фрагмент не содержит ветвлений, или время выполнения каждой ветви примерно одинаково[3]), будем называть такой фрагмент *простым*. В терминах [4] каждый простой фрагмент может быть редуцирован.

Легко видеть, что если тело цикла является простым фрагментом, то и сам цикл является простым фрагментом и может быть редуцирован. Мы будем называть такие циклы *простыми*. Подобные соображения применимы и к соответствующим гнездам циклов, которые тоже будут называться *простыми*. Если можно доказать, что гнездо циклов является простым, можно при определении временного профиля метода измерить время выполнения такого гнезда и заменить его на редуцированный блок с соответствующим временем выполнения.

Следует отметить, что гнездо циклов, содержащее операции обмена данными через *MPI*, не может быть простым, хотя внутренние циклы, не содержащие операций обмена, могут быть простыми и допускать редукцию.

Для параллельных *MPI*-программ, написанных на одном из традиционных языков (C/C++, Fortran), модель должна строиться не над исходным кодом программы, а над ее промежуточным представлением с учетом результатов статической оптимизации, так как оптимизаторы современных компиляторов могут существенно изменить структуру программы, а следовательно, и ее модель.

Для параллельных *Java*-программ, использующих *MPI*, структура программы, выполняемой на *Java VM*, практически не отличается от структуры исходной

[2] В данной работе рассматриваются фрагменты, которые в литературе обычно называются областями или регионами, т.е. подграфы графа потока управления метода, имеющие единственный входной базовый блок, доминирующий над всеми остальными блоками фрагмента.

[3] Времена выполнения ветвей различаются на p%, где p – достаточно мало (например, $p = 10$).

программы: статическая оптимизация, как правило, производится только внутри базовых блоков и не влияет на структуру программы. В процессе выполнения *Java*-программы на *Java VM* определяются наиболее часто выполняемые фрагменты программы (*«горячие» методы*) и для них выполняется динамическая компиляция с агрессивной оптимизацией (разработчики динамических компиляторов считают, что такие фрагменты составляют менее 20% всей программы [7]). Таким образом, большая часть методов программы продолжает интерпретироваться на *Java VM*, причем такие методы обычно удовлетворяют условиям редукции и при интерпретации модели вносят некоторый постоянный вклад во время выполнения соответствующих фрагментов (время интерпретации таких методов в среде *ParJava* сравнимо с временем интерпретации одного базового блока).

Что касается «горячих» методов, то для ускорения интерпретации их моделей с сохранением точности оценок времени выполнения необходимо уметь выделять как можно более крупные фрагменты, в пределах которых бывает сосредоточена большая часть структурных изменений кода, связанных с оптимизацией, и при профилировании измерять время выполнения таких фрагментов в целом, по возможности игнорируя их структуру. В параллельных программах такими крупными фрагментами являются гнезда циклов.

3. Измерение времени выполнения сбалансированных гнезд циклов.

Рассмотрим гнездо циклов **for**. Оно может быть представлено в виде дерева, корнем которого является самый внешний цикл **for**, а поддеревьями являются циклы **for** глубины вложенности 1, далее по рекурсии. Если у одного из циклов гнезда нет вложенных в него циклов, то соответствующий ему узел дерева не имеет поддеревьев и является терминальным в дереве, представляющем гнездо циклов.

В *MPI*-программах, как правило, распараллеливается один из внешних циклов гнезда (часто это бывает самый внешний цикл). В дальнейшем будут рассматриваться только распараллеливаемые подгнезда, у которых распараллеливается самый внешний цикл.

Рассмотрим цикл **for**, количество повторений которого равно *n*. Интерпретация этого цикла на интерпретаторе среды *ParJava* (либо на *JavaVM*) позволит измерить:

(1) время выполнения цикла T_{Loop};

(2) время выполнения тела цикла на *i*-ой итерации $T^{\mathrm{i}}_{\mathrm{LoopBody}}$.

Сравнение $T^{\mathrm{i}}_{\mathrm{LoopBody}}$ со средним временем выполнения тела цикла $T^{\mathrm{mean}}_{\mathrm{LoopBody}} = \frac{T_{\mathrm{Loop}}}{\mathrm{n}}$ позволит определить степень сбалансированности цикла $D = \max_i \left| T^{\mathrm{i}}_{\mathrm{LoopBody}} - T^{mean}_{\mathrm{LoopBody}} \right|$. Будем говорить, что рассматриваемый цикл *сбалансирован*, если его степень сбалансированности *D* не превышает $\alpha \cdot T^{\mathrm{mean}}_{\mathrm{LoopBody}}$. (поскольку допустимая погрешность оценки времени выполнения фрагментов программы порядка 10% [1], можно принять α = 0,1). В противном случае цикл будет считаться *разбалансированным*.

Поскольку оптимизирующие преобразования цикла (такие, как вынесение из цикла вычислений, инвариантных относительно цикла, или минимизация числа индуктивных переменных) не влияют на его степень сбалансированности *D*, измерение *D* можно выполнять для исходного кода программы при его интерпретации на интерпретаторе среды *ParJava* , либо на *JavaVM*.

Гнездо циклов будем называть сбалансированным, если каждый цикл, входящий в его состав, сбалансирован. Анализ сбалансированного гнезда циклов выполняется, начиная с самых внутренних («листовых») циклов. Для каждого такого цикла измеряется время его выполнения, после чего указанный цикл редуцируется. Поэтому на каждом этапе анализа рассматривается цикл, у которого нет вложенных циклов (после редукции каждый вложенный цикл заменяется «эквивалентным» редуцированным базовым блоком). Если сбалансированы как сам цикл, так и все подциклы, входящие в его тело, то при снятии временного профиля получается оценка времени выполнения всего цикла, а оценка времени выполнения отдельной итерации цикла определяется делением оценки времени выполнения цикла на число его итераций.

Рассмотрим случай сбалансированного цикла, число итераций которого определяется во время выполнения программы (например, в объемлющем цикле). В этом случае нужно получить оценку времени выполнения цикла для достаточно большого числа итераций, потом получить оценку времени выполнения одной итерации цикла, причем последняя будет использоваться для вычисления оценки времени выполнения рассматриваемого цикла на каждой итерации объемлющего цикла. Аналогичный метод применим и к вычислению оценок времени выполнения циклов, распределяемых среди узлов вычислительной системы для параллельного выполнения. Результаты численных расчетов подтвердили правомерность описанного подхода к оценке времени выполнения сбалансированных циклов.

4. Измерение степени разбалансированности гнезда циклов и времени выполнения разбалансированных гнезд циклов в среде ParJava.

Рассмотрим цикл вида:

```
for(i = 0; i ≤ n; i++) {
      {fr1};
      if(c(i)) {fr2};
      else {fr3};
      {fr4}
}
```

Интерпретация фрагментов **{fr1}**, **{fr2}**, **{fr3}** и **{fr4}** и условия **c(i)** на интерпретаторе среды *ParJava* (либо на *JavaVM*) позволяет получить предварительные оценки времени их выполнения: T_{fr1}, T_{fr2}, T_{fr3}, T_{fr4}, $T_{c(i)}$. При этом оценка времени выполнения тела цикла может быть вычислена по формуле $T_{LoopBody} = T_{fr1} + T_{c(i)} + T_{Cond} + T_{fr4}$, где T_{Cond} определяется следующим образом: на интерпретаторе оцениваются частоты выполнения ветвей условного оператора F_{fr2} и F_{fr3}, после чего T_{Cond} определяется по формуле

$$T_{Cond} = \frac{F_{fr2} \cdot T_{fr2} + F_{fr3} \cdot T_{fr3}}{F_{fr2} + F_{fr3}}$$

Если фрагменты **{fr1}**, **{fr2}**, **{fr3}** и **{fr4}** содержат ветвления, оценки времена их выполнения должны быть получены заранее, с помощью описанного метода.

Для определения степени разбалансированности цикла (или гнезда циклов) интерпретатор модели запускается не менее чем в P потоках (P – достаточно большое число, например $P \geq 16$), и время интерпретации гнезда циклов измеряется в каждом потоке. Пусть T_{Cond}^{p} – оценка T_{Cond} в p-ом потоке ($1 \leq p \leq P$). Тогда степень разбалансированности вычисляется по формуле $D = \max_{p}(T_{Cond}^{p}) - \min_{p}(T_{Cond}^{p})$. Легко видеть, что рассмотренный метод применим для оценки разбалансированности циклов общего вида.

Ввиду того, что при вычислении времени выполнения цикла и его отдельных итераций измерение времени производится лишь в начале и конце итерации, оценка получается правильной, несмотря на то, что структура выполняемого цикла после динамической компиляции может быть оптимизирована, в результате чего она может существенно отличаться от структуры исходного цикла. Может измениться даже число итераций цикла из-за его частичной раскрутки. Интерактивный сценарий устранения обнаруженной разбалансированности гнезда циклов состоит в распределении по потокам

групп итераций цикла таким образом, чтобы времена выполнения указанных групп итераций стали близки (различались на достаточно малую величину). В случае недостаточно низкой (с точки зрения пользователя) степени разбалансированности (это выясняется с помощью интерпретатора) пользователь вручную меняет распределение и снова с помощью интерпретатора выясняет степень разбалансированности, после чего либо процесс заканчивается, либо проверяется новое распределение. Если цикл разбалансирован, оценка времени его выполнения получается как сумма оценок времен выполнения всех его итераций. При этом время измеряется только в начале и в конце итерации: интерпретации итераций не требуется, так что оптимизация не влияет на оценку.

5. Оценка времени выполнения циклов, требующих синхронизации.

В предыдущих разделах рассматривались циклы, целиком выполняемые на одном из узлов кластера, т.е. циклы, которые во время своего выполнения не требуют периодических обменов данными между узлами кластера (и связанной с такими обменами синхронизации вычислений на разных узлах). В данном разделе рассматриваются оценки, необходимые для правильного (оптимального) размещения вызовов функций библиотеки *MPI* в теле цикла, выполняемого параллельно на нескольких узлах кластера. Указанные вызовы должны быть размещены таким образом, чтобы (1) передача данных велась параллельно с расчетами, и (2) передача данных заканчивалась прежде, чем потребуется начать их обработку [8]. Таким образом, оптимизация (ее обычно называют настройкой параллельного цикла) состоит в нахождении такого взаимного расположения фрагментов тела цикла, при котором выполняются условия (1) и (2), и для ее решения необходимо знать оценки времени выполнения каждого из рассматриваемых фрагментов тела цикла. Пример такой оптимизации рассмотрен в [4].

Следует отметить, что методы, содержащие параллельные циклы, входят в число наиболее часто выполняемых методов программы и, следовательно, оптимизируются во время динамической компиляции. Значит для оптимизации (настройки) такого метода необходимо уточнить его модель, отразив в ней оптимизирующие преобразования, выполненные при его динамической компиляции. Основная трудность в том, что не все динамические компиляторы языка *Java* (в частности, [9]) предоставляют доступ к коду оптимизированной программы. Тем не менее, один из наиболее современных динамических компиляторов [7], разработанный в IBM, обеспечивает такой доступ, а тем самым и принципиальную возможность уточнения модели рассматриваемого метода после его динамической компиляции. Анализируя оптимизированный код можно построить его граф потока управления, а затем и дерево управления, лежащее в основе модели параллельной *Java*-программы среды *ParJava*. Используя эту возможность, в

среде *ParJava* строится уточненная модель тела параллельного цикла, позволяющая оценить время выполнения каждого его фрагмента и тем самым выбрать оптимальные точки расположения вызовов функций обмена данными. Такой подход является достаточно трудоемким, но он позволяет достаточно точно оценить задержки, вызванные асинхронностью выполнения частей циклов на узлах кластера, и добиться их минимизации.

Для сбалансированного цикла, различные итерации которого требуют примерно одинаковых объемов вычислений, планы вычислений на различных узлах кластера одинаковы. Время t, требуемое на обмен данными между такими узлами, складывается из времени t_{dt}, необходимого на передачу требуемого объема данных через канал связи, и *времени синхронизации* t_s, которое идет на компенсацию асинхронности работы узлов кластера (и ядер указанных узлов). Зная параметры коммуникационной сети кластера можно достаточно точно определить t_{dt}. Для времени t_s возможны лишь вероятностные оценки, так как в это время входит ожидание в случае, когда к моменту обращения к данным, оказывается, что обмен еще не завершен. Исследование детального профиля сбалансированного цикла позволяет настроить цикл, обеспечив приемлемые значения ускорения и масштабируемости.

В случае разбалансированного цикла, время вычисления различных итераций которого может отличаться на большие значения, настройка программы существенно усложняется. Но и в этом случае детальный профиль тела цикла может помочь найти методом проб приемлемое распределение данных по узлам кластера.

6. Результаты численных расчетов.

Реализация рассмотренных методов оценки времени выполнения *Java*-программ и их фрагментов позволила применять среду *ParJava* при разработке достаточно сложных параллельных программ. В данном разделе приведены оценки времени выполнения для двух программ-примеров, использовавшихся при реализации среды *ParJava*. Эти примеры позволили выявить некоторые особенности кластеров с многоядерными узлами. Чтобы показать применимость среды *ParJava* при разработке реальных прикладных программ приведены результаты ее применения при разработке прикладной программы численного решения системы уравнений, моделирующей процессы и условия генерации интенсивных атмосферных вихрей (ИАВ) в трехмерной сжимаемой атмосфере, исходя из теории мезомасштабных вихрей В. Н. Николаевского [10]. На приведенных графиках «Действительное ускорение» представляет собой ускорение программы, измеренное при выполнении программы на целевой платформе, «Прогнозированное ускорение» - ускорение, предсказанное интерпретатором ParJava на инструментальной машине, «Разность ускорений» - модуль разности между предсказанным и действительным ускорением.

Пример 1. Программа умножения матриц. На рисунке 1 представлены графики зависимости ускорения от числа процессов для программы умножения квадратных плотных матриц *A* и *B* порядка *n*. Программа выполнялась на кластере ИСП РАН, содержащем 12 узлов. Каждый узел этого кластера состоит из двух четырехъядерных процессоров Intel® Xeon® 5355, так что всего на узле 8 ядер. Каждое ядро имеет собственный кэш *L1* размером 32Kb и кэш *L2* размером 4Mb, разделяемый с соседним ядром. Каждый процесс обрабатывал *n*/*P* строк матрицы *A* (*P* – число процессов) и всю матрицу *B*, вычисляя соответствующие *n*/*P* строк матрицы-произведения *C*. На графиках представлены *действительное ускорение*, полученное при выполнении программы, и *прогнозированное ускорение*, полученное при интерпретации программы в среде *ParJava*.

Сначала каждом узле кластера было запускалось 8 процессов (по 1 процессу на каждое ядро). Оказалось, что в этом случае ускорение (нижняя пара кривых на рисунке 1) примерно в два раза меньше максимального, определяемого по формуле Амдаля (пунктирная кривая на рисунке 1). При исследовании этого вопроса выяснилось, что неудовлетворительное ускорение связано с особенностями архитектуры используемых узлов: ресурсы узлов делятся между их ядрами, и при полной загрузке борьба ядер за ресурсы приводит к разбалансированию вычислений и падению ускорения. И действительно, расчеты, в которых на каждом узле кластера запускалось по 4 процесса (половина максимально возможного количества), показали ускорение, достаточно близкое к максимальному (верхняя пара кривых на рисунке 1). Отметим, что именно такое ускорение ожидалось для такой простой программы. Оказалось, что для рассматриваемого примера прогнозиреванное ускорение в обоих случаях было настолько близко к реально полученному, что соответствующие пары кривых на графике почти слились. Поэтому на нижнем графике рисунка 1 приведены графики модуля разности между прогнозированным и реальным значениями (абсолютная погрешность прогноза). Высокая точность прогноза обусловлена тем, что исследуемое гнездо циклов является хорошо сбалансированным, а обмен данными между процессами во время выполнения программы не производится. Последовательная часть программы составляет всего 0,6%.

В рассматриваемом примере порядок матриц *n* был равен 5000, а число процессов *P* в первом случае принимало значения 8 (вычисления проводились на 1 узле), 16 (на 2 узлах), …, 48 (на 6 узлах), т.е. на каждом ядре каждого узла был запущен *MPI*-процесс. Во втором случае вычисления проводились таким образом, что на каждом узле кластера запускалось 4 процесса, т.е. число процессов *P* принимало значения 4 (вычисления проводились на 1 узле), 8 (на 2 узлах), …, 48 (на всех 12 узлах).

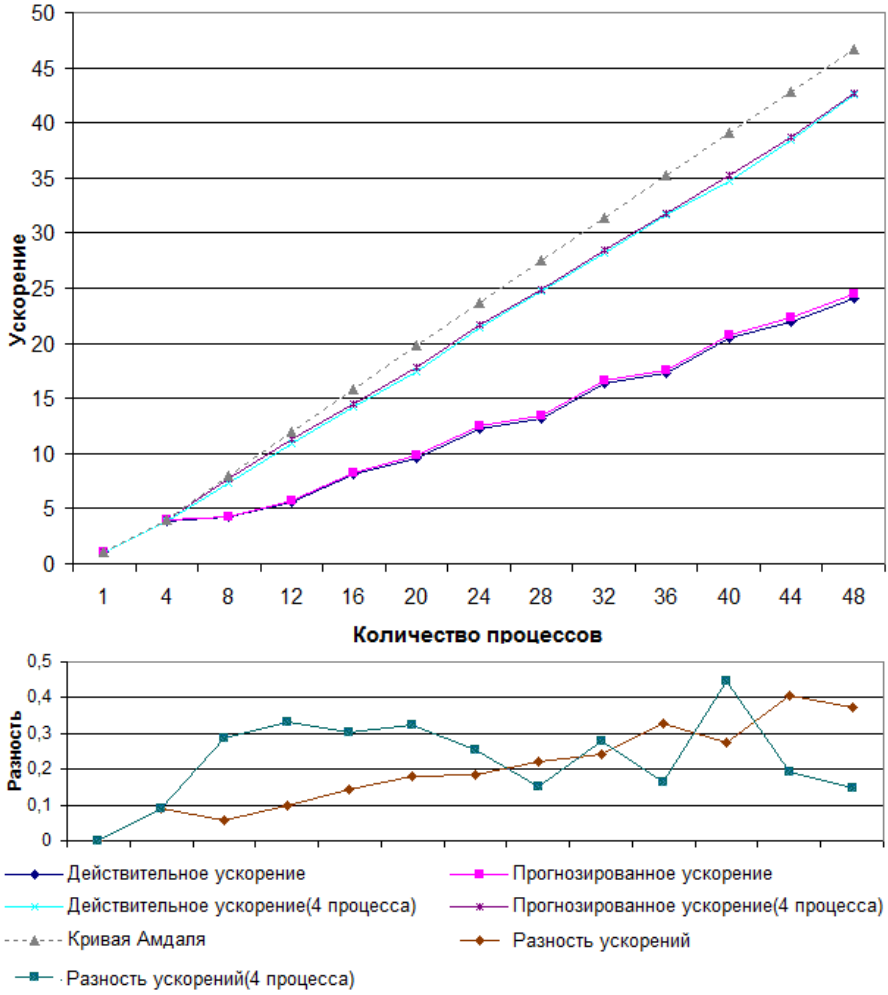


Рис. 1. Зависимость ускорения программы от числа используемых узлов кластера.

Использовалаясь 32-разрядная версия среды Java, так что каждый элемент матрицы занимал 8 байтов (тип `double`), а строка (столбец) матрицы – 40000 байтов. Запуск процессов (не потоков[4]) на каждом ядре процессора связан с

[4] В данной статье термин *поток* соответствует понятию «легковесный процесс» (англ. *thread*).

существенными накладными расходами, в частности, для интерпретации параллельной *Java*-программы на каждом ядре должна быть запущена своя *JavaVM*. В случае потоков это не так: в среде *Java* реализован системный класс *java.util.concurrent.ExecutorService*, обеспечивающий запуск требуемого числа потоков на одной *JavaVM*. Если число потоков не превышает числа ядер на узле, то каждый поток запускается на отдельном ядре.

На рисунке 2 представлены графики, на которых сравниваются ускорения параллельной *Java*-программы, запущенной на ядрах одного процессора, сначала с использованием *MPI*-процессов, а затем - *Java*-потоков. Графики показывают, что на одном узле использование потоков позволяет получить большее ускорение, чем в случае процессов.

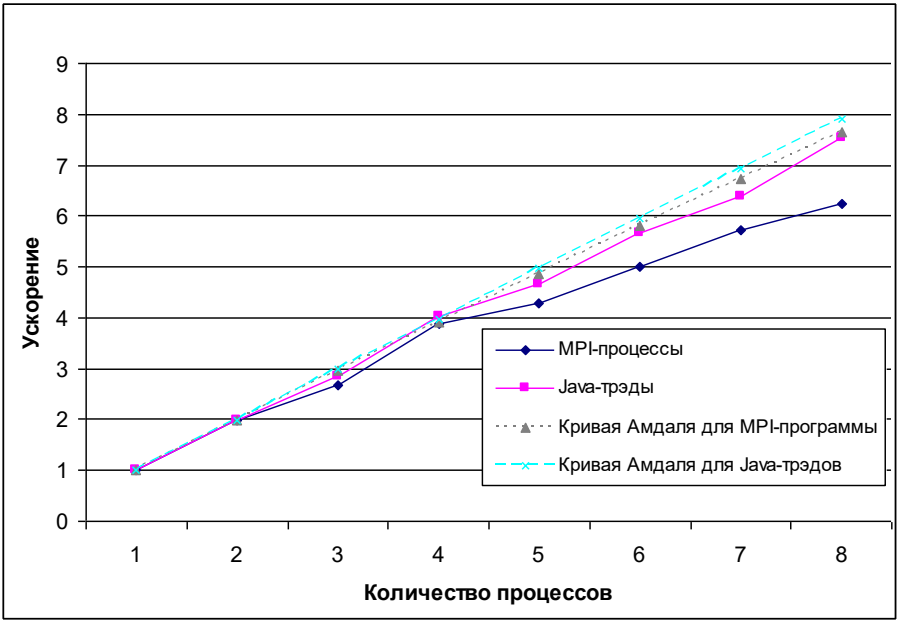


Рис. 2. Сравнение ускорения программы умножения матриц при ее выполнении на многоядерном узле кластера с использованием MPI-процессов или Java-потоков.

При выполнении MPI-программ на кластерах с многоядерными узлами оказалось, что хорошая масштабируемость получается в тех случаях, когда число процессов (или потоков), запущенных на одном узле кластера, меньше числа ядер на соответствующем узле. На рисунке 3 это явление иллюстрируется на примере программы умножения матриц (в данном примере порядок матриц n равен 5500). Видно, что с увеличением числа процессов, запущенных на узле, масштабируемость уменьшается сначала незначительно,

когда число запущенных процессов намного меньше числа ядер, а потом весьма существенно. Это явление можно объяснить резким возрастанием накладных расходов на синхронизацию обращения к данным параллельных процессов (потоков), запущенных на узле. Этот график был получен на кластере МСЦ РАН МВС-100К, который имеет узлы примерно такие же, что и кластер ИСП РАН (на каждом узле два 4-ядерных процессора Intel® Xeon® CPU X5365 с частотой 3.00GHz и с интерфейсной платой HP Mezzanine Infiniband DDR), но узлов у него гораздо больше (1410).

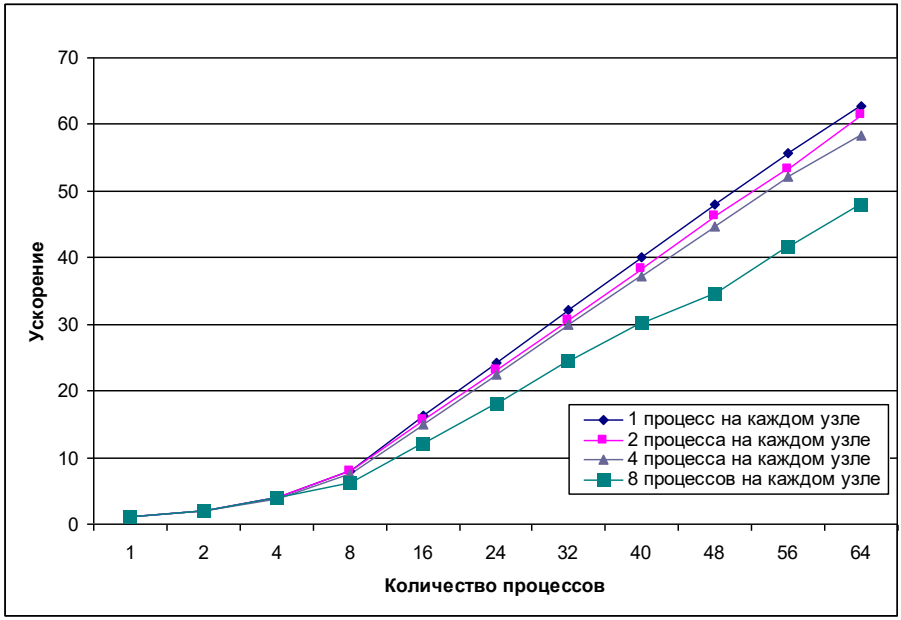


Рис. 3. Деградация масштабируемости параллельной программы с увеличением числа процессов (потоков), запущенных на многоядерном узле кластера.

Пример 2. Программа решения уравнения теплопроводности. Было рассмотрено линейное уравнение теплопроводности:

$$\frac{\partial}{\partial x}\left(k\frac{\partial u}{\partial x}\right)+F(x,t)=c\rho\frac{\partial u}{\partial t},$$

где $u(x, t)$ – температура в точке n-мерного пространства x в момент времени t, k – коэффициент теплопроводности, c – удельная теплоемкость, ρ - плотность (вообще говоря, параметры k, c и ρ могут зависеть от (x, t)).

В модельном примере рассматривался случай неоднородной плоской квадратной пластины (n = 2), т.е. рассматривалось уравнение

$$\frac{\partial}{\partial x}\left(k\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(k\frac{\partial u}{\partial y}\right)+F(x,y,t)=c\rho\frac{\partial u}{\partial t},$$

$$u(x,y,t),\quad 0<x<1,0<y<1,0<t.$$

в области $0 < x < 1$, $0 < y < 1$, $t > 0$. Начальные и граничные условия задавались следующим образом:

$$u\big|_{t=0}=\varphi(x,y),\quad 0\le x\le 1,0\le y\le 1,$$

$$u\big|_{x=0}=\psi_1(y,t),\ u\big|_{x=1}=\psi_2(y,t),\ 0\le y\le 1,0\le t,$$

$$u\big|_{y=0}=\psi_3(x,t),\ u\big|_{y=1}=\psi_4(x,t),\ 0\le x\le 1,0\le t.$$

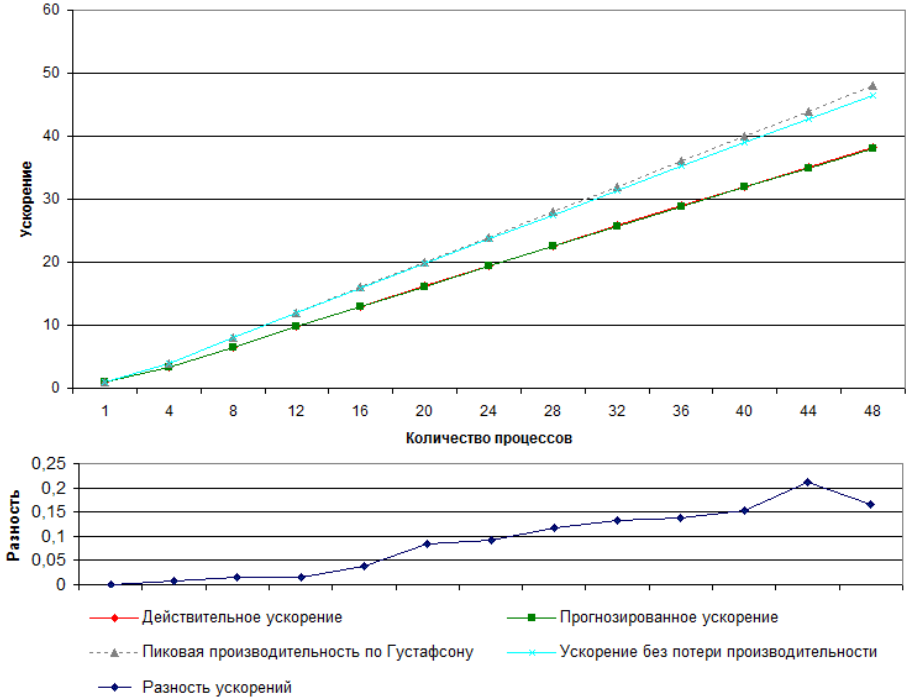


Рис. 4. Сравнение предсказанного и фактически полученного ускорения программы решения линейного уравнения теплопроводности на кластере ИСП РАН (n=6000).

В качестве граничных функций φ и ψ были выбраны $e^{-(x_1+y_1)}$ и $100*e^{-(x_1+y_1)}$ где $x_1=(x-½)^2$, $y_1=(y-½)^2$ (при соответствующих границе значений переменных). С помощью неявной разностной схемы задача свелась к системе линейных алгебраических уравнений, которая решалась итерационно с помощью одного из обобщений метода Зейделя.

На рисунке 4 представлены графики, подтверждающие высокую точность прогноза производительности программы при использовании заданного числа узлов с помощью соответствующего инструмента среды *ParJava*. Следует отметить, что максимально достижимое ускорение для данного числа узлов зависит от доли последовательных вычислений при выполнении программы и в случае фиксированного объема обрабатываемых данных определяется законом Амдаля [11]. Согласно закону Амдаля максимально достижимое ускорение $S_A(p)$ на p процессах определяется по формуле $S_A(p)=\frac{p}{1+(p-1)\cdot f}$, где f - доля не распараллеливаемой (последовательной) части программы. Для рассматриваемого примера f = 0,3%. Пунктирная кривая на рисунке – это график функции $S_A(p)$.

В работе Дж. Густафсона [12] было показано, что если объем обрабатываемых данных (в данном случае – это порядок матрицы n) меняется таким образом, чтобы объем обрабатываемых данных в каждом процессе был постоянным, то максимально достижимое ускорение определяется по формуле Густафсона $S_G(p)=p-(p-1)\cdot f$. Основное отличие функций Амдаля и Густафсона в их асимптотике: $\lim_{p\to\infty} S_A(p)=\frac{1}{f}$, тогда как $\lim_{p\to\infty} S_G(p)=\infty$.

На рисунке 5 представлены графики предсказанного и реально полученного ускорения для программы решения линейного уравнения теплопроводности на кластере ИСП РАН для случая, когда объем обрабатываемых данных в каждом процессе поддерживался постоянным (по Густафсону): в каждом процессе обрабатывалась матрица размеров $n \times k$ (n = *6000, k= 6000*). График показывает, что и в этом случае ускорение прогнозируется достаточно точно: относительная погрешность прогноза не превышает 1%. Для сравнения пунктиром показан график функции $S_G(p)$.

Пример 3. Прикладная параллельная программа численного решения системы уравнений, моделирующей процессы и условия генерации интенсивных атмосферных вихрей (ИАВ) в трехмерной сжимаемой атмосфере, исходя из теории мезомасштабных вихрей В.Н. Николаевского.

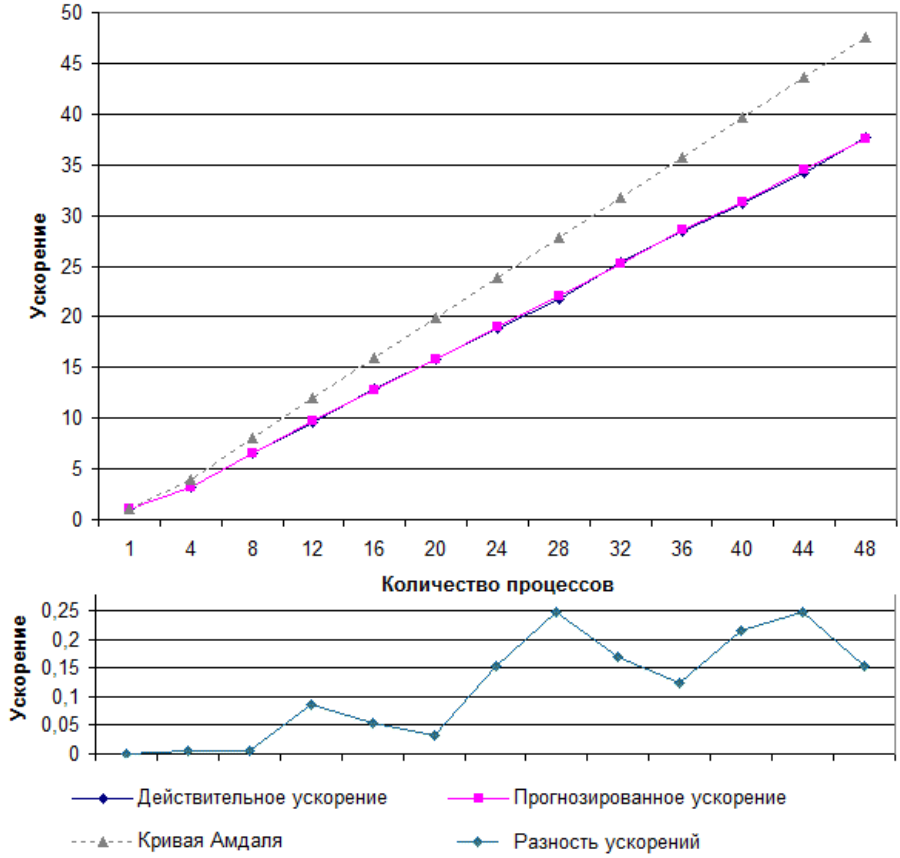


Рис. 5. Сравнение предсказанного и фактически полученного ускорения программы решения линейного уравнения теплопроводности по Густафсону.

Математическая модель описывается сильно нелинейной многокомпонентной трехмерной системой уравнений в частных производных смешанного типа, которая была выведена в работе [13]; аналитические решения ее неизвестны, а численное решение впервые было получено в работе [14]. Соответствующая параллельная программа была разработана и реализована в среде ParJava. В процессе разработки инструменты ParJava использовались для определения области масштабируемости и оптимального числа параллельных процессов, а также для настройки параллельной программы с целью ее оптимизации.

Исследования показали, что при выбранной схеме вычислений доля последовательных вычислений f = 0,83%. Разработанная программа показала достаточно высокую производительность и хорошую масштабируемость. На

рисунках приведены графики зависимости ускорения программы моделирования ИАВ для кластера ИСП РАН (рисунок 6) и МСЦ РАН (рисунок 7). Графики показывают достаточно хорошее совпадение предсказанных и реальных значений производительности программы. Моделировалась 64-разрядная версия программы на Java 1.6. Как видно из графика, начиная со 120 процессов, реальное ускорение программы идет на спад. Интерпретатор также фиксирует, что в данной точке производительность

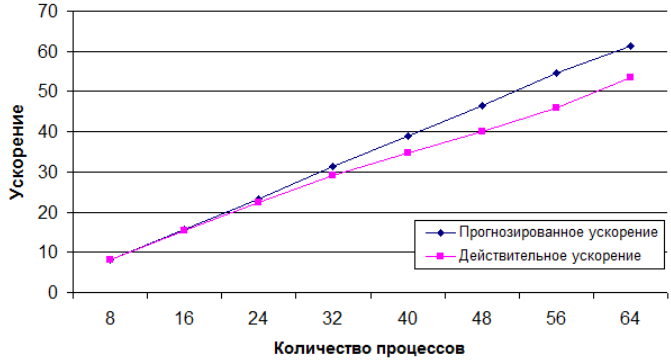


Рис. 6. Ускорение программы моделирования ИАВ на кластере ИСП РАН.

программы падает. Отметим, что такое падение производительности связано с асимптотикой формулы Амдала ($\lim_{p\to\infty} S_A \ \ p = \frac{1}{f}$). В самом деле, 1/0,0083 ≈ 120.

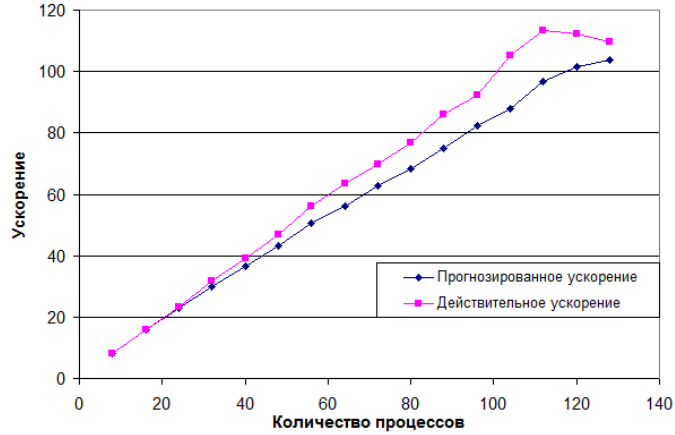


Рис. 7. Ускорение программы моделирования ИАВ на кластере МСЦ РАН.

7. Заключение.

Предложенные методы оценки времени выполнения гнезд циклов реализованы и интегрированы в инструментальную среду разработки параллельных программ ParJava [4]. Предложенный в данной статье подход позволяет существенно сократить степень участия прикладного программиста при получении указанных оценок и в то же время уменьшить погрешность предсказания времени выполнения. Эти методы применимы к реальным прикладным параллельным программам. Рассмотренные методы позволили не только получить приемлемую относительную погрешность оценки (погрешность прогноза не превышает 15%), но и ускорило время интерпретации модели.

Новые методы оценки времени выполнения гнезд циклов особенно актуальны при использовании кластеров с многоядерными узлами, так как позволяют учесть эффект потери производительности при чрезмерной загрузке узлов кластера и определить оптимальное число процессов, запускаемых на узлах.

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] Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express: Towards Thread Safe Java HPC, Submitted to the IEEE International Conference on Cluster Computing (Cluster 2006), Barcelona, Spain, 25-28 September, 2006.

[3] Markus Bornemann , Rob V. Van Nieuwpoort , Thilo Kielmann. MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java. Euro PVM/MPI 2005, volume 3666

[4] 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

[5] В.П. Иванников, А.И. Аветисян, С.С. Гайсарян, В.А. Падарян. Прогнозирование произ-водительности MPI-программ на основе моделей. «Автоматика и телемеханика», 2007, №5, с. 8-17

[6] А.И. Аветисян, В.В. Бабкова и А.Ю. Губарь. Возникновение торнадо: трехмерная численная модель в мезомасштабной теории турбулентности по В.Н. Николаевскому// ДАН/Геофизика, т. 419, №4, с. 547-552. Москва 2008.

[7] Alpern A.B, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K.S. Mckinley, M. Mergen, J.E.B. Moss, T. Ngo, V. Sarkar. The Jikes Research Virtual Machine project: IBM Systems Journal, Vol. 44, No 2, 2005

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jaffrey D. Ullman. Compilers: principles, techniques, and tools. – 2nd ed., Pearson Education Inc., 2007, с.836.

[9] M. Paleczny, C. Vick, and C. Click. The Java HotSpotTM server compiler. In Proceedings of the Java Virtual Machine Research and Technology Symposium, pages 1–12, 2001.

[10] V.P. Ivannikov, A.I. Avetisyan, V.V. Babkova, A.Yu. Gubar “Tornado arising modeling using high performance cluster systems” Sixth International Conference on Computer Science and Information Technologies (CSIT’2007), 24-28 September, Yerevan, Armenia

[11] Amdahl G.M. Validity of single-processor approach to achieving large-scale computing capability, Proceedings of AFIPS Conference, Reston, VA. 1967. pp. 483-485
[12] Gustafson J.L., Reevaluating Amdahl's Law, CACM, 31(5), 1988. pp. 532-533.
[13] Аветисян А.И., Бабкова В., Гайсарян С.С., Губарь А.Ю.. Рождение торнадо в теории мезомасштабной турбулентности по Николаевскому. Трехмерная численная модель в ParJava. Журнал «Математическое моделирование», том 20, №8, с. 28-40, 2008
[14] Аветисян А.И., Бабкова В.В., Губарь А.Ю. «Моделирование интенсивных атмосферных вихрей в среде ParJava.» Всероссийская научная конференция «Научный сервис в сети Интернет: технологии параллельного программирования», г. Новороссийск, 2006. с. 109-112.

The methods of precise measurement of the loop nests' execution time during JavaMPI-programs analysis in ParJava environment

A.I. Avetisyan, M.S. Akopyan, S.S. Gaissaryan
arut@ispras.ru, manuk@ispras.ru, ssg@ispras.ru

Abstract. The methods for estimating execution time of the model of a parallel program using instrumental computer are discussed. The methods are based on accurate prediction of the execution time of fragments of the parallel program using the target computational platform. The model was developed for SPMD programs using explicit data exchange by Java MPI library and is the part of ParJava IDE. Certain kinds of loops (homogeneous, reducible) are marked out in the model and then estimated on the node of a target computational platform (high performance cluster). The technique allows to reduce prediction error and to accelerate the model simulation on the instrumental computer.

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

Поддержка команд с условным выполнением в селективном планировщике команд

Дмитрий Мельник <dm@ispras.ru>,
Александр Монаков <amonakov@ispras.ru>

Аннотация. Условное выполнение — аппаратная возможность, реализованная в некоторых процессорах, позволяющая аннотировать команды условным предикатом, при этом команда исполняется только в случае истинности предиката. В данной работе предлагается метод для поддержки условного выполнения во время планирования команд, а также рассматриваются преимущества данного подхода по сравнению с отдельной оптимизацией, работающей до планирования команд. Предложенный метод был реализован в селективном планировщике в компиляторе GCC. Тестирование реализации показало рост производительности на тестах SPECFP набора SPEC CPU2000 в среднем почти на 2% (и до 16% на отдельных тестах).

Ключевые слова: планирование команд, условное выполнение, GCC.

1. Введение

Современные микропроцессоры используют параллелизм на различных уровнях архитектуры. Одним из видов параллелизма является параллелизм на уровне команд: несколько независимых команд программы могут быть выполнены на отдельных функциональных устройствах процессора. Чтобы использовать такой вид параллелизма, необходимо уметь выбирать из потока программы независимые команды, и назначать их на функциональные устройства процессора. Для этого применяется либо аппаратное динамическое переупорядочение команд во время выполнения, либо порядок команд определяется во время компиляции, либо комбинация этих методов. Достоинством первого метода является то, что на этапе выполнения доступна более полная информация о допустимости переупорядочения команд и фактической загрузке функциональных устройств, однако он увеличивает энергопотребление процессора за счет работы дополнительной логики в процессоре, которая при этом должна быть сравнительно простой из соображений энергоэффективности. Именно такой подход используется в процессорах семейств x86 и x86_64. Соответственно, использование второго метода позволяет снизить энергопотребление процессора, в то время как на этапе компиляции статическая оптимизация может быть сколь угодно

сложной (но и менее точной). Именно этот метод используется в архитектурах с очень длинным командным словом, например, в *архитектуре с явным параллелизмом команд* — EPIC (Explicitly Parallel Instruction Computing [1]), реализованной в процессорах семейства Itanium. EPIC-программа содержит явные указания на то, какие инструкции должны выполняться параллельно, и на каких функциональных устройствах. Идея EPIC состоит в том, чтобы переложить задачу по поиску независимых инструкций на компилятор, при этом предоставив ему дополнительные средства для выполнения оптимизации, такие как спекулятивное выполнение, условное выполнение, вращающиеся регистровые окна. Также архитектура предоставляет большое число функциональных устройств и большое число регистров.

В виду перечисленных особенностей архитектуры EPIC особую важность в компиляторе приобретает *планирование команд*. Основной задачей этой оптимизации является выбор независимых команд из потока программы и упаковка их машинные слова («пакеты») в определенном порядке, которые выполняются EPIC-процессором параллельно.

В качестве основы для нашей работы был выбран метод селективного планирования [2], основными особенностями которого являются:

- Поддержка произвольных ациклических регионов графа потока управления (регион может иметь несколько входов и выходов). Поддержка больших регионов со сложными структурами управления позволяет находить больше команд, которые могут выполняться параллельно;
- Поддержка конвейеризации циклов с неизвестным заранее числом итераций (non-counted loops);
- Поддержка ряда преобразований команд, позволяющих устранять некоторые зависимости по данным и по управлению, а также отождествлять команды с разных путей;
- Поддержка создания новых команд в процессе планирования, например, в качестве компенсационного кода при перемещении только по одной дуге или дополнительных команд восстановления при спекулятивных загрузках.

В данной работе мы описываем, как поддержка условного выполнения, одной из возможностей архитектуры EPIC, может быть реализована в планировщике команд, а также рассматриваем преимущества данного подхода по сравнению с отдельной оптимизацией, работающей до планирования команд. Мы также приводим результаты тестирования нашего подхода на наборе тестов SPEC 2000.

2. Описание алгоритма селективного планирования

В данном разделе мы в общих чертах даем описание алгоритма селективного планирования, уделяя основное внимание особенностям, существенным с

точки зрения добавления поддержки условного выполнения. Детальное описание исходного алгоритма может быть найдено в работе [2].

2.1. Основные понятия и этапы селективного планирования

Областью работы алгоритма является произвольный ациклический участок графа потока управления[1]. Такой участок мы будем называть *регионом*. Регион может иметь несколько входов и выходов — дуг, входящих в регион из других участков, или исходящих из него.

Параллельная группа команд — группа независимых команд, которые на данной архитектуре могут быть выполнены одновременно. Каждая группа может иметь несколько *границ* (boundaries) — дуг графа потока управления, соединяющих команды, находящиеся внутри группы, с командами за ее пределами. Границы одной параллельной группы образуют *барьер* (fence), который перемещается по мере включения команд региона в параллельную группу. Во время планирования может существовать одновременно несколько параллельных групп (барьеров), на границах которых происходит планирование, но в каждый момент только один из барьеров является активным. На рис. 1 проиллюстрированы эти понятия, а также изменение границ параллельной группы по мере включения в нее новых команд. Заметим, что архитектура EPIC допускает одновременное исполнение до 3-х команд условного перехода за один такт, при этом передачу управления выполняет только первая из команд с истинным предикатом.

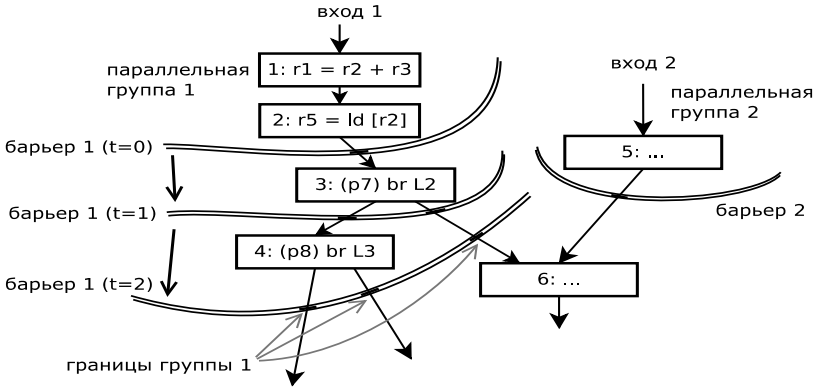


Рис. 1. Перемещение барьера и появление новых границ при последовательном включении команд в параллельную группу.

[1] Под графом потока управления программы будем понимать граф, вершинами которого являются отдельные команды программы, а дуги определяют допустимую последовательность их выполнения.

На самом верхнем уровне, процедура селективного планирования выглядит следующим образом. Сначала на каждом входе в регион создаются пустые параллельные группы команд, в которые в процессе планирования будут переноситься команды. Затем планировщик циклически выполняет следующие шаги:

- Построение множества доступных команд — кандидатов для переноса в текущую параллельную группу.
- Определение множества доступных регистров. Для каждой команды из множества доступных определяется множество регистров назначения, с которыми она может быть запланирована.
- Выбор «наилучшей» команды из множества доступных для планирования с помощью эвристик. Например, учитывается вероятность исполнения дуг, по которым доступна команда, длина критического пути в графе зависимостей, доступна ли команда только в спекулятивной форме, и может ли быть запланирована с исходным регистром. Более подробно эти эвристики рассматриваются в работе [2].
- Фактическое перемещение команды в параллельную группу. Команда перемещается из исходного места в регионе к границам параллельной группы, при этом выполняются все необходимые преобразования, а также создается дополнительный компенсационный код там, где это необходимо для сохранения семантики программы.
- Перемещение границы планирования. Если текущая параллельная группа заполнена (в нее не может быть включено больше команд из-за ограничений по ресурсам), либо множество доступных команд исчерпано, то на границах текущей параллельной группы создаются новые параллельные группы (и барьеры), и планирование продолжается со следующего барьера.

В нашей работе наиболее существенные изменения были внесены в первые два шага алгоритма, поэтому рассмотрим их более подробно.

2.2. Построение множества доступных выражений

Для того чтобы заполнить параллельную группу, прежде всего необходимо определить множество команд, которые потенциально могут быть включены в нее. При решении этой задачи рассматриваются не только машинные команды целиком, но и отдельно правые части выражений, соответствующих этим командам. Такой подход позволяет отождествлять правые части выражений, доступных вдоль различных путей в графе потока управления, а также выбирать любой доступный регистр в точке планирования, если исходный регистр назначения используется на пути переноса команды к точке планирования. Правая часть может быть выделена, если соответствующее ей

выражение не имеет побочных эффектов, как, например, у команды *a = b * c*. В то же время, команды из ассемблерных вставок, команды с автоинкрементом и т.п. не могут быть разделены.

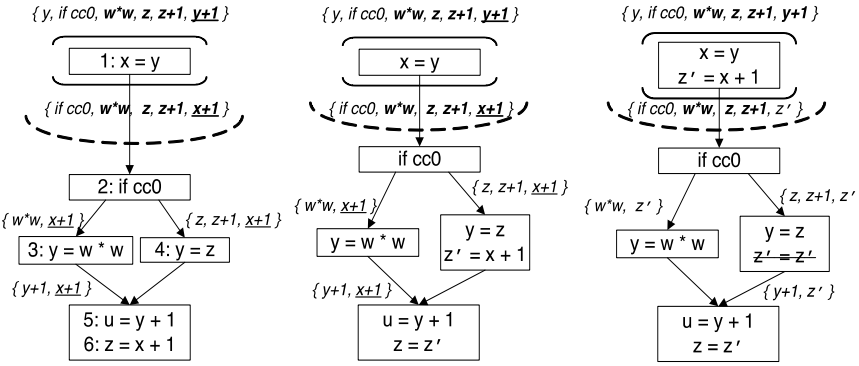


Рис. 2. Пример работы алгоритма выборочного планирования. а) исходный граф; б) команда z' =y+1 перемещается вдоль левой ветви; в) команда перемещается вдоль правой ветви.

Множество доступных команд в точке программы, предшествующей команде *i*, содержит команды, которые доступны для планирования в этой точке, и обозначается как *av_set(i)*. Здесь и далее под командами, в качестве элементов этих множеств, понимаются либо команды целиком, либо их правые части без регистра назначения. Последние становятся полноценными командами только после выбора целевого регистра, который происходит непосредственно перед планированием команды.

Множества доступных команд вычисляются для каждой команды в регионе в обратном топологическом порядке, начиная с «листовых» команд. Сначала вычисляется объединение множеств потомков команды *i*, а затем из него удаляются команды, недоступные над *i* (т.е. команды, имеющие зависимости с *i*). К результирующему множеству добавляется сама команда *i*, либо ее правая часть (ниже она обозначена как *av_op(i)*):

$$av\_set(i) = moveup\_set(i, \bigcup_{x \in Succ(i)} av\_set(x)) \bigcup av\_op(i)$$

Здесь функция *moveup_set(i, S)* удаляет из множества *S* команды, имеющие зависимости с командой *i*. Кроме того, она может выполнять преобразование команд в форму, в которой они не зависят от *i*. В частности, поддерживаются подстановка через присваивание, условное выполнение, а также спекулятивное выполнение.

Рассмотрим построение множеств *av_set* на примере рис. 2(а), где они приведены в фигурных скобках для первых команд базовых блоков. Так, команда *z + 1* из в множества *av_set(4)* была получена из команды *y + 1*, принадлежащей множеству *av_set(5)*, с помощью подстановки через присваивание *y = z*. Без выполнения этого преобразования поменять местами команды 4 и 5 было бы невозможно, т.к. они имеют истинную зависимость по *y*. В то же время, команда 5 ни в какой форме не содержится в *av_set(3)* из-за зависимости с командой 3, которая не может быть устранена с помощью подстановки, т.к. как она не является простой пересылкой, в отличие от команды 4.

После того как вычислены множества доступных команд на границах параллельной группы, строится множество доступных команд для параллельной группы в целом. Оно вычисляется аналогичным образом, распространяя множества *av_set* от границ параллельной группы к ее вершине через уже запланированные операции, при этом также выполняется подстановка аргумента при прохождении на пути вверх через операции копирования. Это позволяет в одной параллельной группе планировать операции, которые в противном случае имели бы истинные зависимости по данным. На рис. 2 выражение *x + 1*, доступное на границе параллельной группы, преобразуется в *y + 1*, которое теперь может быть выполнено параллельно вместе с уже запланированной командой *x = y*.

2.3. Определение множества доступных целевых регистров для выражения

Так как некоторые команды в множестве доступных присутствуют в нем только в виде своих правых частей без указания целевого регистра, то сначала необходимо выбрать целевой регистр, с которым команда будет запланирована. Для этого необходимо определить множество регистров, доступных на пути перемещения команды от ее исходного места в регионе до точки планирования.

Критерием доступности регистра является выполнение следующих условий:

1. Регистр не читается и не записывается ни на одном из путей, вдоль которых доступна выбранная команда;
2. Регистр не жив непосредственно после одной из первоначальных команд (за исключением целевого регистра соответствующей команды);
3. Регистр не жив в начале лежащих вне путей переноса ветвях условных переходов, через которые эта команда была перенесена

Для того чтобы проверить эти условия, для каждого выражения выполняется обход тех путей графа потока управления, по которым доступно исходное выражение, при этом выполняются преобразования, обратные тем, которые выполнялись при подъеме данного выражения к границе планирования.

2.4. Конвейеризация циклов при селективном планировании

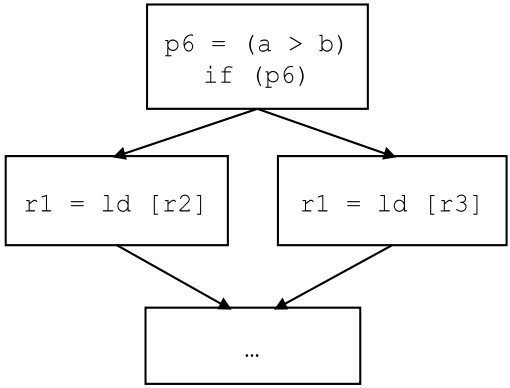
С помощью селективного планировщика можно также выполнять конвейеризацию циклов [3]. Т.к. планировщик может работать только с ациклическими регионами, для представления тела цикла в виде ациклического региона применяется специальная техника. Она заключается в том, что в каждый момент подразумевается, что цикл «разорван» по текущим барьерам планирования, что делает возможным перемещение команд через обратную дугу, и невозможным — через текущий барьер. После планирования текущей параллельной группы барьер перемещается, что делает только что запланированные команды доступными для повторного планирования. Таким образом, эффект конвейеризации достигается за счет перемещения команд через обратную дугу, а пролог цикла создается в качестве компенсационного кода в момент такого перемещения команды.

2.5. Реализация селективного планировщика в GCC

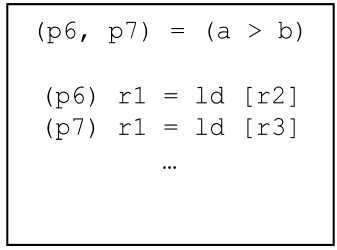
Ранее алгоритм селективного планирования был реализован нами в свободном компиляторе GCC [4]. Реализованный планировщик включен в основную ветвь разработки компилятора, начиная с версии 4.4, и используется по умолчанию с уровнем оптимизации `-O3` на архитектуре Itanium. В нашей реализации также была добавлена поддержка спекулятивного выполнения по данным и управлению [5, 6], а также выполнен ряд усовершенствований алгоритма , прежде всего касающихся его быстродействия [7]. Среди них стоит отметить оптимизацию вычисления доступности регистров, сохранение истории преобразований команд, изменение процедуры вычисления приоритетов команд, а также ряд других эвристик. Реализация поддержки команд с условным выполнением, рассматриваемая в данной работе, была выполнена в селективном планировщике GCC.

3. Условное выполнение команд

Одним из факторов, ограничивающих эффективность конвейеризации циклов, является наличие зависимостей по управлению внутри тела конвейеризуемого цикла. Поскольку, вообще говоря, невозможно заранее вычислить, в каком направлении будет выполнен переход на очередной команде передачи управления, чтобы не прекращать наполнение конвейера до того момента, как эта команда будет полностью выполнена, в процессорах применяются схемы предсказания переходов. Их функционирование основывается на эвристиках и сборе статистики о переходах во время выполнения программы; также в некоторых процессорах (в т.ч. в Itanium) существует возможность программно влиять на поведение модуля предсказания переходов. Однако, если предсказание перехода было выдано неверно, происходит сброс конвейера — выполнение всех команд, выданных после неверно предсказанной команды условного перехода, останавливается, и начинается выполнение команды с противоположной ветви условия.



а)



б)

Рис. 3. Преобразование ветвления в программе в последовательный код

Чтобы избежать потерь в производительности при неправильном предсказании условного перехода, в некоторых процессорах реализована поддержка условного выполнения команд. В процессоре Itanium существует возможность аннотировать любую из команд одним из 64 парных предикатных регистров, при этом команда будет выполнена в том и только в том случае, если указанный предикатный регистр имеет значение «истина». Вместо регистра флагов, команды сравнения устанавливают пару соседних предикатных регистров в противоположные значения; таким образом, можно одновременно хранить результат 31 сравнения (значение нулевого предикатного регистра фиксировано и равно логической единице). В коде каждой команды есть шестибитное поле, в котором записан номер предикатного регистра, контролирующего её выполнение (наличие всегда установленного в единицу предикатного регистра позволяет единообразно записывать условно и безусловно выполняющиеся команды). Условные переходы записываются как безусловные, защищённые соответствующим предикатом.

Возможность условного выполнения в процессоре требует соответствующей поддержки со стороны компилятора. В простейшем случае, она будет заключаться в удалении условных переходов внутри тела цикла, при этом все команды в базовых блоках, выполнение которых зависело от этого перехода, защищаются соответствующим предикатом (рис. 3). Чтобы избежать избыточного увеличения размера кода, обычно применяются эвристики. Так, в компиляторе GCC максимальное число команд для преобразованного блока для архитектуры EPIC задано равным 12 (что соответствует максимальному числу команд, которое может быть выполнено за два такта), а в работе [8] преобразование применяется только к тем базовым блокам, которые не содержат вызовов функций и длинных цепочек зависимостей по данным. В

работе [9] предлагается производить полное преобразование ветвлений для всех базовых блоков программы сравнительно рано в процессе оптимизации кода. Это позволяет упростить граф потока управления для последующих оптимизаций, но для эффективной работы требует добавления учёта условного выполнения в анализ зависимостей по данным. Чтобы отменить негативный эффект слишком агрессивного преобразования команд в условную форму, во время первого прохода планировщика команд производится обратное преобразование ветвлений. В работе [10] рассматривается подход к оптимизации ветвлений на основе минимизации логических схем, описывающих зависимости условий переходов. Это позволяет упростить граф потока управления и оптимизировать вычисление предикатов.

Общим недостатком всех рассмотренных подходов является то, что выгодность преобразования оценивается только исходя из количества преобразованных команд, а их требования к ресурсам процессора не учитываются (например, на процессорах архитектуры Itanium 2, 12 команд с плавающей точкой не могут быть выполнены быстрее чем за 6 тактов, так как процессор имеет только два устройства для вещественных вычислений, но в то же время 12 простых арифметических команд могут быть выполнены за 2 такта). Кроме того, в некоторых случаях имеет смысл преобразовывать в условную форму не целые блоки, а их части, например, чтобы иметь возможность начать загрузки из памяти раньше, чем произойдёт переход на содержащий её базовый блок (при условии что предикат, контролирующий этот переход, может быть вычислен достаточно рано).

Описанные недостатки могут быть устранены, если осуществлять преобразование ветвлений в процессе планирования команд, т.к. в планировщике известна наиболее точная информация о распределении команд на функциональные блоки процессора, и её можно использовать для принятия решений о переносе команд через ветвления. В данной работе рассматривается, как поддержка условного выполнения может быть реализована в селективном планировщике [2].

4. Поддержка условного выполнения в селективном планировщике

Добавление поддержки команд с условным выполнением в селективный планировщик включает следующие подзадачи:

1. Вычисление предиката условно выполняющейся команды, сохранение его в атрибутах команды. Добавление предиката к команде.
2. Добавление условно выполняющихся команд в множество готовых инструкций.

3. Обеспечение корректности при переносе выбранной в условно выполняющейся форме команды к границе планирования.
4. Обеспечение корректности преобразований, применяемых селективным планировщиком к команде в условно выполняющейся форме.

В данном разделе мы рассмотрим подробно каждую из этих задач.

4.1. Обработка условий в планировщике

Для реализации поддержки команд с условным выполнением потребовалось внести ряд изменений в структуры данных и функции планировщика. В частности, в множестве готовых команд вместе с самой командой должен храниться также и предикат, с которым она доступна. Этот предикат вычисляется на этапе инициализации структур данных планировщика в функции, выполняющей анализ команды в форме внутреннего представления компилятора GCC. Среди других изменений можно отметить добавление кэша преобразований, позволяющего хранить различные условные формы для команд, доступных с разных ветвей условного перехода (и получающие предикат условного перехода или его отрицание, соответственно), а также поддержку разделения инструкций в условной форме на левую и правую части на этапе анализа, что позволяет в таких командах выполнять переименование регистров.

4.2. Добавление условно выполняющихся команд в множество готовых инструкций

Преобразование команд в форму с условным выполнением отличается от других преобразований в селективном планировщике, таких как подстановка через присваивание и спекулятивное выполнение. Во-первых, для выполнения данного преобразования необходимо знать, на какой ветви условного перехода доступна команда: к командам с ветви, на которую осуществляется переход, должен быть присоединён предикат, контролирующий переход, а для команд с противоположной ветви — обратный ему. Поэтому присоединение предикатов необходимо выполнять во время объединения множеств доступных команд от обеих ветвей на команде условного перехода, а не при вычислении множества доступных команд, как это делается, например, при спекулятивном выполнении. Во-вторых, при преобразовании команд в условную форму, исходные команды также остаются в множестве доступных, таким образом сохраняется возможность выдать исходную команду в спекулятивной форме, например, еще до того, как вычислен предикат.

Пусть *cond* — условие, контролирующее переход, $avset_1$ и $avset_2$ — множества доступных команд, вычисленные для команд на ветвях условного перехода. Тогда множество доступных команд непосредственно после перехода *J* с добавленными командами в условной форме имеет вид:

$$\text{avset\_below}(J) = \bigcup_{I' \in \text{Succ}(I)} \text{avset}(I') \cup$$

$$\bigcup_{I' \in \text{avset}_1} \{apply\_predicate(I', cond)\} \cup \bigcup_{I' \in \text{avset}_2} \{apply\_predicate(I', \neg cond)\}$$

Здесь функция *apply_predicate(I, cond)* присоединяет предикат *cond* к команде *I*.

Чтобы при последующем подъёме *avset_below(J)* через условный переход *J* команды в условной форме не были удалены из множества готовых инструкций, необходимо явно разрешать перенос условных команд через переходы с тем же предикатом в функции *move_up*. Исключение составляют команды, модифицирующие предикатный регистр, контролирующий этот условный переход.

4.3. Перенос к границе планирования

Реализация поддержки условного выполнения требует следующих изменений в процедуре перемещения выбранной команды к текущей границе планирования (функция *move_op* [2]). На этапе поиска изначальных команд при спуске через команду условного перехода необходимо преобразовывать команды, контролируемые тем же предикатом, из условной формы в обычную, удаляя соответствующий предикат у команд из текущего множества изначальных команд *orig_ops*. При этом поиск необходимо продолжать только на одной ветви, соответствующей предикату выбранной команды. Поиск изначальных команд на ветви условного оператора, соответствующей противоположному значению предиката, даже если они там есть в какой-либо форме, не должен производиться из соображений корректности: в противном случае условная команда, которая не могла бы выполниться на данном пути (т.к. он содержит условный переход по противоположному предикату), будет поднята выше условного перехода. В этом случае команда может быть исполнена при соответствующем значении предиката, что противоречит семантике исходной программы.

Еще одно изменение касается создания компенсационного кода. Во время поиска изначальных команд в функции *move_op* планировщик удаляет все найденные формы перемещаемой к границе планирования команды, включая и созданный ранее компенсационный код. Так, на рис. 4 показано, как при планировании обычной команды *a = a + b* сначала будет найдена по левой ветви и удалена изначальная команда 1, что на обратном пути приведет к созданию компенсационного кода 3, а затем и кода 2. После этого, уже при проходе по правым ветвям, сначала будет найдена и удалена команда 2, а затем команда 3, после чего команда *a = a + b* будет перемещена в верхний блок.

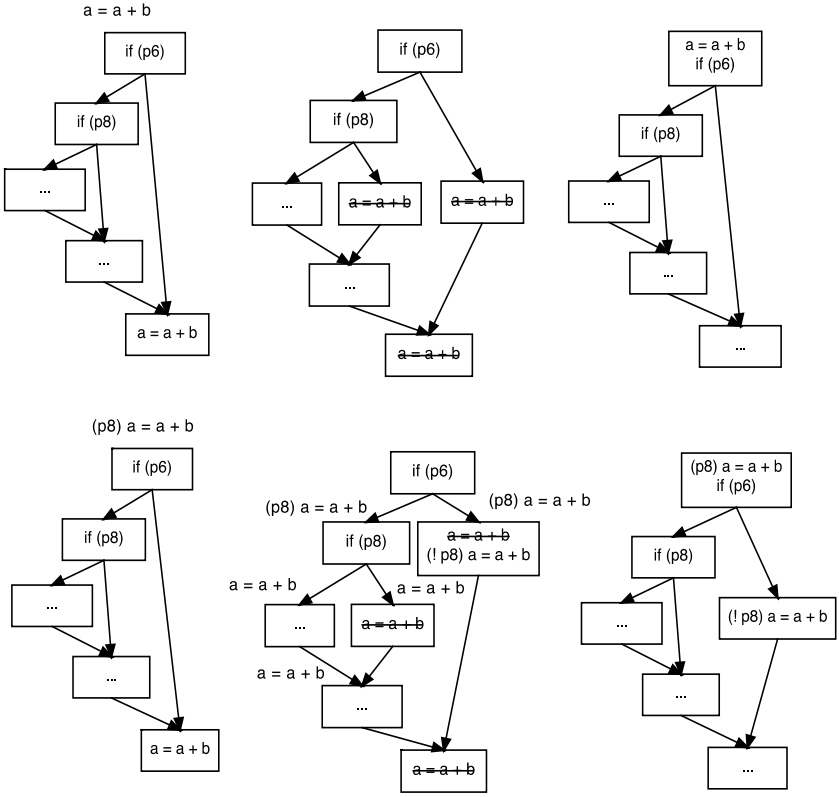


Рис. 4. Создание компенсационного кода при планировании: а) обычной команды; б) команды с условным предикатом

В случае планирования команд с условным предикатом, такое удаление созданного ранее компенсационного кода может привести к созданию некорректного кода. На рис. 4(б) показано планирование того же участка программы, но для планирования выбрана команда с условным выполнением *(p8) a = a + b*. Если бы мы поступили прежним образом и просто удалили компенсационный код, то переменной *a* не присваивалось бы правильное значение в случае, когда предикат *p8* не установлен. Для сохранения корректности кода на всех путях выполнения при нахождении изначальной команды необходимо вместо ее удаления добавить к ней предикат, противоположный тому, который эта команда имеет в множестве изначальных команд *orig_ops*. Удаление команды происходит только в том случае, если команда в этом множестве находится в безусловной форме.

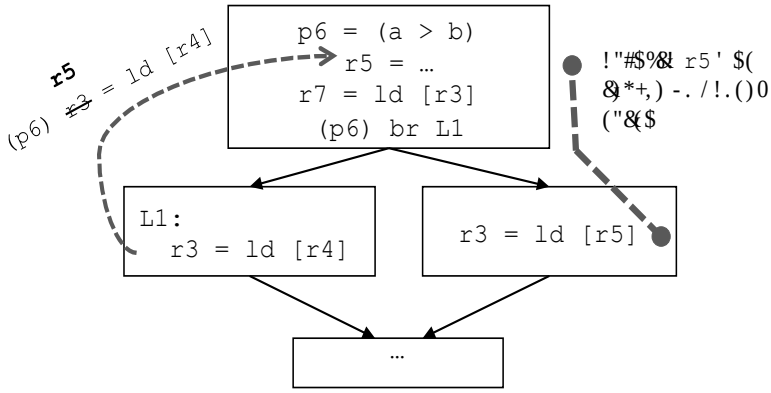


Рис. 5. Выбор регистра назначения для команды с условным выполнением при переименовании регистров

4.4. Взаимодействие с другими преобразованиями

4.4.1 Подстановка через присваивание

В дополнение к подстановке регистра при перемещении команды в условной форме через безусловное присваивание, добавляется возможность осуществлять подстановку через условно выполняемое присваивание, если выполнение команды, в которой выполняется подстановка, контролируется тем же предикатом (табл. 1).

Код до преобразования	Код после преобразования
(p6) movl r32=r33 (p6) ld4 r34=[r32]	(p6) ld4 r34=[r33] (p6) movl r32=r33

4.4.2 Переименование регистров

Заметим, что перенос команды присваивания через условный переход с преобразованием её в условную форму не влияет на значение её целевого регистра на ветке перехода, на которой эта команда не доступна. Например, на рис. 5 для команды `r3 = ld [r4]`, которую можно преобразовать в условную форму и выдать сразу после вычисления предиката, в качестве регистра назначения может быть выбран `r5`, несмотря на то, что его значение используется на другой ветви. Соответственно, условие (3) из критерия доступности регистра в разделе 2.3 можно уточнить следующим образом:

(3). Регистр не жив в начале лежащих вне путей переноса ветвях условных переходов, через которые эта команда была перенесена без

преобразования в форму с условным выполнением по предикату, контролирующему этот переход.

Соответственно, механизм вычисления множества недоступных целевых регистров и атрибута *target_available* [7] был модифицирован с учётом этого уточнения. Для атрибута *target_available* достаточно пропускать его обновление при объединении множеств доступных команд на команде ветвления для команд, получивших при переносе через ветвление условную форму.

4.4.3 Использование спекулятивных загрузок

Условное выполнение может также комбинироваться со спекулятивным выполнением команд в произвольном порядке. Например, перенесённая через два ветвления и ставшая спекулятивной загрузка может быть защищена предикатом одного из переходов, что выгодно отличается от простой спекулятивной формы тем, что при ложном предикате обращения к памяти не происходит.

4.5. Конвейеризация циклов с использованием условного выполнения

Поддержка условного выполнения позволяет улучшить конвейеризацию циклов, выполняемую селективным планировщиком.

Во-первых, перенос команд через обратную дугу цикла с добавлением условия, контролирующего выход из цикла, не приводит к увеличению количества команд, которые будут выполнены при запуске конвейеризованного цикла (на последней итерации цикла предикаты команд, перенесённых через обратную дугу, будут ложными).

Во-вторых, поскольку доступность целевого регистра не сбрасывается при переносе через условный переход в конце цикла (даже если этот регистр жив на выходе из цикла), необходимость в использовании переименования регистров возникает реже. Это позволяет сократить увеличение размера кода.

Наконец, появляется возможность переносить команды загрузки из памяти через обратную дугу цикла без использования спекулятивного чтения, что позволяет обойтись без команды проверки спекулятивного чтения. Это позволяет не только сократить размер кода, но и дополнительно оказывает положительный эффект на производительность за счет того, что увеличивается расстояние между командами спекулятивной загрузки и проверки, т.к. если при планировании спекулятивной команды на место исходной вставляется проверка спекулятивного чтения, не всегда оказываются разнесены на достаточное расстояние, чтобы первая закончила свое выполнение до того, как второй понадобится ее результат, что может привести к задержке.

Кроме того, появляется возможность переносить команды, зависящие от этой загрузки (в случае, если используются спекулятивные чтения, это требует генерирования компенсирующего кода в блоке, на который осуществляется переход командой проверки; некоторые зависимые команды, например команды записи в память, не могут быть перенесены, так как не допускают использование непроверенных результатов спекулятивных загрузок).

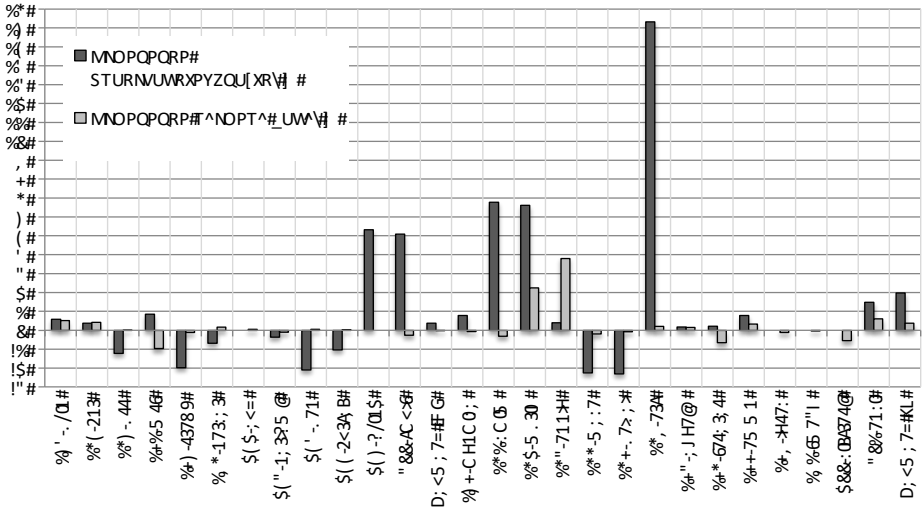


Рис. 6. Изменение производительности и размера кода приложений SPEC CPU 2000 для селективного планировщика с

5. Экспериментальные результаты

Разработанный алгоритм был опробован на наборе SPEC CPU 2000. При компиляции для базовых измерений использовались ключи компиляции `-O3 -ffast-math -fno-auto-inc-dec`; на рис. 6 показаны процентные изменения производительности и размера кода тестов при включении поддержки условного выполнения в селективном планировщике.

Пять тестов показали заметный рост производительности (на 5% и более), несколько тестов замедлились, но не более, чем на 2.5%. В целом, производительность набора тестов вещественных вычислений увеличилась на 2%, изменение производительности тестов целочисленных вычислений находится в пределах погрешности измерений. Размер кода тестовых приложений практически не изменился.

6. Заключение

В данной работе был предложен новый подход к поддержке команд с условным выполнением во время планирования команд. Был разработан алгоритм, осуществляющий преобразование команд в условную форму во время работы селективного планировщика. Алгоритм был реализован в селективном планировщике компилятора GCC, и подан для включения в основную ветку разработки компилятора начиная с версии 4.7.

Реализация была протестирована на наборе тестов SPEC CPU 2000. Рост производительности на тестах SPECFP набора SPEC CPU 2000 составил почти 2%, при этом отдельные тесты ускорились на 7 и 16%, а также на 5% выросли два теста из SPECINT. Размер кода при этом изменился незначительно.

Список литературы

[1] Intel Itanium 2 Processor Reference Manual for Software Development and Optimization,

[2] http://www.intel.com/design/itanium/manuals/iiasdmanual.htm. Дата обращения: 01.05.2011.

[3] Soo-Mook Moon and Kemal Ebcioglu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. ACM TOPLAS, Vol 19, No. 6, pp. 853—898, November 1997.

[4] S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann, 3rd ed., 1997.

[5] GCC, the GNU Compiler Collection, http://gcc.gnu.org/. Дата обращения: 01.05.2011.

[6] Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, and Dmitry Zhurikhin. An interblock VLIW-targeted instruction scheduler for GCC. Proceedings of GCC Developers' Summit 2006.

[7] Белеванцев, М. Кувырков, Д. Мельник. Использование параллелизма на уровне команд в компиляторе для Intel Itanium. Труды Института системного программирования РАН, 2006.

[8] Andrey Belevantsev, Maxim Kuvyrkov, Alexander Monakov, Dmitry Melnik, Dmitry Zhurikhin. Implementing an instruction scheduler for GCC: progress, caveats, and evaluation. Proceedings of GCC Developers' Summit 2007.

[9] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, Wen-mei W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. Proceedings of the 22nd Annual International Symposium on Computer Architecture 1995. No. 22. P. 138—149.

[10] David I. August, Wen-mei W. Hwu, Scott A. Mahlke. A Framework for Balancing Control Flow and Predication International Symposium on Microarchitecture 1997. P. 92—103.

[11] David I. August, John W. Sias, Jean-Michel Puiatti, Scott A. Mahlke, Daniel A. Connors, Kevin M. Crozier, Wen-mei W. Hwu. The Program Decision Logic Approach to Predicated Execution. Proceedings of the 22nd Annual International Symposium on Computer Architecture 1999. P. 208-219.

Support for Conditional Execution in Selective Scheduling

Dmitry Melnik <dm@ispras.ru>
Alexandr Monakov <amonakov@ispras.ru>

Abstract. Conditional execution is a hardware feature available in some CPUs that allows assigning instruction a predicate so that instruction will only execute if the predicate is true. In this paper we propose a method for supporting conditional execution in the instruction scheduler and discuss the advantages of this approach compared to a stand-alone optimization pass that precedes the instruction scheduling. We have implemented the proposed optimization in the selective scheduling in GCC compiler. Experimental results show an average improvement of 2% (and up to 16% on selected tests) for SPECFP part of SPEC CPU2000 benchmark.

Keywords: instruction scheduling, conditional execution, predication, GCC.

Метод автоматического восстановления переменных из трассы исполнения программы[1]

М.А. Климушенкова, В.А. Макаров,
maria.klimushenkova@ispras.ru, Vladimir.Makarov@orimtech.com

Аннотация. В работе описывается метод восстановления локальных переменных из трассы исполнения программы. Метод использует одну из схем анализа потоков данных – достигающие определения. В ней также рассматриваются существующие подходы к решению задачи восстановления переменных.

1. Введение

В целях обеспечения безопасности использования программных средств, разработанных сторонними исполнителями, часто возникает необходимость анализировать их исполняемые файлы. Это очень сложный и трудоемкий процесс. Для облегчения работы аналитика разработано много различных инструментальных средств. Применение описываемого алгоритма повышает уровень абстракции данных, что позволяет аналитику сконцентрироваться на логике обработки данных, а не на особенностях их представления в коде.

2. Обзор работ в области восстановления переменных

В настоящее время существует не так много работ в области восстановления переменных и их типов. В работах [1], [2] и [3] используется схожий подход при выделении локальных переменных. Он основан на анализе потоков данных, вычислении достигающих определений и построении def-use chains. При этом обработка в них происходит не отдельными инструкциями, а базовыми блоками.

Все рассмотренные решения задачи восстановления локальных переменных в качестве входных данных используют ассемблерный код программы. Описываемый метод предполагает анализ трассы исполнения программы вместо анализа ее статического кода. В процессе выполнения программы могут быть пройдены не все ветки, и некоторые инструкции могу остаться

[1] Работа поддержана грантом РФФИ (11-07-00353).

необработанными. Несмотря на это, использование трассы имеет ряд следующих преимуществ. Исходный код программы не всегда может быть доступен, а анализ ассемблерного кода, полученного при дисассемблировании исполняемого файла, осложняется применяемыми методами обфускации. При динамическом анализе может быть использована информация не только о типах команд и их аргументах, но и о значениях, принимаемых элементами памяти.

3. Описание предлагаемого метода

Любая программа оперирует ограниченным количеством элементов памяти. Один и то же регистр может последовательно представлять разные переменные. Предлагаемый метод позволяет вычислить время жизни переменных и их размещение.

Анализируются только регистры общего назначения и ячейки памяти, принадлежащие стеку (предполагается, что каждая ячейка в статической памяти является отдельной глобальной переменной). Выделение параметров, передаваемых в функцию, является отдельной задачей и в данной работе не рассматривается.

Алгоритм последовательно анализирует все принадлежащие функции инструкции. Они могут *определять* элементы памяти, то есть присваивать им новые значения, или просто использовать их. Для каждой инструкции проходим все элементы, которые она использует или изменяет. Если инструкция определяет элемент, то создаем новое определение, а также новую переменную, и делаем изменения в карте достигающих определений.[4] Если инструкция использует элемент, то из карты достигающих определений узнаем, какой переменной он соответствует. Далее вносим информацию в таблицу соответствия элемент-переменная. Из этой таблицы мы можем узнать, какую переменную представлял тот или иной элемент на каждом шаге.

Будем считать, что определение достигает произвольной точки, если на пути от определения до этой точки элемент не переопределяется. Следует так же раскрыть понятия «определение элемента» и «использование элемента».

Каждая инструкция может создавать четыре типа зависимостей [5]:

- Kill – возникает при изменении значения элемента на новое, не зависящее от других элементов.
- Check – возникает при проверке значение элемента.
- Update – возникает при изменении значений элементов.
- Get/Set – возникает при чтении из памяти/записи в память, во вход зависимости включается элементы от которых зависит адрес.

Выходные элементы зависимости Kill получают новые значения, таким образом, для них возникают новые определения. Входные элементы

зависимости Check не изменяют своего значение, то есть используются. Для зависимости Update возможны три варианта: элемент присутствует только на входе, то есть его значение не изменяется – происходит использование; элемент присутствует и на входе и на выходе зависимости, то есть его значение изменяется, также происходит использование; элемент присутствует только на выходе, то есть получает новое значение – возникает новое определение. Ситуации для зависимостей Get/Set схожи. Элементы на выходе получают новые значения – возникают новые определения, а элементы на входе только используются.

Использование трассы в качестве входных данных для алгоритма позволяет обработать несколько проходов исполнения функции. При анализе каждого прохода происходит уточнение информации, полученной на предыдущих итерациях. Если выполнение происходило по разным веткам, то возможно возникновение ситуации, изображенной на рис. 1.

При первом проходе для регистра eax будет создана переменная var1, которая далее используется в команде сложения. При втором проходе для регистра eax будет создана другая переменная var2, на следующем шаге нужно регистру eax поставить в соответствие переменную var2, но он уже соответствует переменной var1. Необходимо объединить переменные var1 и var2 в одно множество, представляющее собой одну переменную.

Последовательно обработав все проходы функции, мы получаем информацию о разбиении элементов памяти, используемых ее командами, на отдельные переменные.

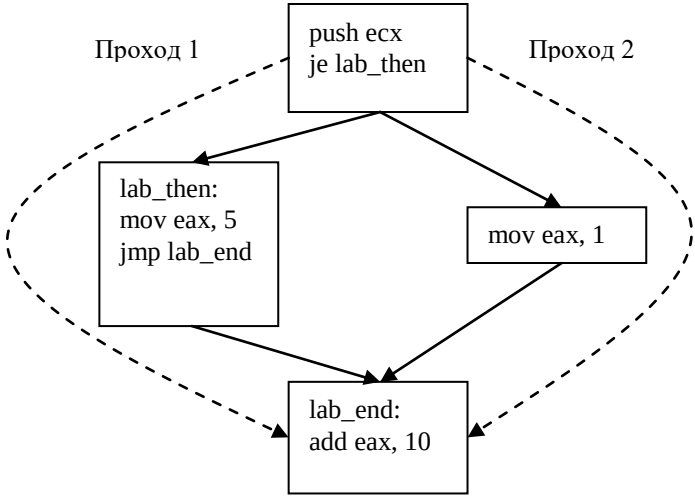


Рис 1. Варианты исполнения программы.

4. Заключение

В данной работе был предложен метод решения двух задач: задачи разделения элементов памяти на локальные переменные и задачи слияния информации, полученной при анализе нескольких проходов функции. Он реализован в виде плагина к среде TrEx.

В ходе дальнейших исследований планируется разработать алгоритм, который на основе информации, полученной при динамическом анализе, будет восстанавливать базовые и производные типы данных.

Список литературы

[1] Cifuentes, C. Reverse Compilation Techniques, PhD thesis, Queensland University of Technology, 1994.

[2] Долгова К. Н., Чернов А. В. Автоматическое восстановление типов в задаче декомпиляции. Программирование, номер 2, журнал Российской академии наук, Москва - 2009.

[3] A. Mycroft. Type-based decompilation. European Symp. on Programming, 1576:208 – 223, 1999.

[4] Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы. Принципы, технологии и инструментарий. Второе издание. «Вильямс» Москва - 2011

[5] В.А. Падарян, А.И. Гетьман, М.А. Соловьев. Программная среда для динамического анализа бинарного кода. Труды Института Системного Программирования. Том 16. 2009. стр. 51-72.

Детерминированное воспроизведение процесса выполнения программ в виртуальной машине[1]

Павел Довгалюк
Pavel.Dovgaluk@ispras.ru

Аннотация. В статье описывается разработка технологии, позволяющей записывать и воспроизводить сценарии выполнения программ в виртуальной машине. Данная технология позволяет выполнять детерминированную отладку приложений, а также должна в дальнейшем лечь в основу реализации различных механизмов динамического анализа программ (в том числе снятия трассы с выполняющейся программы) и реверсивной отладки.

1. Введение

Зачастую программисты в своей работе сталкиваются с ошибками, которые трудно выявить традиционными методами. Например, проявления ошибок могут пропадать в режиме отладки, возникать от случая к случаю, быть результатом сложного взаимодействия потоков в многопоточных приложениях, или требовать сложной настройки окружения, выполняемой перед каждым запуском. Во всех этих случаях процесс отладки мог бы быть значительно упрощен, если можно было бы автоматически записать сценарий, в котором ошибка проявляется, а лишь затем приступить к отладке, многократно проигрывая данный сценарий в симуляторе.

Это становится возможным при использовании механизмов записи недетерминированных событий, происходящих в отлаживаемой системе и последующем их детерминированном воспроизведении.

2. Методы детерминированного воспроизведения программ

Большинство их существующих на настоящий момент технологий детерминированного воспроизведения процесса выполнения программ ограничивают сферу своего применения поддержкой определенной операционной системы, записью только одного процесса, слабой поддержкой

[1] Работа поддержана грантом РФФИ (11-07-00353)

многопоточности, небольшим подмножеством записываемых системных вызовов [[1]], [[2]], [[3]], [[4]].

Из известных решений только одно основано на записи журнала недетерминированных событий в процессе симуляции виртуальной машины – это симулятор VMWare [[5]]. Для этого симулятора реализованы механизмы реверсивной отладки, однако, т.к. он является программой с закрытым исходным кодом, добавить в него связанную с динамическим анализом кода функциональность не представляется возможным. Кроме того, он симулирует только одну аппаратную платформу – i386.

3. Описание предлагаемого подхода

Для реализации технологии детерминированного воспроизведения сценариев выполнения программ был выбран симулятор Qemu. Данный симулятор поддерживает множество архитектур симулируемых машин (i386, ARM, MIPS, PowerPC и т.д.), используется в составе SDK для платформы Android, а также имеет открытый код, что позволяет вносить в него изменения, необходимые для динамического анализа программ.

3.1. Структура симулятора

Симулятор Qemu имеет структуру, показанную на рис. 1Рис. 1.

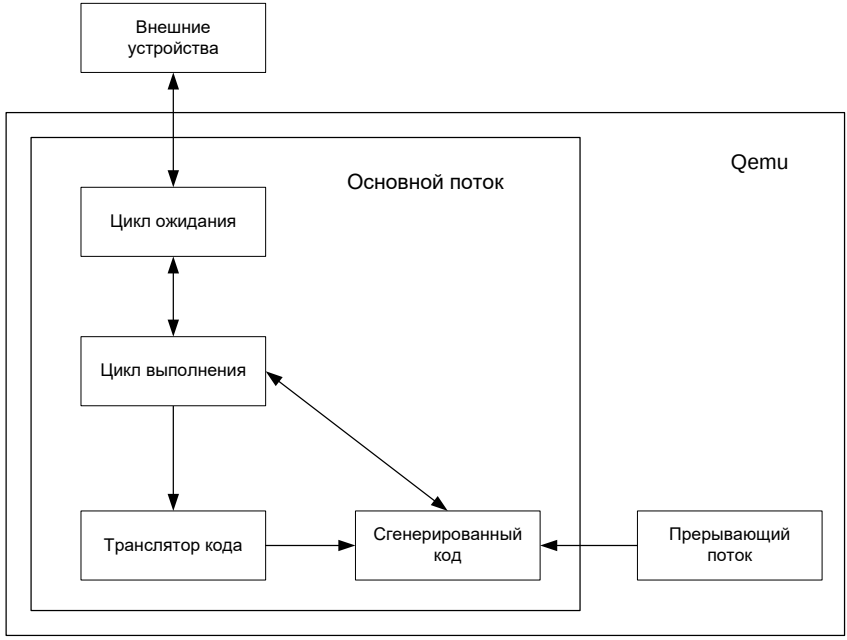


Рис. 1. Структура симулятора Qemu

Цикл выполнения в Qemu производит трансляцию кода симулируемой машины и исполнение сгенерированного кода, обрабатывая небольшую его часть в каждой итерации.

Параллельно с этим работает прерывающий поток, останавливающий выполнение кода с определенной периодичностью. При этом цикл выполнения прерывается и происходит возврат в цикл ожидания.

В цикле ожидания симулятор сначала проверяет состояние внешних устройств и интерфейсов управления симулятором, а затем возвращает управление в цикл выполнения.

Так как прерывающий поток не синхронизирован с основным, детерминированное воспроизведение моментов возникновения прерываний таймера невозможно без переработки этого механизма.

Кроме того, Qemu взаимодействует с внешними устройствами, включая мышь, клавиатуру, сетевую карту, аппаратные таймеры. Через эти внешние устройства в Qemu могут приходить сообщения, которые не являются детерминированными и, следовательно, их обработчики также должны быть доработаны.

3.2. Запись журнала событий

Для того чтобы воспроизводить процесс выполнения программ в виртуальной машине, была реализована запись всех недетерминированных событий в специальный журнал.

События, записываемые в журнал, делятся на синхронные и асинхронные. Синхронные события вызываются действиями, выполняемыми основным потоком (выполнение инструкции, итерации цикла ожидания, чтение часов). Асинхронные события приходят в симулятор извне в произвольный момент времени (прерывание таймерного потока, нажатие клавиши на клавиатуре, движение мыши, приход сетевого пакета).

3.2.1. Выполнение очередной инструкции

Выполнение инструкции является внутренним детерминированным событием. Однако, для того, чтобы корректно воспроизводить моменты возникновения недетерминированных событий (внутренний таймерный поток, события от внешних устройств), необходимо учитывать количество выполненных инструкций.

Для подсчета инструкций был изменен модуль, отвечающий за трансляцию симулируемого кода в код симулирующей машины. После кода, соответствующего каждой инструкции, выполняется код, отвечающий за увеличение счетчика инструкций.

3.2.2. События от часов

В процессе своей работы симулятор осуществляет считывание показаний часов реального времени как для своей работы, так и для передачи в виртуальную машину.

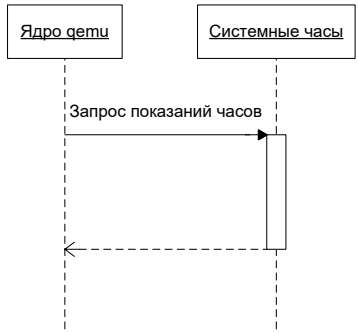


Рис. 2. Получение показаний часов в исходной версии Qemu

Чтобы показания часов реального времени, передаваемые внутрь симулируемой системы, не изменились при воспроизведении поведения системы, в модуль, осуществляющий работу с аппаратными часами, были внесены изменения, позволяющие записывать считанные показания часов в журнал.

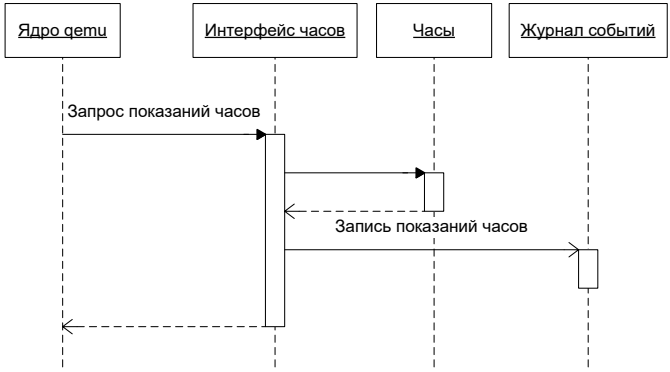


Рис. 3. Запись показаний часов в журнал

При воспроизведении журнала событий вместо считывания показаний аппаратных часов, выполняется чтение их из журнала. Так как воспроизведение является детерминированным, операция чтения происходит в тот же момент (относительно позиции в журнале), что и при записи.

Однако, в случае выхода симулятора в цикл ожидания из цикла выполнения (например, при отладке), могут выполняться операции чтения времени, которых не было в исходном потоке событий. Для того, чтобы они завершались успешно, при воспроизведении журнала выполняется кэширование показаний часов. Таким образом, в цикле ожидания будут считываться одни и те же показания, пока выполнение симулируемого кода не продолжится.

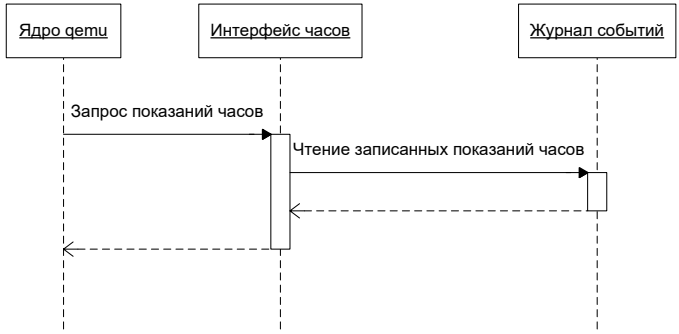


Рис. 4. Чтение показаний часов из журнала

3.2.3. Прерывание выполнения симулируемого кода с помощью таймера

Для решения проблемы с синхронизацией потоков внутри Qemu, было решено зафиксировать точки в коде основного потока, в которых он может взаимодействовать с прерывающим потоком. Таким образом, в процессе воспроизведения журнала событий появится возможность точно восстанавливать моменты этих взаимодействий.

Несколько точек возможных прерываний были размещены в циклах ожидания и выполнения, а также был модифицирован код транслятора для добавления таких точек перед выполнением каждой инструкции. Таким образом становится возможным детерминированное воспроизведение поведения системы даже при пошаговом выполнении кода.

3.2.4. События от периферии

Запись и воспроизведение событий от мыши, клавиатуры и сетевой карты отличается от предыдущих видов событий тем, что они инициируются извне. Поэтому запись информации о них в журнал производится непосредственно в момент возникновения соответствующего события.

Так как основные действия в Qemu, а также запись в журнал, выполняются в одном потоке, сообщения от периферийных устройств будут обрабатываться строго между какими-либо другими событиями, не пересекаясь с ними по времени. Поэтому при воспроизведении журнала (при поиске очередного синхронного события) может встретиться такое асинхронное событие (например, нажатие клавиши), которое может быть тут же обработано.

3.3. Детерминированное воспроизведение журнала событий

Технология детерминированного воспроизведения журнала событий заключается в выполнении кода симулятора и считывании необходимых для воспроизведения программы данных из журнала:

- Считывание показаний часов из журнала, когда их запрашивает какой-либо код. Если соответствующих показаний нет – возвращается кэшированное значение.
- Передача сообщений от клавиатуры, мыши, звуковой и сетевой карт в тот момент, когда они считываются из журнала в соответствующие обработчики.
- Обновление внутреннего счетчика команд, когда происходит выполнение очередной инструкции.

Чтобы воспроизведение журнала было детерминированным, оно должно начинаться с того же самого состояния симулируемой системы, что и запись. Состояние системы включает в себя состояния всех симулируемых устройств, включая образы используемых дисков.

Остальные устройства инициализируется при старте симулятора и, поэтому, их состояния не отличаются при записи и воспроизведении.

3.4. Фрагмент журнала событий

События в журнал записываются последовательно одно за другим. Какой-либо индексации не производится, так как для целей воспроизведения журнала необходимо лишь последовательное чтение и исполнение недетерминированных событий, записанных в журнал.

Ниже приведен пример нескольких событий, записываемых с начала журнала:

1. Системное время
2. Часы 1
3. Часы 0
4. Часы 2

5. Таймер
6. Цикл ожидания 1
7. Цикл ожидания 0
8. Цикл ожидания 2
9. Часы 2
10. Часы 1
11. Цикл ожидания 1
12. Таймер
13. Выполнение инструкции
14. Цикл ожидания 0
15. Цикл ожидания 2
16. Часы 2
17. Часы 1
18. Цикл ожидания 1
19. Выполнение инструкции
20. Выполнение инструкции
21. Выполнение инструкции
22. Выполнение инструкции
23. Выполнение инструкции

В данный фрагмент журнала попали несколько показаний различных часов, а также значения системного времени. Кроме того, происходило выполнение цикла ожидания. В нем расставлено 3 контрольных точки, позволяющих управлять воспроизведением фрагментов цикла (инициирование прерываний и обработка внешних событий) в том же порядке, в котором они выполнялись при записи.

Также в журнале присутствует несколько событий «выполнение инструкции». Здесь они показаны по отдельности, однако запись их в журнал производится группой – одна запись соответствует целой последовательности выполненных инструкций.

3.5. Контроль корректности воспроизведения журнала

Для того чтобы контролировать, является ли процесс воспроизведения детерминированным, при считывании событий из файла журнала проверяется следующее ограничение: в момент, когда ожидается наступление события «выполнение инструкции» не должно возникать ни события от часов, ни события «цикл ожидания».

Если это ограничение нарушается, это означает, что ход симуляции был нарушен. В этом случае выдается сообщение об ошибке и процесс воспроизведения останавливается. Это позволяет контролировать отсутствие

ошибок в механизме воспроизведения и выполнять отладку при их возникновении.

3.6. Замедление работы симулятора при записи журнала событий

Так как при симуляции с включенной записью (и воспроизведением) журнала выполняются дополнительные действия (например, запись в журнал) по сравнению с нормальным режимом работы, происходит уменьшение скорости симуляции.

Для измерения степени замедления рассматривался сценарий загрузки Windows XP до появления графического интерфейса.

- Загрузка Windows XP без записи журнала – 80 секунд.
- Загрузка Windows XP с записью журнала – 520 секунд.
- Загрузка Windows XP при воспроизведении журнала – 606 секунд.

Такое замедление (менее чем в 10 раз) не является критичным для большинства приложений, что позволяет использовать для них данный механизм.

Объем файла с записанным журналом этого сценария составил 18 Мегабайт.

4. Детерминированная отладка с использованием журнала событий

Детерминированная отладка – это способ поиска ошибок в недетерминированных приложениях, при котором недетерминированность устраняется с помощью записи сценария работы системы (или программы) в журнал. Разработанная технология позволяет выполнять детерминированную отладку недетерминированных приложений следующим образом:

1. Тестировщик записывает сценарий при выполнении которого проявляется дефект в тестируемой программе в журнал, а затем передает этот журнал вместе с образами дисков системы разработчику. Здесь сценарий выступает не только в роли исходных данных для отладки, но и в роли описания способа воспроизведения дефекта.
2. Разработчик может неоднократно проигрывать полученный сценарий в симуляторе, анализируя причины появления дефекта. Разработчику не нужно настраивать сложное окружение системы так же, как это делал тестировщик, так как все особенности взаимодействия с этим окружением уже записаны в журнал.

Таким образом, отладка недетерминированных приложений с применением разработанной технологии становится детерминированной, что позволяет

сократить время, затрачиваемое разработчиками на локализацию дефектов в программе, а тестировщиками – на описание процесса их воспроизведения.

5. Заключение

Таким образом, были получены следующие результаты:

1. Разработана технология детерминированного воспроизведения, реализованы механизмы записи и воспроизведения журнала недетерминированных событий в симуляторе Qemu.
2. Данная технология была реализована для платформ x86 и x86_64, что позволяет выполнять отладку приложений, работающих под ОС для этих платформ (в частности, Windows и Linux), а также вновь создаваемых операционных систем.
3. Запись событий производится для системы целиком, включая пользовательские процессы и внутренние механизмы операционной системы, установленной в виртуальной машине, что позволяет учесть все межпроцессные взаимодействия.
4. В журнал событий записываются сетевые пакеты, полученные через внешние интерфейсы. Таким образом, можно анализировать и отлаживать приложения, работающие с сетью.
5. Реализована возможность многократной отладки сложных сценариев воспроизведения ошибок с помощью их детерминированного воспроизведения.
6. Также технология предоставляет возможности для реализации других механизмов динамического анализа программ (в частности, снятия трассы их выполнения для последующего анализа) и реверсивной отладки.
7. Детерминированное воспроизведение было протестировано как для изолированной виртуальной машины, так и работающей в сетевом окружении. При тестировании использовались виртуальные машины с установленными Windows XP и Linux. Для большинства сетевых приложений вносимое механизмом сохранения журнала замедление является незначительным, поэтому разработанная технология может быть использована для отладки и анализа таких приложений.

Дальнейшее развитие технологии детерминированного воспроизведения планируется в следующих направлениях:

1. Реализация технологии детерминированного воспроизведения для платформ, отличных от i386 (в частности, ARM), что позволит выполнять детерминированное воспроизведение сценариев работы, записанных на симуляторах мобильных устройств.
2. Одно из других важных направлений – разработка и реализация технологии, позволяющей выполнять реверсивную отладку

приложений в симуляторе с использованием детерминированного воспроизведения недетерминированных событий. Средства реверсивной отладки позволяют отладчику возвращаться от места проявления ошибки к месту в программе, где эта ошибка возникает на самом деле (например, запись некорректного значения в переменную или чтение неинициализированной ячейки памяти).

3. Также будет проводиться работа как по уменьшению объема журнала событий, так и по уменьшению накладных расходов при его создании, так как для некоторых высокопроизводительных сетевых приложений создаваемое замедление оказывается критичным.
4. Четвертым направлением является расширение поддержки внешних устройств при записи журнала. В частности, работа с внешними устройствами, подключаемыми через USB.

Список литературы

[1] J. Choi and H. Srinivasan, “Deterministic replay of java multithreaded applications”, In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, Aug. 1998, pages 48-59.

[2] H. Patil, C. Pereira, M. Stallcup, G. Lueck, J. Cownie, “PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs”, Proceedings of the 8th annual IEEEACM international symposium on Code generation and optimization (2010), pages 2-11.

[3] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, “Flashback: a lightweight extension for rollback and deterministic replay for software debugging”, USENIX 2004 Annual Technical Conference, Pp. 29–44 of the Proceedings

[4] ReplayEngine. Record and replay debugging race conditions and deadlocks in Linux applications, 2011. http://www.roguewave.com/products/totalview-family/replayengine.aspx

[5] Better Software Development with Replay Debugging, 2009. http://www.replaydebugging.com/

Оценка производительности программного обеспечения в виртуализованном окружении на основе атомарных тестов

П.А. Клеменков
parser@cs.msu.su

Аннотация. В настоящее время все больше растет интерес к использованию платформ виртуализации (VMWare, XEN и др.) в различных сферах, включая консолидацию серверов, организацию хостинга и облачные вычисления. Производительность приложения в виртуальной машине может очень сильно отличаться от производительности вне виртуализованного окружения из-за взаимодействий с гипервизором и другими виртуальными машинами.

В этой статье описывается обобщенный подход к оценке требуемых программному обеспечению ресурсов при переносе его в виртуализованное окружение. Основной принцип предложенного подхода заключается в представлении сложной нагрузки в виде комбинации простых задач и замены этих простых задач на синтетические атомарные тесты. Оценка производительности атомарных тестов в среде виртуализации и вне нее позволяет определить накладные расходы на виртуализацию.

Ключевые слова: виртуализация; оценка производительности по; атомарные тесты; xen

1. Введение

Виртуализация и автоматизация – это основные составляющие современных вычислительных центров, позволяющие создавать более динамичную IT-инфраструктуру. Виртуализация помогает изолировать владельца вычислительных ресурсов от владельца программного обеспечения, позволяя стандартизировать и автоматизировать процессы настройки, управления и мониторинга. Скрывая детали распределения ресурсов от пользователей, виртуализация способствует оптимальному разделению вычислительных мощностей между приложениями, выполняющимися в разных виртуальных машинах, благодаря возможности быстрого перепрофилирования ресурсов по запросу. Такие функции позволяют лучше удовлетворять потребности приложений и более эффективно отвечать на изменяющиеся требования бизнеса.

В таких условиях необходимыми становятся удобные и точные модели оценки производительности, которые позволяют консолидировать приложения и предоставлять необходимые для их выполнения ресурсы. Для того чтобы понять, какие нагрузки и на каких серверах можно объединять, требуется провести планирование мощностей и анализ существующей нагрузки. В простейшем случае можно собрать информацию о пиковых нагрузках, создаваемых множеством приложений, и оценить совместные требования к ресурсам как сумму пиковых значений. Однако такой подход может привести к выделению излишних мощностей, т.к. он не учитывает эффекты, возникающие при разделении ресурсов между взаимодополняющими программами.

Более перспективный и точный подход основывается на трассировке поведения приложения [1], [2]. Метод заключается в создании профиля использования ресурсов за некоторый период времени (обычно 3-6 месяцев). Затем этот профиль используется для планирования мощностей и распределения нагрузки при консолидации [3], [4]. Суть подхода заключается в том, что поведение приложения в будущем можно попытаться определить на основе наблюдений за создаваемой им нагрузкой в прошлом.

Однако планирование ресурсов при переходе в виртуализованное окружение усложняется возникновением дополнительных накладных расходов, вызванных взаимодействием с гипервизором. Эти накладные расходы зависят от типа виртуализации и конкретной реализации решения [5], [6], [7]. Зачастую дополнительные накладные расходы при использовании ЦПУ прямо пропорциональны объему выполняемого ввода-вывода [8], [9]. Существующие решения на основе трассировки имеют возможности масштабировать оценку на основе заданного множителя, чтобы учитывать различия в аппаратном обеспечении, однако такой подход не всегда применим при переходе в среду виртуализации, где объем накладных расходов оценить не так просто.

2. Оценка производительности на основе «микрозамеров»

Существует и другой подход к оценке производительности приложений при переносе их в виртуализованную среду [10]. Основу подхода составляют два процесса:

1. Проведение заданного набора микрозамеров для профилировки накладных расходов на виртуализацию данной платформы.
2. Создание модели, связывающей профиль потребления ресурсов вне среды виртуализации с профилем внутри среды, на основе регрессионного алгоритма. Полученная модель позволяет предсказывать потребление ресурсов любым приложением на данной платформе.

Во время проведения каждого микрозамера собирается информация о потреблении ресурсов для построения профиля, используемого как обучающий набор для модели. Вне среды виртуализации собирается информация о метриках, связанных с потреблением ресурсов ЦПУ, сетевой активности и дисковом вводе-выводе. Далее, для нахождения закономерностей между потреблением ресурсов в среде виртуализации и вне нее, используются профили, собранные при проведении микрозамеров на «голом железе» и в виртуализованном окружении. Используя данные профили, формулируется система уравнений, выражающая потребление ресурсов ЦПУ как линейную комбинацию различных метрик:

$$\begin{aligned} U_{virt}^{1} &= c_0 + c_1 * M_1^1 + c_2 M_2^1 + \ldots + c_k * M_k^1 \\ U_{virt}^{2} &= c_0 + c_1 * M_1^2 + c_2 M_2^2 + \ldots + c_k * M_k^2 \\ &\ldots \end{aligned} \quad (1)$$

где

- M_i^j - это значение метрики M_i, измеренное во временной интервал j вне среды виртуализации
- U_{virt}^j - это мера потребления ресурсов ЦПУ в среде виртуализации, измеренная во временной интервал j

Пусть $c_0^{virt}, c_1^{virt}, \ldots, c_k^{virt}$ являются приближенным решением системы уравнений (1), тогда объем потребляемых ресурсов ЦПУ для произвольного приложения в виртуализованном окружении можно оценить как:

$$\hat{U}_{virt}^j = c_0^{virt} + \sum_{i=1}^{k} M_i^j \cdot c_i^{virt} \quad (2)$$

Для получения приближенного решения $c_0^{virt}, c_1^{virt}, \ldots, c_k^{virt}$ можно применить один из известных регрессионных методов. Одним из самых известных является метод наименьших квадратов, который минимизирует ошибку:

$$e = \sqrt{\sum_j \left(\hat{U}_{virt}^j - U_{virt}^j\right)^2}$$

Набор коэффициентов $c_0^{virt}, c_1^{virt}, \ldots, c_k^{virt}$ и определяют модель, описывающую связь между потреблением ресурсов ЦПУ в среде виртуализации и потреблением ресурсов сервера вне нее.

Авторы метода утверждают, что данный подход достаточно полно характеризует различные накладные расходы на виртуализацию, а медиана ошибки предсказания модели на тестах RUBiS и TPC-W составляет менее 5%.

3. Ограничения метода «микрозамеров»

Описанный выше подход показывает достаточно хорошие результаты на избранных нагрузках. «Микрозамеры» предлагают решение задачи оценки производительности ПО в виртуализованном окружении в общем, но зачастую важнее делать предположения, основываясь на преобладающих характеристиках нагрузки. Очевидно, что некоторые приложения производят интенсивный ввод-вывод, некоторые большую часть времени исполняются в пространстве ядра, производя системные вызовы и т.д. Однако каждая программа представляет собой совокупность различных активностей. Эти активности воздействуют друг на друга и на производительность всего приложения. С другой стороны, одна и та же активность может вызывать различные события в разных гипервизорах. Именно поэтому так важно учитывать эти особенности при оценке производительности.

Другой проблемой метода «микрозамеров» является природа тестов. Каждый микрозамер по своей сути является синтетической нагрузкой. Поэтому выбор подходящего микрозамера является сложной задачей, которую трудно решить с удовлетворительной точностью.

4. Метод разделения сложной нагрузки на атомарные тесты

Описываемый мной подход основан на предположении о том, что некоторые типы сложных нагрузок могут быть представлены как совокупность более простых активностей. Пусть T - время выполнения некоторой нагрузки. Это может быть создание процесса, обработка запроса на ввод-вывод и т.д. T может быть выражена как линейная комбинация нескольких атомарных активностей операционной системы (обработка исключения, переключение контекста, обработка ошибки страницы и т.п.):

$$T = \sum_{i=1}^{N} \alpha_i r_i + U + F\ \bar{r}\ , \qquad (3)$$

где

- r_i - время выполнения атомарной активности i;
- $\bar{r}$ - вектор времен выполнения всех атомарных активностей;
- α_i - коэффициенты;
- U - константа, выражающая неучтенные значения;

- $F(\bar{r})$ - отклонение модели. Функция также выражает взаимовлияние атомарных активностей r_i.

Для того чтобы воспользоваться уравнением (3) необходимо измерить все r_i. Это довольно сложная задача, несмотря на то, что существует целый ряд подходящих инструментов [11], [12]. Еще сложнее произвести измерения с удовлетворительной точностью. Для практического применения необходимо упростить модель.

Для упрощения предлагается заменить атомарные активности r_i так называемыми атомарными тестами. Атомарный тест – это простое приложение, которое эмулирует поведение атомарной активности операционной системы. Для примера, мы можем измерить время переключения контекста с помощью приложения, создающего два процесса, взаимодействующих через неименованный канал. Обозначим время выполнения атомарного теста через t_i. Далее выразим t_i:

$$t_i = \beta_i r_i + u_i \tag{4}$$

где

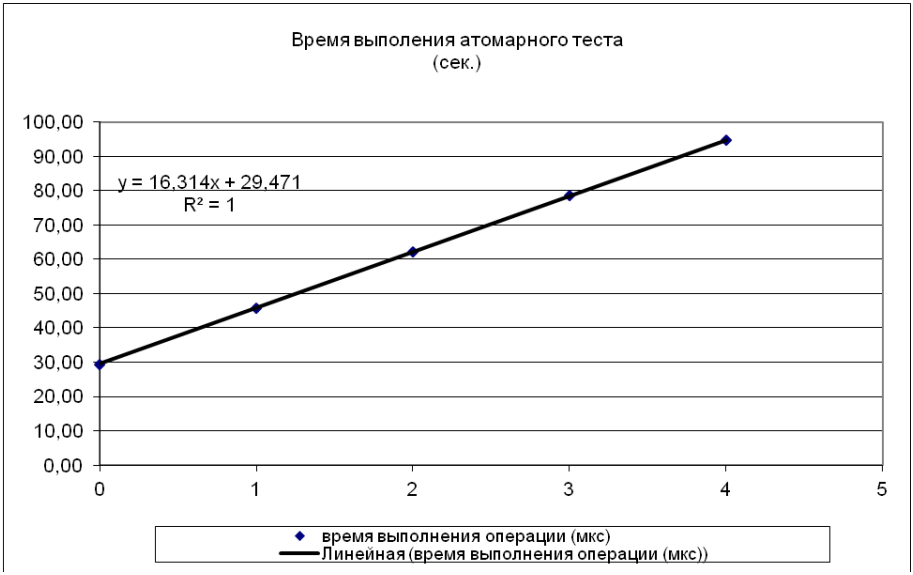
- β_i - коэффициенты;
- u_i - константы, выражающие неучтенные эффекты.

Теперь докажем линейность выражения (4). Для этого внесем некоторые изменения в гипервизор и выполним тест в виртуализованном окружении. Для того чтобы повлиять на t_i, можно реализовать в мониторе виртуальных машин искусственную задержку. Время выполнения атомарного теста t_i должно расти пропорционально:

$$\begin{aligned} t_i^1 &= \beta_i\left(r_i + 1\cdot\Delta\right) + u_i \\ t_i^2 &= \beta_i\left(r_i + 2\cdot\Delta\right) + u_i \\ t_i^3 &= \beta_i\left(r_i + 3\cdot\Delta\right) + u_i \\ &\cdots \\ t_i^k &= \beta_i\left(r_i + k\cdot\Delta\right) + u_i \end{aligned} \tag{5}$$

В качестве доказательства линейности выражения (4) была разработана упомянутая выше программа взаимодействия двух процессов через неименованный канал. Выполнение программы осуществлялось в виртуальной машине под управлением гипервизора XEN. Задержка была реализована в виде простого цикла в функции, обновляющей контрольный

регистр CR3 при переключении контекста. Линейная зависимость времени выполнения атомарного теста от числа итераций цикла показана на графике:



5. Оценка производительности по методу атомарных тестов

Теперь можно выразить r_i из формулы (4) и сделать соответствующие замены в (3):

$$T = \sum_{i=1}^{N} \frac{\alpha_i}{\beta_i} t_i - \sum_{i=1}^{N} \frac{\alpha_i}{\beta_i} u_i + U + F(\bar{r}) = \sum_{i=1}^{N} \gamma_i t_i + U^* + F(\bar{r}) \qquad (6)$$

Далее уравнение (6) можно преобразовать к более удобному, для применения на практике, виду:

$$T = T_0 + \sum_{i=1}^{N} \gamma_i \Delta t_i + F^*(\bar{r}) \qquad (7)$$

Практическая ценность формулы (7) заключается в том, что время выполнения сложной нагрузки T_0 можно измерить только один раз, а затем использовать его для оценки времени работы этой нагрузки в виртуализованном окружении T, подставляя характеристические коэффициенты γ_i используемого гипервизора. Норма функции $\left\| F^*(\bar{r}) \right\|$

выражает отклонение измеренного времени T_0 от времени, предсказанного моделью.

Также необходимо отметить некоторые особенности приведенной модели:

- Условие (4) должно выполняться всегда, т.к. атомарные тесты обязаны линейно зависеть от атомарных активностей.
- Система должна быть полной относительно выражения (3). Это значит, что набор атомарных тестов должен полностью покрывать рассматриваемую нагрузку. Отследить наличие неучтенных атомарных активностей можно по изменениям значения константы U.

Следует отметить, что значение константы U необходимо подбирать для каждого гипервизора в отдельности, ограничивая норму функции $\left\| F^*(\bar{r}) \right\|$.

6. Заключение

Перенос программного обеспечения в виртуализованное окружение всегда приводит к снижению общей производительности. Поэтому важной задачей является оценка требуемых приложениями ресурсов перед их консолидацией. В этой статье был представлен подход к оценке производительности ПО, основанный на атомарном тестировании. Суть подхода заключается в том, что любая сложная нагрузка может быть представлена, как совокупность простых активностей. А предсказать время работы приложения в виртуальной машине, можно собрав данные о производительности атомарных тестов в используемом виртуализованном окружении и вне него.

На данном этапе исследования сформулированной задачи, была представлена теоретическая модель, подтвержденная некоторыми практическими данными. Дальнейшая работа будет направлена на доказательство справедливости модели и ее уточнение, путем расширения набора атомарных тестов и сбора данных в реальной продуктивной среде.

Список литературы

[1] Gmach D., Rolia J., Cherkasova L., Kemper A.: Capacity Management and Demand Prediction for Next Generation Data Centers. Proceedings of the International IEEE Conference on Web Services, 2007, pp. 43-50.

[2] Rolia J., Cherkasova L., Arlitt M., Andrzejak A.: A Capacity Management Service for Resource Pools. Proceedings of the 5th international workshop on software and performance, 2005, pp. 229-237.

[3] HP Integrity Essentials Capacity Advisor. http://h71036.www7.hp.com/enterprise/cache/262379-0-0-0-121.html. Дата обращения 10.09.2011.

[4] VMware Capacity Planner, http://www.vmware.com/products/capacity-planner/index.html. Дата обращения 10.09.2011.

[5] Barham P., Dragovic B., Fraser K., Hand S., Harris T., Ho A., Neugebauer R., Pratt I., Warfield A.: Xen and the art of virtualization. Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003, pp. 164-177.
[6] King S., Dunlap G., Chen P.: Operating system support for virtual machines. Proceedings of the USENIX Annual Technical Conference, 2003, p. 6.
[7] Sugerman J., Venkitachalam G., Lim B.-H.: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. Proceedings of the General Track: 2002 USENIX Annual Technical Conference, 2001, pp. 1-14.
[8] Cherkasova L., Gardner R.: Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. Proceedings of the USENIX Annual Technical Conference, 2005, p. 24.
[9] Gupta D., Cherkasova L., Gardner R., Vahdat A.: Enforcing Performance Isolation Across Virtual Machines in Xen. Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware, 2006, pp. 342-362.
[10] Wood T., Cherkasova L., Ozonat K., Shenoy P.: Profiling and Modeling Resource Usage of Virtualized Applications. Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware, 2008, pp. 366-287.
[11] McVoy L., Staelin C.: Portable Tools for Performance Analysis. Proceedings of the USENIX Annual Technical Conference, 1996, p. 23.
[12] Staelin C., McVoy L.: Anatomy of a micro-benchmark. Proceedings of the USENIX Annual Technical Conference, 1998, p.13.

Software performance estimation in a virtualized environment based on atomic tests

Pavel A. Klemenkov
parser@cs.msu.su
Lomonosov Moscow State University

Annotation. Virtual Machine (VM) environments are experiencing a resurgence of interest for diverse uses including server consolidation and shared hosting. An application performance in a virtual machine environment can differ markedly from its performance in a non-virtualized environment because of interactions with the underlying virtual machine monitor and other virtual machines.
In this paper, I describe a general approach for estimating the resource requirements of applications when they are transferred to a virtual environment. The core principle of this approach is splitting complex workload into a combination of smaller tasks and replacing these tasks with synthetic atomic tests. Performance evaluation of atomic tests on native hardware and in the virtual machine allows us to define virtualization overhead for the given platform.

Keywords: virtualization; software performance estimation; atomic tests; xen

Применение алгебры подстановок для унификации программ

В.А.Захаров, Т.А. Новикова
{zakh@cs.msu.su} {taniaelf@mail.ru}

Аннотация. Для решения многих задач системного программирования, к числу которых относятся задачи реорганизации программ, деобфускации программ, выявления уязвимостей в программном коде и др., желательно иметь инструментальное средство, позволяющее обнаруживать фрагменты программ, имеющие сходное поведение. Современные средства обнаружения программных клонов позволяют выявлять лишь фрагменты программ, имеющие сходное синтаксическое устройство, поскольку более глубокий семантический анализ программ сталкивается с алгоритмической неразрешимостью проблемы функциональной эквивалентности программ. Для того чтобы избежать алгоритмически трудных задач проверки функциональной эквивалентности, авторы настоящей статьи предлагают воспользоваться более сильным разрешимым отношением эквивалентности программ – логико-термальной эквивалентностью, – введенной в 1972 г. В.Э. Иткиным. В данной статье разработан новый алгоритм проверки логико-термальной эквивалентности программ, основанный на операции вычисления точной нижней грани в решетке конечных подстановок. На основе этого алгоритма авторам статьи удалось также решить задачу логико-термальной унификации программ, которая состоит в построении для двух заданных фрагментов программного кода такой процедуры, которая представляет собой наиболее общую специализацию этих двух фрагментов.

Ключевые слова: программа, обнаружение клонов, эквивалентность, подстановка, унификация, антиунификация.

1. Введение

Результаты статистических исследований, о которых говорится в работах [1], [2], [3], свидетельствуют о том, что суммарный объём клонов (фрагментов программ, имеющих сходное синтаксическое устройство) в больших проектах обычно составляет 7–20%, а в некоторых случаях доходит и до 50%. Наличие большого числа клонов приводит к увеличению стоимости поддержки кода, к возрастанию ресурсов, необходимых для компиляции и хранения скомпилированной программы, а также к увеличению вероятности возникновения новых и распространения существующих ошибок [4,5]. Поэтому от избыточного дублирования кода целесообразно избавляться при

помощи существующих методов реорганизации (рефакторинга) кода, например, при помощи процедурной абстракции, также называемой “Выделением метода” (Extract Method) в терминологии Фаулера [6]. В статье [7] предложен метод процедурной абстракции, позволяющий для заданной совокупности программных клонов построить процедуру, вызов которой заменяет каждый из выделенных клонов. Однако применение этого метода требует предварительного выделения подобных фрагментов программ (клонов). К сожалению, специалисты в области анализа программного обеспечения ещё не выработали общепринятого формального определения понятия клона. В отличие от многих других широко используемых в программировании понятий (например, процедуры, алгоритма) в настоящее время не существует строгого математического определения понятия клона. Все исследователи сходятся в том, что клоном называются фрагменты, имеющие незначительные синтаксические отличия друг от друга, но до сих пор еще не сложилось общего мнения о том, какие именно синтаксические отличия можно признать незначительными.

В обзоре [8] описано большое разнообразие методов выделения клонов на разных уровнях представления программы (текстовое представление, абстрактное синтаксическое дерево, граф потоков управления и данных). Однако при всем разнообразии этих методов они ограничиваются только поверхностным синтаксическим анализом программ и полностью игнорируют проверку семантических (функциональных) свойств программ. Вместе с тем, именно на основании анализа функций, вычислимых программами, можно дать строгое математическое определение отношения подобия фрагментов программ и разработать более изощренные алгоритмы выделения клонов. Задачу выделения программных клонов можно сформулировать, например, так: для пары программ π_1, π_2, реализующих функции $\Phi_1(x)$ и $\Phi_2(x)$, требуется выяснить, существует ли такая программа π_0 (программа-шаблон), реализующая функцию $\Phi_0(x)$, для которой равенства $\Phi_1(x) = \psi_1(\Phi_0(\varphi_1(x)))$ и $\Phi_2(x) = \psi_2(\Phi_0(\varphi_2(x)))$ выполняются для некоторых функций $\psi_i, \varphi_i,\ i = 1,2,$ вычислимых очень простыми программами; эти программы осуществляют инициализацию переменных и последующую специализацию вычисленного результата. В этом случае программу-шаблон π_0 можно выделить в отдельный модуль (процедуру) и заменить обе программы π_1, π_2 соответствующими вызовами этой процедуры.

Нетрудно видеть, что в самой общей постановке задача распознавания клонов всегда имеет вырожденное решение – в качестве шаблона π_0 можно выбрать пустую программу, – и поэтому интерес представляют лишь такие варианты

задачи, в которых на функции инициализации φ_i и специализации ψ_i налагаются дополнительные ограничения. В настоящей статье рассматривается один вариант задачи распознавания клонов, в котором на статус программы-шаблона π_0 претендует одна из двух анализируемых программ π_1, π_2, а в качестве процедуры инициализации переменных разрешается использовать только последовательную композицию операторов присваивания вида $x_1 := t_1; x_2 := t_2; \ldots x_n := t_n$. В этом случае программа-шаблон (например, π_1) становится процедурой, а вторая программа (в данном случае π_2) заменяется вызовом этой процедуры с набором параметров $(t_1, t_2, \ldots, t_n)$. В такой постановке задача распознавания клонов является частным случаем более общей задачи унификации программ.

В задаче унификации выражений E_1 и E_2 требуется отыскать пару наиболее общих эквивалентных примеров $E_1' = E_1\theta_1$ и $E_2' = E_2\theta_2$ этих выражений. Чаще всего эквивалентность примеров E_1' и E_2' предполагает синтаксическое совпадение этих выражений. Интерес к задаче синтаксической унификации был обусловлен, в первую очередь, разработкой и применением метода резолюций в логике предикатов первого порядка [9] и развитием на его основе парадигмы логического программирования. Фактически, алгоритм унификации – это основной механизм вычисления логических программ. Задача унификации изучалась также и для языков более высокого порядка; в этих языках эта задача оказалась алгоритмически неразрешимой (см. [10]). Наряду с задачей синтаксической верификации во многих приложениях интерес представляет также и более сложная задача семантической унификации (или E-унификации); в этом случае эквивалентность выражений E_1' и E_2' предполагает равенство этих выражений в тех или иных аксиоматических теориях. Некоторые результаты исследований задачи семантической унификации в различных логических и алгебраических теориях представлены в работе [11]. В настоящей статье впервые сформулирована и исследована задача семантической унификации последовательных императивных программ. Эта задача состоит в том, чтобы для произвольной пары программ π_1, π_2 вычислить такую пару инициализирующих последовательностей операторов θ_1, θ_2 вида $x_1 := t_1; x_2 := t_2; \ldots x_n := t_n$, для которой последовательные композиции программ $\theta_1; \pi_1$ и $\theta_2; \pi_2$ оказываются эквивалентными.

Очевидно, что алгоритмическая разрешимость задачи унификации программ возможна лишь в том случае, когда разрешимо то отношение эквивалентности

программ, в рамках которого сформулирована эта задача. Известно, однако, что функциональная эквивалентность программ, семантика которых задана в свободных алгебраических системах, неразрешима [12]. Более того, как было показано в статье [13] нерекурсивно всякое невырожденное отношение эквивалентности программ, которое определяется на основании вычислений программ в интерпретациях (алгебраических системах). Поэтому в настоящей статье нами было выбрано одно из наиболее слабых разрешимых неинтерпретационных отношений эквивалентности программ – отношение логико-термальной эквивалентности.

Две программы π_1 и π_2 считаются логико-термально 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. Логико-термальная (л.-т.) эквивалентность была введена в статье [14]. Интерес к л.-т. эквивалентности программ был обусловлен двумя ее свойствами:

1) л.-т. эквивалентность программ π_1 и π_2 влечет функциональную эквивалентность этих программ [14],

2) отношение л.-т. эквивалентности программ разрешимо за полиномиальное время [15].

Было установлено также, что л.-т. эквивалентность – это одно из наиболее слабых рекурсивных отношений эквивалентности программ, аппроксимирующих отношение функциональной эквивалентности.

В настоящей статье представлены два основных результата изучения задачи унификации программ:

1) новый алгоритм проверки л.-т. эквивалентности программ, использующий операции композиции и антиунификации в алгебре конечных подстановок,

2) алгоритм л.-т. унификации программ, разработанный на основе алгоритма проверки л.-т. эквивалентности программ.

Статья состоит из 7 разделов. В разделе 2. введены основные понятия теории конечных подстановок. Во разделах 3 и 4 в терминах теории конечных подстановок определена модель стандартных схем программ и логико-термальная эквивалентность программ. В разделах 5 и 6 описан алгоритм проверки л.т. эквивалентности программ и алгоритм унификации программ. В заключении обсуждаются направления дальнейших исследований задачи унификации программ.

2. Алгебра конечных подстановок

Рассмотрим конечный алфавит, состоящий из множества функциональных символов F, множества предикатных символов P и множества предметных

переменных Var. Множество термов $Term(F,Var)$ – это наименьшее множество, удовлетворяющее следующим двум условиям: 1) $Var \subseteq Term(F,Var)$, и 2) если $f^{(n)} \in F$ и $\{t_1,t_2,\ldots,t_n\} \subseteq Term(F,Var)$, то $f^{(n)}(t_1,t_2,\ldots,t_n) \subseteq Term(F,Var)$. Атомарная формула (или просто атом) – это всякое выражение вида $p^{(m)}(t_1,t_2,\ldots,t_n)$, где $p^{(m)} \in P$, а $t_1,t_2,\ldots,t_n$ – термы. Множество атомарных формул обозначим записью $Atom(F,Var)$. Записи Var_t и Var_A обозначают множество переменных, входящих в терм t и атом A соответственно.

Пусть X и Y – два конечных множества переменных. Подстановкой назовем всякое отображение $\theta : X \to Term(F,Y)$, сопоставляющее каждой переменной из X некоторый терм из $Term(F,Y)$. Множество всех таких подстановок условимся обозначать 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 $\{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$. Если $t_i = x_i$, то пара (связка) $\{x_i/t_i\}$ может быть опущена в этом списке. 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 $t, t \in Term(F,X),$ называется терм $t\theta$, получающийся одновременной заменой в t каждой переменной x_i термом $\theta(x_i)$. Аналогично определяется результат применения подстановки к атому. Выражение $E\theta$, полученное в результате применения к выражению E подстановки θ, будем называть примером выражения E, специализированного подстановкой θ. 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 $\theta : Y \to Y$ называется переименованием.

На множестве подстановок $Subst(X,F,Y)$ определим отношение предпорядка $\prec$ и отношение эквивалентности $\approx$: для пары подстановок θ_1, θ_2 отношение $\theta_1 \prec \theta_2$ выполняется, если существует такая подстановка

$\eta \in Subst(Y,F,Y)$, что $\theta_2 = \theta_1\eta$, и отношение $\theta_1 \approx \theta_2$ выполняется, если $\theta_2 = \theta_1\rho$ для некоторого переименования ρ. Отношение предпорядка $\prec$ индуцирует на множестве классов эквивалентности $Subst(X,F,Y)/\approx$ отношение частичного порядка $\leq$. Частично упорядоченное множество $(Subst(X,F,Y)/\approx, \leq)$ образует решетку, наименьшим элементом которой является класс эквивалентности, порожденный так называемой пустой подстановкой $\{x_1/y_1, x_2/y_2, \ldots, x_n/y_n\}$. Для того чтобы сделать решетку подстановок полной, добавим к множеству подстановок в качестве наибольшего элемента специальную мнимую подстановку τ, удовлетворяющую равенствам $\tau\theta = \theta\tau = \tau$ и $E_1\tau = E_2\tau$ для любой подстановки θ и выражений E_1, E_2. Операция взятия точной верхней грани в этой решетке называется унификацией подстановок и обозначается символом $\uparrow$, а операция взятия точной нижней грани называется антиунификацией подстановок и обозначается символом $\downarrow$. Решетка $(Subst(X,F,Y)/\approx, \leq)$ удовлетворяет условию обрыва убывающих цепей. Далее записи $\theta_1 \uparrow \theta_2$ и $\theta_1 \downarrow \theta_2$ будем использовать для обозначения произвольной подстановки из классов эквивалентности $\theta_1^{\approx} \uparrow \theta_2^{\approx}$ и $\theta_1^{\approx} \downarrow \theta_2^{\approx}$. Также для любой пары подстановок $\theta', \theta' \in Subst(X',F,Y)$ и $\theta'', \theta'' \in Subst(X'',F,Y)$ в том случае, если $X' \cap X'' = \varnothing$, мы будем использовать запись $\theta' \cup \theta''$ для обозначения подстановки η, которая представляет собой теоретико-множественное объединение связок подстановок θ' и θ''. Решетка подстановок была подробно исследована в статьях [16], [17]. В частности, установлено, что операция композиции обладает следующими двумя важными свойствами:

1. свойством монотонности: $\theta_1 \prec \theta_2 \quad \Rightarrow \quad \eta\theta_1 \prec \eta\theta_2$
2. свойством левой дистрибутивности относительно операции антиунификации: $\eta(\theta_1 \downarrow \theta_2) = (\eta\theta_1) \downarrow (\eta\theta_2)$.

Именно эти свойства композиции подстановок обеспечивают корректность преложенного нами алгоритма проверки логико-термальной эквивалентности.

Операции унификации и антиунификации подстановок – это основные алгебраические операции, которые будут использоваться для решения задачи унификации программ. Поэтому в этом разделе мы приведем также два наиболее эффективных алгоритма вычисления этих операций в решетке подстановок.

2.1. Алгоритм унификации подстановок

Описанный ниже алгоритм унификации подстановок был предложен в работе [18]. В том случае, если множества термов из области значений подстановок задаются ориентированными ациклическими графами, этот алгоритм имеет линейную сложность как по времени, так и по объему используемой памяти.

Пусть заданы две подстановки $\theta' = \{x_1 / t'_1, x_2 / t'_2, \ldots, x_n / t'_n\}$ и $\theta'' = \{x_1 / t''_1, x_2 / t''_2, \ldots, x_n / t''_n\}$, принадлежащие множествам $Subst(X, F, Y')$ и $Subst(X, F, Y'')$ соответственно таким, что $Y' \cap Y'' = \varnothing$. Для того чтобы вычислить их унификацию $\theta' \uparrow \theta''$ составим систему термальных уравнений

$$t'_1 = t''_1$$
$$t'_2 = t''_2$$
$$\vdots$$
$$t'_n = t''_n$$

и будем применять в произвольном порядке к этой систем уравнений, до тех пор пока это возможно, следующие 6 правил переписывания уравнений.

1. Уравнение $f(s'_1, s'_2, \ldots, s'_m) = f(s''_1, s''_2, \ldots, s''_m)$ замещается системой уравнений
$$s'_1 = s''_1$$
$$s'_2 = s''_2$$
$$\vdots$$
$$s'_m = s''_m$$
2. Уравнение $f(s'_1, s'_2, \ldots, s'_m) = g(s''_1, s''_2, \ldots, s''_k)$ в том случае, если $f \neq g$, завершает работу алгоритма с результатом $\theta' \uparrow \theta'' = \tau$;
3. Уравнение (тождество) $t = t$ исключается из системы;
4. Уравнение $t = y$ в том случае, если y – это переменная, а t – это терм, отличный от переменной, замещается уравнением $y = t$;
5. Уравнение $y = t$ в том случае, если y – это переменная, а t – это терм, содержащий y и отличный от y, завершает работу алгоритма с результатом $\theta' \uparrow \theta'' = \tau$;
6. Уравнение $y = t$ в том случае, если y – это переменная, t – это терм, не содержащий y, и переменная y содержится еще хотя бы в

одном уравнении системы, требует применения подстановки $\{y/t\}$ ко всем остальным уравнениям системы (их левым и правым частям);

В том случае, когда образуется система уравнений, к которой неприменима ни одно из указанных выше правил, эта система будет состоять из уравнений $y_1 = t_1, \ldots, y_k = t_k$, в которых все переменные $y_1, \ldots, y_k$ попарно отличны и не содержатся в термах $t_1, \ldots, t_k$. Тогда $\theta' \uparrow \theta'' = \theta'\{y_1/t_1, \cdots, y_k/t_k\} = \theta''\{y_1/t_1, \cdots, y_k/t_k\}$.

Операция унификации возникает при решении задачи унификации, которая состоит в вычислении наиболее общего примера двух выражений E', E''. Если $E' = p(t_1, \ldots, t_n)$, $E' = p(s_1, \ldots, s_n)$ – это атомарные формулы, и при этом $Var_{E'} \cap Var_{E''} = \varnothing$, то наиболее общим примером этих выражений является атом $E_0 = p(x_1, \ldots, x_n)\theta_0$, где $\theta_0 = \{x_1/t_1, \ldots, x_n/t_n\} \uparrow \{x_1/s_1, \ldots, x_n/s_n\}$.

Приведенный выше алгоритм унификации можно применять и для решения более общей задачи унификации конечных множеств пар выражений $\{(E'_1, E''_1), (E'_2, E''_2), \ldots, (E'_m, E''_m)\}$. Эта задача состоит в вычислении подстановки θ, применение которой приводит каждую пару выражений к общему виду. В этом случае работа алгоритма унификации начинается с системы уравнений $E'_1 = E''_1, E'_2 = E''_2, \ldots, E'_m = E''_m$.

2.2. Алгоритм антиунификации подстановок

Операция антиунификации подстановок имеет гораздо меньшую область применения, чем операция унификации. Антиунификация была введена Г. Плоткиным в статье [19] и активно использовалась для решения задач суперкомпиляции в статье [20]. В работе [21] было показано применение операции антиунификации подстановок для вычисления инвариантов программ и выделения синтаксических клонов. Оптимальный по времени алгоритм вычисления антиунификации подстановок, заданных ориентированными ациклическими графами, описан в статье [22]. Ниже приводится более простой алгоритм антиунификации подстановок, заимствованный из статьи [23].

Пусть заданы две подстановки $\theta' = \{x_1/t'_1, x_2/t'_2, \ldots, x_n/t'_n\}$ и $\theta'' = \{x_1/t''_1, x_2/t''_2, \ldots, x_n/t''_n\}$ из множества $Subst(X, F, Y)$, и пусть $Z = Y \setminus (\bigcup_{i=1}^{n} Var_{t'_i} \cup \bigcup_{i=1}^{n} Var_{t''_i})$ – множество вспомогательных переменных, не

входящих в состав термов из области значений подстановок θ' и θ''. Антиунификация $\theta' \downarrow \theta''$ вычисляется итеративным алгоритмом, который строит неубывающую последовательность подстановок $\eta_0, \eta_{1,\ldots}$, преобразуя систему аннотированных уравнений, порожденных подстановками θ' и θ''. Каждое уравнение системы помечается некоторой вспомогательной переменной из множества Z. В начале работы алгоритма система аннотированных уравнений имеет вид

$$\begin{aligned} z_1 &: \quad t_1' = t_1'' \\ z_2 &: \quad t_2' = t_2'' \\ &\vdots \\ z_n &: \quad t_n' = t_n'' \end{aligned}$$

и $\eta_0 = \{x_1 / z_1, x_2 / z_2, \ldots, x_n / z_n\}$. Далее на каждом i-ом шаге работы алгоритма к этой систем уравнений, до тех пор пока это возможно, в произвольном порядке применяются следующие 2 правила переписывания уравнений и вычисления последовательности подстановок $\eta_0, \eta_1, \ldots$.

1. Аннотированное уравнение $z : f(s_1', s_2', \ldots, s_m') = f(s_1'', s_2'', \ldots, s_m'')$ замещается системой уравнений

$$\begin{aligned} z_{N+1} &: \quad s_1' = s_1'' \\ z_{N+2} &: \quad s_2' = s_2'' \\ &\vdots \\ z_{N+m} &: \quad s_m' = s_m'' \end{aligned}$$

   где $z_{N+1}, z_{N+2}, \ldots z_{N+m}$ – переменные из множества Z, ранее не использованные для разметки других уравнений; при этом $\eta_i = \eta_{i-1}\{z / f(z_{N+1}, z_{N+2}, \ldots z_{N+m})\}$.
2. Если в системе есть пара одинаковых аннотированных уравнений $z' : t' = s'$ и $z'' : t'' = s''$, то одно из уравнений (например, $z' : t' = s'$) удаляется из системы; при этом $\eta_i = \eta_{i-1}\{z' / z''\}$.

Как только будет построена такая система аннотированных уравнений $z_{M+1} : s_1' = s_1'', \ldots, z_{M+k} : s_k' = s_k''$, к которой не применимо ни одно из указанных двух правил, алгоритм прекращает работу. Результатом его работы является последняя из построенных подстановок $\eta_j = \theta' \downarrow \theta''$. Пары подстановок $\{z_{M+j} / s_j'\}$ и $\{z_{M+j} / s_j''\}$, $j = 1, \ldots, k$, соответствующие

уравнениям сформировавшейся в конце работы алгоритма системы аннотированных уравнений, мы будем называть аннотирующими подстановками для вспомогательных переменных $z_{M+1},\ldots,z_{M+k}$; для них справедливы равенства $\theta' = (\theta' \downarrow \theta'')\{z_{M+1} / s_1',\ldots,z_{M+k} / s_k'\}$, $\theta'' = (\theta' \downarrow \theta'')\{z_{M+1} / s_1'',\ldots,z_{M+k} / s_k''\}$.

Операция антиунификации возникает при вычислении наиболее специального шаблона двух выражений E_1, E_2, т.е. такого выражения E_0, примером которого является как E_1, так и E_2. Если $E_1 = p(t_1,,\ldots,t_n)$, $E_1 = p(s_1,,\ldots,s_n)$ – это атомарные формулы, то наиболее специальным шаблоном этих выражений является атом $E_0 = p(x_1,,\ldots,x_n)\theta_0$, где $\theta_0 = \{x_1 / t_1,\ldots,x_n / t_n\} \downarrow \{x_1 / s_1,\ldots,x_n / s_n\}$.

3. Стандартные схемы программ

Стандартные схемы программ были введены в статье [12] в качестве математической модели для решения задач верификации и оптимизации последовательных императивных программ. Стандартная схема программ представляет собой конечную систему переходов (ориентированный размеченный граф), вершины которой соответствуют операторам ветвления, проверяющим выполнимость логических условий (предикатов), а переходы между вершинами (дуги графа) соответствуют линейным участкам программы, на которых выполняются последовательности операторов присваивания. Каждое логическое условие описывается атомарной формулой. Каждый оператор присваивания вида $y := t$ может быть ассоциирован с подстановкой $\{y / t\}$; таким образом, линейному участку программы, состоящему из операторов $y_1 := t_1; y_2 := t_2; \ldots, y_n := t_n;$, может быть сопоставлена композиция подстановок $\theta = \{y_1 / t_1\}\{y_2 / t_2\}\cdots\{y_n / t_n\}$, которая адекватно отражает в рамках алгебры подстановок вычислительный эффект выполнения операторов этого линейного участка.

Формальное определение стандартной схемы программ (далее просто программы) таково. Пусть заданы два конечных множества переменных $X = \{x_1,\ldots,x_n\}$ и $Y = \{y_1,\ldots,y_m\}$. Переменные множества X – это входные переменные (параметры) программы, переменные множества Y – это внутренние (локальные) переменные программы. Помимо этих двух множеств переменных при выполнении операции антиунификации подстановок мы будем использовать множество вспомогательных переменных Z.

Программой над множеством переменных X, Y называется размеченная система переходов $\pi = \langle X, Y, V, entry, exit, \to, B, \lambda_0 \rangle$, в которой

V – это конечное множество точек программы, включающее точку входа $entry$ и точку выхода $exit$,

$B : V \to Atom(F, Y)$ – это функция привязки, сопоставляющая каждой точке программы атомарную формулу (логическое условие, проверяемое в этой точке), $\to : (V \setminus \{exit\}) \times \{0,1\} \to Subst(Y, F, Y) \times V$ – это функция переходов, которая для каждой точки программы, отличной от точки выхода, и для каждого истинностного значения логического условия в этой точке указывает подстановку, описывающую результат выполнения линейного участка программы, и точку программы, в которую передается управление после выполнения этого линейного участка,

$\lambda_0 \in Subst(Y, F, X)$ –подстановка, инициализирующая локальные переменные программы.

В дальнейшем при обращении к функции переходов вместо записи $\to(v, \Delta) = (\theta, u)$ мы будем использовать более привычную запись $v \xrightarrow{\Delta, \theta} u$.

Чтобы подчеркнуть особую роль некоторых вершин программы, условимся считать, что точке выхода программы $exit$ приписана атомарная формула вида $OUT(y_{i_1}, y_{i_2}, \ldots, y_{i_k})$, которая выделяет выходные переменные среди локальных переменных программы. В этом случае можно ввести определение вычисления программы в заданной интерпретации на заданной оценке входных переменных (см. в [24]).

Пусть задана произвольная интерпретация $I = \langle D, \overline{F}, \overline{P} \rangle$ сигнатуры (F, P). Оценкой множества переменных Var в интерпретации I называется всякое отображение $\overline{d} : Var \to D$, сопоставляющее каждой переменной элемент из области интерпретации. Если $Var = \{z_1, \ldots, z_k\}$, то для обозначения оценки δ будем использовать запись $\{z_1 \leftarrow \overline{d}(z_1), \ldots, z_k \leftarrow \overline{d}(z_k)\}$. Для обозначения множества всех оценок переменных Var в интерпретации I будем использовать запись $Val(Var, I)$.

Для каждой интерпретации $I = \langle D, \overline{F}, \overline{P} \rangle$ и оценки $\overline{d}, \overline{d} \in Val(X, I)$, обычным образом определяются элемент области интерпретации $t[\overline{d}]$, являющийся значением терма $t, t \in Term(X, F)$, и истинностное значение

$A[\bar{d}]$ атомарной формулы $A, A \in Atom(X, F)$. Всякая подстановка $\theta, \theta \in Subst(X, F, Y)$, в интерпретации I преобразует множество оценок переменных $Val(X, I)$ в множество оценок переменных $Val(Y, I)$ следующим образом: если $\theta = \{y_1 / t_1, y_2 / t_2, \ldots, y_n / t_n\}$, то $\theta(\bar{d}) = \{y_1 \leftarrow t_1[\bar{d}], y_2 \leftarrow t_2[\bar{d}] \ldots, y_n \leftarrow t_n[\bar{d}]\}$.

Для каждой программы $\pi = \langle X, Y, V, entry, exit, \rightarrow, B, \lambda_0 \rangle$ и интерпретации I оценки переменных $Val(X, I)$ играют роль входных данных, а оценки данных $Val(Y, I)$ выступают роли состояний данных вычислений программы. Вычисление программы π в интерпретации I для оценки входных переменных $\bar{d}, \bar{d} \in Val(X, I)$ – это максимальная последовательность

$$comp(\pi, I, \vec{d}) = (v_0, \bar{d}_0), (v_1, \bar{d}_1), (v_2, \bar{d}_2), \ldots, (v_i, \bar{d}_i), (v_{i+1}, \bar{d}_{i+1}), \ldots,$$

удовлетворяющая следующим условиям:

1) $v_0 = entry, \quad \bar{d}_0 = \lambda_0(\bar{d})$;

2) для любого $i, i \geq 1$, в программе π существует переход $v_i \xrightarrow{\Delta, \theta} v_{i+1}$, где $\Delta = B(v_i)(\bar{d}_i)$, и при этом $\bar{d}_{i+1} = \theta(\bar{d}_i)$.

Если вычисление $comp(\pi, I, \vec{d})$ бесконечно, то результат его не определен. Если вычисление $comp(\pi, I, \vec{d})$ завершается парой $(exit, \bar{d}_N)$, и точке выхода приписана атомарная формула $OUT(y_{i_1}, y_{i_2}, \ldots, y_{i_k})$, то результатом вычисления является набор значений переменных $(\bar{d}_N(y_{i_1}), \bar{d}_N(y_{i_2}), \ldots, \bar{d}_N(y_{i_k}))$. Таким образом, для каждой интерпретации I значением программы π в этой интерпретации является частичная функция $\Phi_{\pi, I} : Val(X, I) \rightarrow D^k$, осуществляющая отображение оценок входных переменных в наборы значений переменных на выходе программы. Программы π' и π'' считаются функционально эквивалентными, если для любой интерпретации I эти программы вычисляют одинаковые функции, т.е. $\Phi_{\pi', I} =: \Phi_{\pi'', I}$.

В работе [12] было доказано, что определенная таким образом функциональная эквивалентность программ алгоритмически неразрешима. Последующие исследования показали, что любое невырожденное отношение

эквивалентности программ, определяемое на основе вычислений в интерпретации, неразрешимо даже для очень простых сигнатур (F,P). В связи с этим в статье [14] была предложена иная разновидность программной эквивалентности, – логико-термальная эквивалентность, – которая занимает промежуточное положение между семейством чисто семантических отношений эквивалентности программ, наподобие функциональной эквивалентности, и семейством чисто синтаксических отношений эквивалентности, наподобие изоморфизма программ.

4. Логико-термальная эквивалентность программ

В отличие от функциональной эквивалентности, опирающейся на семантическое понятие вычисления программы в интерпретации, определение логико-термальная эквивалентности программ основывается лишь на сопоставлении синтаксических характеристик анализируемых программ – множеств трасс, ведущих из входов программ в их выходы.

Последовательность переходов в программе π

$$tr = v_0 \xrightarrow{\Delta_0,\theta_0} v_1 \xrightarrow{\Delta_1,\theta_1} v_2 \xrightarrow{\Delta_2,\theta_2} \cdots v_N \xrightarrow{\Delta_N,\theta_N} v_{N+1},$$

в которой $v_0 = entry$, мы будем называть трассой в программе π, ведущей в точку v_{N+1}. Если $v_{N+1} = exit$, то трасса tr будет называться полной трассой. Множество всех полных трасс программы π обозначим записью $CompTr(\pi)$. Префикс длины $i, 0 \le i \le N+1,$ трассы tr будет обозначаться записью $tr|_{\le i}$; при этом мы полагаем $tr|_{\le 0} = v_0$ и $tr|_{\le N+1} = tr$. Каждая трасса tr в программе π вычисляет подстановку $\theta[tr] = \theta_N \cdots \theta_2\theta_1\theta_0\lambda_0$, которая представляет собой композицию инициализирующей подстановки λ_0 и всех подстановок $\theta_0, \theta_1, \theta_2, \ldots \theta_N$, приписанных переходам этой трассы. Логико-термальная история (л.-т. история) полной трассы tr – это последовательность пар

$$lth(tr) = (B(v_0)\theta(tr|_{\le 0}), \Delta_0), \ldots, (B(v_i)\theta(tr|_{\le i}), \Delta_i), \ldots (B(v_N)\theta(tr|_{\le N}), \Delta_N), (B(v_{N+1})\theta(tr), 0),$$

в которой первые компоненты пар – это примеры атомов $B(v_i)$, приписанных точкам трассы tr и специализированные подстановками $\theta(tr|_{\le i})$, вычисленными в этих точках трассы, а вторые компоненты пар – это истинностные значения этих атомов, обеспечивающие прохождение трассы. Множество $Det(\pi) = \{lth(tr) : tr \in CompTr(\pi)\}$ л.-т. историй всех полных трасс программы π называется детерминантом программы. Программы π' и π'' считаются л.-т. эквивалентными, если $Det(\pi') = Det(\pi'')$.

Далее, не ограничивая общности, мы будем полагать, что в рассматриваемых программах через каждую точку проходит хотя бы одна полная трасса, т.е. в программах отсутствуют недостижимые и тупиковые точки. Программы такого вида будем называть редуцированными.

Логико-термальная эквивалентность программ была предложена В.Э. Иткиным в статье [14]. В этой же статье была доказана теорема, устанавливающая взаимосвязь отношений логико-термальной и функциональной эквивалентности программ.

Теорема [14]. Если программы π' и π'' л.-т. эквивалентны, то в любой интерпретации эти программы вычисляют одинаковые функции.

Эта теорема показывает, что для проверки функциональной эквивалентности двух программ достаточно проверить их л.-т. эквивалентность. На основании этой теоремы л.-т. эквивалентность программ можно использовать при решении задач верификации и оптимизации программ.

5. Алгоритм проверки логико-термальной эквивалентности программ

Первый алгоритм, разрешающий л.-т. эквивалентность стандартных схем программ был предложен в статье [14]. Сложность его оценивается двойной экспонентой, зависящей от размера проверяемых программ. Впоследствии были созданы более эффективные алгоритмы; в частности, в работе [15] был разработан алгоритм проверки л.-т. эквивалентности программ за время, полиномиальное относительно размеров программ. В этом разделе статьи описан еще более простой алгоритм проверки л.-т. эквивалентности программ. Мы сводим проверку л.-т. эквивалентности программ к вычислению точной нижней грани в регулярном множестве подстановок; это вычисление осуществляется итеративной процедурой, в которой применяются только операции композиции и антиунификации. Для завершения работы процедуре требуется совершить полиномиальное число шагов. Кроме того, после небольшой модификации эту процедуру можно использовать для решения задачи логико-термальной унификации программ.

Следующее необходимое и достаточное условие л.-т. эквивалентности программ вытекает непосредственно из определения этого отношения эквивалентности программ.

Теорема 1. Для того чтобы редуцированные программы $\pi' = \langle X, Y', V', entry', exit', \to', B', \lambda_0' \rangle$ и $\pi'' = \langle X, Y'', V'', entry'', exit'', \to'', B'', \lambda_0'' \rangle$ были л.-т. эквивалентными необходимо и достаточно, чтобы для любой конечной двоичной

последовательности $\omega = \Delta_0, \Delta_1, \ldots, \Delta_N$, всякий раз, когда в одной из программ (например, в программе π') существует трасса

$$tr' = v'_0 \xrightarrow{\Delta_0, \theta'_0} v'_1 \xrightarrow{\Delta_1, \theta'_1} v'_2 \xrightarrow{\Delta_2, \theta'_2} \cdots v'_N \xrightarrow{\Delta_N, \theta'_N} v'_{N+1},$$

переходы которой размечены символами последовательности ω в другой программе (в данном случае, в программе π'') существует трасса

$$tr'' = v''_0 \xrightarrow{\Delta_0, \theta''_0} v''_1 \xrightarrow{\Delta_1, \theta''_1} v''_2 \xrightarrow{\Delta_2, \theta''_2} \cdots v''_N \xrightarrow{\Delta_N, \theta''_N} v''_{N+1},$$

переходы которой размечены символами той же последовательности ω, и при этом выполняется равенство $B'(v'_{N+1})\theta[tr'] = B''(v''_{N+1})\theta[tr'']$

Трассы tr' и tr'', о которых говорится в теореме 1, будем называть логически согласованными трассами. Таким образом, теорема 1 сводит проверку л.т. эквивалентности программ к анализу пар логически согласованных трасс в этих программах. Этот анализ осуществляется в графе согласованных трасс программ π' и π'', представляющем все пары логически согласованных трасс в этих программах.

Предположим, что требуется проверить логико-термальную эквивалентность двух программ

$\pi' = \langle X, Y', V', entry', exit', \rightarrow', B', \lambda'_0 \rangle$ и $\pi'' = \langle X, Y'', V'', entry'', exit'', \rightarrow'', B'', \lambda''_0 \rangle$,

имеющих одно и то же множество входных переменных X и непересекающиеся множества локальных переменных $Y', Y'', \quad Y' \cap Y'' = \varnothing$. Граф логически согласованных трасс этих программ $\Gamma[\pi', \pi''] = \langle V' \times V'', w_0, \Rightarrow, \lambda_0 \rangle$ устроен следующим образом Вершинами графа являются всевозможные пары $w = (v', v'')$ точек программ π' и π'', и каждой такой вершине приписана пара атомарных формул $(B'(v'), B''(v''))$. В графе $\Gamma[\pi', \pi'']$ особо выделена корневая вершина $w_0 = (entry', entry'')$. Дуги (переходы) графа $\Gamma[\pi', \pi'']$ определяются отношением переходов

$$\Rightarrow \quad \subseteq \quad V' \times V'' \times Subst(Y' \cup Y'', F, Y' \cup Y'') \times V' \times V''$$

Как и для программ, вместо записи $(v', v'', \theta, u', u'') \in \Rightarrow$ мы будем использовать более естественную запись отношения переходов $(v', v'') \overset{\theta}{\Rightarrow} (u', u'')$. Это отношение подчиняется следующему требованию:

$$(v', v'') \overset{\theta}{\Rightarrow} (u', u'') \quad \Leftrightarrow \quad \exists \Delta \in \{0,1\} : v' \xrightarrow{\Delta, \theta'} v'' \wedge v'' \xrightarrow{\Delta, \theta''} u'' \wedge \theta = \theta' \cup \theta''$$

для каждой пары точек (v', v'') и (u', u'') графа $\Gamma[\pi', \pi'']$ и подстановки $\theta, \theta \in Subst(Y' \cup Y'', F, Y' \cup Y'')$. И, наконец, $\lambda_0 = \lambda'_0 \cup \lambda''_0$ – это инициализирующая подстановка графа логически согласованных трасс.

Маршрутом в графе логически согласованных трасс $\Gamma[\pi', \pi'']$ будем называть всякую последовательность дуг $path = (v'_0, v''_0) \overset{\theta_0}{\Rightarrow} (v'_1, v''_1) \overset{\theta_1}{\Rightarrow} (v'_2, v''_2) \overset{\theta_{21}}{\Rightarrow} \cdots \overset{\theta_N}{\Rightarrow} (v'_{N+1}, v''_{N+1})$, начинающуюся в корневой вершине $w_0 = (entry', entry'')$. Как и в случае программ, выражение $\theta[path]$ будет обозначать композицию $\theta_N \cdots \theta_2 \theta_1 \theta_0 \lambda_0$, состоящую из инициализирующей подстановки λ_0 графа логически согласованных трасс и всех подстановок, приписанных дугам этого пути.

Из теоремы 1 и описания устройства графа логически согласованных трасс $\Gamma[\pi', \pi'']$ вытекает следующий критерий л.-т. эквивалентности программ π' и π''.

Теорема 2. Редуцированные программы π' и π'' л.-т. эквивалентны тогда и только тогда, когда для любого конечного маршрута $path$, ведущем в графе $\Gamma[\pi', \pi'']$ из корневой вершины $w_0 = (entry', entry'')$ в вершину $w = (v', v'')$, выполняется равенство $B'(v'_{N+1})\theta[path] = B''(v''_{N+1})\theta[path]$.

Поскольку в графе $\Gamma[\pi', \pi'']$ могут существовать вершины, в которые ведет бесконечно много маршрутов, условия проверки л.-т. эквивалентности программ, представленные в теореме 2, нуждаются в дальнейшем упрощении. Для этого воспользуемся следующей леммой.

Лемма 1. Для любой пары атомарных формул A_1, A_2 и для любой пары подстановок θ', θ'' справедливо соотношение

$$A_1\theta' = A_2\theta' \wedge A_1\theta'' = A_2\theta'' \quad \Leftrightarrow \quad A_1(\theta' \downarrow \theta'') = A_2(\theta' \downarrow \theta'')$$

Доказательство. ($\Rightarrow$) Если $A_1\theta' = A_2\theta'$ и $A_1\theta'' = A_2\theta''$, то это означает, что атомы A_1, A_2 унифицируемы и имеют наиболее общий унификатор μ, для которого верны неравенства $\mu \prec \theta'$ и $\mu \prec \theta''$. Операция антиунификации вычисляет точную нижнюю грань $\theta' \downarrow \theta''$ подстановок θ', θ'', и поэтому $\mu \prec \theta' \downarrow \theta''$. Следовательно, подстановка $\theta' \downarrow \theta''$ унифицирует атомы A_1, A_2, т.е. верно равенство $A_1(\theta' \downarrow \theta'') = A_2(\theta' \downarrow \theta'')$.

(⇐) Очевидно в силу определения точной нижней грани подстановок. □

Для заданного графа $\Gamma[\pi', \pi'']$ логически согласованных трасс в программах π' и π'' мы будем использовать запись $PathSet(w)$ для обозначения множества всех маршрутов, ведущих из корня графа в вершину w. Тогда из теоремы 2 и леммы 1 следует

Теорема 3. Редуцированные программы π' и π'' л.-т. эквивалентны тогда и только тогда, когда для любой вершины $w = (v', v'')$ в графе $\Gamma[\pi', \pi'']$, выполняется равенство $B'(v')\theta[w] = B''(v'')\theta[w]$, где $\theta[w] = \downarrow_{path \in PathSet(w)} \theta[path]$.

Таким образом, задача проверки л.-т. эквивалентности программ сводится к задаче вычисления для каждой вершины w графа $\Gamma[\pi', \pi'']$ точной нижней грани множества всех подстановок, вычислимых на маршрутах, ведущих в вершину w. Для решения этой задачи предлагается следующая процедура глобальной разметки графа логически согласованных трасс.

5.1. Процедура глобальной разметки графа $\Gamma[\pi', \pi'']$

В этой процедуре глобальной разметки графа $\Gamma[\pi', \pi'']$ каждой вершине w этого графа приписывается подстановка η_w, которая приближает сверху искомую точную нижнюю грань $\theta[w]$; это приближение уточняется по ходу работы алгоритма. В начале работы процедуры корневой вершине $w_0 = (entry', entry'')$ приписывается подстановка $\eta_{w_0} = \lambda_0$, а всем остальным вершинам $w, w \neq w_0$, приписывается наибольшая в решетке подстановок мнимая подстановка τ. Далее, до тех пор пока это возможно, применяется следующее правило переписывания подстановок η_w, которыми помечены вершины графа:

> если в графе $\Gamma[\pi', \pi'']$ существует дуга $u \overset{\theta}{\Rightarrow} w$, для которой не выполняется неравенство $\eta_w \prec \theta\eta_u$, то вершине w вместо подстановки η_w приписывается подстановка $\eta'_w = \eta_w \downarrow \theta\eta_u$.

Процедура переписывания завершает работу в том случае, если для всех дуг $u \overset{\theta}{\Rightarrow} w$ графа $\Gamma[\pi', \pi'']$ выполняется неравенство $\eta_w \prec \theta\eta_u$.

Теорема 4. Каковы бы ни были программы π' и π'' процедура глобальной разметки графа $\Gamma[\pi', \pi'']$ логически согласованных трасс в программах π' и

π'' завершает свою работу и вычисляет в каждой вершине w подстановку $\eta_w = \theta[w] = \downarrow_{path \in PathSet(w)} \theta[path]$.

Доказательство. Свойство обрыва убывающих цепей в решетке подстановок гарантирует завершаемость работы процедуры переписывания. Воспользовавшись индукцией по числу шагов процедуры и применяя свойство левой дистрибутивности композиции подстановок относительно операции антиунификации $\eta(\theta_1 \downarrow \theta_2) = (\eta\theta_1) \downarrow (\eta\theta_2)$, можно показать, что на каждом шаге работы процедуры неравенство $\downarrow_{path \in PathSet(w)} \theta[path] \prec \eta_w$ выполняется для каждой вершины w графа $\Gamma[\pi', \pi'']$. Воспользовавшись индукцией по длине маршрута и применяя свойство монотонности композиции подстановок $\theta_1 \prec \theta_2 \quad \Rightarrow \quad \eta\theta_1 \prec \eta\theta_2$, можно показать, что для каждой вершины w графа $\Gamma[\pi', \pi'']$ и для каждого маршрута $path$, ведущего в вершину w, по окончании работы процедуры переписывания выполняется неравенство $\eta_w \prec \theta[path]$. Это означает, что вычисленная в конце работы процедуры глобальной разметки подстановка η_w удовлетворяет равенству $\eta_w = \downarrow_{path \in PathSet(w)} \theta[path]$. □

Таким образом, для проверки л.-т. эквивалентности программ π' и π'' достаточно построить граф $\Gamma[\pi', \pi'']$ логически согласованных трасс в программах π' и π'', применить описанную выше процедуру глобальной разметки и затем проверить для каждой вершины $w = (v', v'')$ в графе $\Gamma[\pi', \pi'']$ выполнимость равенства $B'(v')\eta_w = B''(v'')\eta_w$, где η_w – это подстановка, вычисленная процедурой глобальной разметки для вершины w. Теоремы 2-4 гарантируют завершаемость и корректность предложенного алгоритма проверки л.-т.

Заметим также, что процедура глобальной разметки графа $\Gamma[\pi', \pi'']$, применяющая алгоритм антиунификации, вычисляет не только точные нижние грани подстановок $\theta[w] = \downarrow_{path \in PathSet(w)} \theta[path]$ для каждой вершины w, но также и конечные множества пар аннотирующих подстановок для вспомогательных переменных. Эти аннотирующие подстановки будут играть ключевую роль в решении задачи унификации программ.

6. Логико-термальная унификация программ

Условимся, что для каждой программы $\pi = \langle X, Y, V, entry, exit, \to, B, \lambda_0 \rangle$ и подстановки η запись $\pi\eta$ будет обозначать программу $\pi = \langle X, Y, V, entry, exit, \to, B, \theta_0\eta \rangle$, полученную из исходной программы заменой инициализирующей подстановки θ_0 композицией подстановок $\theta_0\eta$. Программа $\pi\eta$ называется примером программы π. Подстановка η называется логико-термальным унификатором пары программ π' и π'', если примеры $\pi'\eta$ и $\pi''\eta$ этих программ л.-т. эквивалентны. Унификатор η программ π' и π'' считается наиболее общим л.-т. унификатором, если для любого л.-т. унификатора μ этих программ выполняется неравенство $\eta \prec \mu$. Задача л.-т. унификации программ состоит в том, чтобы для любой заданной пары программ вычислить их наиболее общий л.-т. унификатор.

Задачу л.-т. унификации программ можно свести к задаче унификации множества пар атомарных формул, но это множество в общем случае может содержать бесконечно много пар атомов. Предположим, что имеются две программы $\pi' = \langle X', Y', V', entry', exit', \to', B', \lambda_0' \rangle$ и $\pi'' = \langle X'', Y'', V'', entry'', exit'', \to'', B'', \lambda_0'' \rangle$, у которых множества входных переменных X', X'' и локальных переменных Y', Y'' попарно не пересекаются. Рассмотрим граф $\Gamma[\pi', \pi'']$ логически согласованных трасс в программах π' и π''. Как и прежде, запись $PathSet(w)$ обозначает множество всех маршрутов, ведущих из корня графа в вершину w.

Теорема 5. Подстановка η, $\eta \in Subst(X' \cup X'', F, X' \cup X'')$, является л.-т. унификатором программ π' и π'' тогда и только тогда, когда η является унификатором множества пар атомарных формул

$$\{(B'(v')\theta[path],\ B''(v'')\theta[path]) \ : \ (v', v'') \in V' \times V'',\ path \in PathSet((v', v''))\}..$$

Справедливость теоремы 5 следует из теоремы 3 и определения л.-т. унификатора программ. Таким образом, для унификации программ π' и π'' достаточно разработать способ унификации бесконечного множества пар атомарных формул. Решить эту задачу при помощи одного лишь алгоритма унификации, описанного в разделе 2.1, невозможно. Для ее решения мы воспользуемся комбинацией двух алгоритмов – процедурой глобальной разметки графа логически согласованных трасс и алгоритмом унификации конечного множества пар атомарных формул. Поочередно проводя глобальную разметку графа логически согласованных трасс и унификацию конечного числа пар атомарных формул, описанная ниже процедура л.-т.

унификации программ формирует конечную последовательность подстановок $\eta_1, \eta_2, \ldots, \eta_N$, приближающих снизу л.-т. унификатор заданной пары программ η_N, который вычисляется на последнем этапе работы процедуры.

6.1. Процедура л.-т. унификации программ.

В начале первого этапа работы процедуры полагаем $\eta_1 = \{x_1'/x_1', \ldots, x_n'/x_n', x_1''/x_1'', \ldots, x_m''/x_m''\}$ (тождественная подстановка)

На каждом i-ом этапе работы процедуры строится граф $\Gamma[\pi'\eta_i, \pi''\eta_i]$ логически согласованных трасс в примерах $\pi'\eta_i, \pi''\eta_i$ программ π' и π''.

Затем к графу $\Gamma[\pi'\eta_i, \pi''\eta_i]$ применяется процедура глобальной разметки, которая вычисляет для каждой вершины w подстановку $\theta[w], \theta[w] \in Subst(Y' \cup Y'', F, X' \cup X'' \cup Z)$, и конечное множество пар аннотирующих подстановок $\{(\{z/s'\}, \{z/s''\}) : z \in Z\}$ для вспомогательных переменных, возникающих по ходу применения операции антиунификации. В графе $\Gamma[\pi'\eta_i, \pi''\eta_i]$ выделяем множество вершин $W_i = \{w : w = (v', v'') \wedge B'(v')\theta[w] \neq B''(v'')\theta[w]\}$, в которых нарушается условие л.-т. эквивалентности примеров программ $\pi'\eta_i, \pi''\eta_i$, описанное в теореме 3.

Если $W_i = \varnothing$, то согласно теореме 3 примеры программ $\pi'\eta_i, \pi''\eta_i$ являются л.-т. эквивалентными, и подстановка η_i объявляется результатом работы процедуры л.-т. унификации программ π' и π''.

Если $W_i \neq \varnothing$, то к конечному множеству пар атомарных формул применяется модифицированный алгоритм унификации. На вход этого алгоритма поступает система уравнений

$$\{ B'(v')\theta[w] = B''(v'')\theta[w] \quad : \quad w = (v', v'') \in W_i \}.$$

К этой системе, до тех пор пока это возможно, применяются 7 правил переписывания уравнений, заимствованных из алгоритма унификации. Первые 5 правил в точности совпадают с правилами 1-5 из раздела 2.1., в котором описан алгоритм выполнения операции унификации подстановок. Последние два правила представляют собой следующие модификации правила 6 из раздела 2.1:

6.1. Уравнение $x = t$ в том случае, если x – это входная переменная из множества $X' \cup X''$, t – это терм, не содержащий x, и переменная x содержится еще хотя бы в одном уравнении

системы, требует применения подстановки $\{x/t\}$ ко всем остальным уравнениям системы (их левым и правым частям);

6.2. Уравнение $z = t$ в том случае, если z – это вспомогательная переменная из множества Z, снабженная парой аннотирующих подстановок $\{z/s'\}$ и $\{z/s''\}$, а t – это терм, не содержащий z, требует замены каждого уравнения системы $t' = t''$ (включая само уравнение $z = t$) парой уравнений $t'\{z/s'\} = t''\{z/s'\}$ и $t'\{z/s''\} = t''\{z/s''\}$.

Если ни одно из правил 2 или 5 «аварийного» завершения работы алгоритма унификации не применяется, то модифицированный алгоритм унификации прекращает свое выполнение, как только будет построена система уравнений, к которой неприменимо ни одно из перечисленных правил 1-5, 6.1, 6.2. Образованная в этом случае система уравнений состоит только из уравнений $x_1 = t_1, \ldots, x_k = t_k$, в которых все переменные $x_1, \ldots, x_k$ – это попарно различные входные переменные из множества $X' \cup X''$, не содержащиеся в термах $t_1, \ldots, t_k$. Тогда результатом работы i-ого этапа процедуры л.-т. унификации объявляется подстановка $\eta_{i+1} = \eta_i\{x_1/t_1, \ldots, x_k/t_k\}$.

Теорема 6. Каковы бы ни были программы π' и π'', описанная выше процедура унификации программ завершает свое выполнение. Подстановка η_N, вычисленная этой процедурой, является наиболее общим унификатором программ π' и π''.

Доказательство. Чтобы убедиться в завершаемости описанной процедуры, достаточно заметить, что для последовательности подстановок $\eta_1, \eta_2, \ldots, \eta_N$, вычисленных на каждом этапе ее работы, имеет место строгое включение $Var_{\eta_{i+1}} \subset Var_{\eta_i}$. Следовательно, процедура не может иметь более $|X'| + |X''|$ этапов выполнения.

Чтобы обосновать корректность процедуры, достаточно доказать, используя индукцию по числу этапов, справедливость следующих двух утверждений: 1) последовательность $\eta_1, \eta_2, \ldots, \eta_N$ является строго возрастающей последовательностью в решетке подстановок, и 2) для любого члена η_i этой последовательности и для любого л.-т. унификатора ρ программ π' и π'' выполняется неравенство $\eta_i \prec \rho$. Справедливость первого утверждения следует из определения последовательности подстановок $\eta_1, \eta_2, \ldots, \eta_N$. Для доказательства последнего из этих двух утверждений следует воспользоваться

теоремой 5 и заметить, что согласно описанию процедуры л.-т. унификации каждая из подстановок η_{i+1} является наиболее общим унификатором непустого множества пар атомарных формул

$$\{(B'(v')\theta[path], B''(v'')\theta[path]) \ : \ (v',v'') \in W_i,\ path \in R\}$$

для некоторого подмножества маршрутов $R \subseteq PathSet((v',v''))$ в графе логически согласованных трасс в программах π' и π''. □

Предложенный нами алгоритм л.-т. унификации программ можно применять для решения задачи процедурной абстракции. Предположим, что унификатором пары программ $\pi' = \langle X', Y', V', entry', exit', \to', B', \lambda_0' \rangle$ и $\pi'' = \langle X'', Y'', V'', entry'', exit'', \to'', B'', \lambda_0'' \rangle$ является подстановка $\eta_1 = \{x_1''/t_1, \ldots, x_m''/t_m\}$, в которой все термы $t_1, \ldots, t_m$ зависят только от переменных из множества X' (от входных параметров программы π'). Тогда программа π'' является примером программы π'. В этом случае программу π' можно объявить отдельной процедурой, а все вхождения программы π'' можно заменить вызовом этой процедуры $call\ \pi'(t_1, \ldots, t_m)$ со списком параметров $t_1, \ldots, t_m$.

7. Заключение

Основной результат этой статьи – это решение двух задач, возникающих при анализе последовательных императивных программ, – задачи проверки эквивалентности программ и задачи унификации программ. Для решения обеих задач был задействован математический аппарат алгебры подстановок с двумя основными операциями – операцией взятия точной верхней грани (унификация) и точной нижней грани (антиунификация) двух подстановок. На основе этих операций были предложены алгоритм проверки логико-термальной эквивалентности и алгоритм логико-термальной унификации программ. Эти алгоритмы можно использовать для решения многих задач семантического анализа и эквивалентных преобразований программ, включая задачи оптимизации, верификации и реорганизации программ,

Вместе с тем, практическое применение предложенных нами алгоритмов требует решения целого ряда вопросов, касающихся сложности рассмотренных задач. Вот лишь некоторые из них.

1. Оба операции над подстановками – унификации и антиунификация, – могут быть выполнены за время, пропорциональное размерам исходных данных. Однако наибольшая эффективность выполнения этих операций достигается на разных формах представления

подстановок: для эффективного выполнения операции унификации желательно, чтобы термы были представлены в виде ориентированных размеченных ациклических графов, в то время как операция антиунифкации наилучшим образом приспособлена для работы с древесным представлением термов. Поскольку в алгоритме л.-т. унификации программ задействованы обе операции, необходимо отыскать такую форму представления термов, которая была бы оптимальной для обеих операций.

2. Для проверки л.-т. эквивалентности программ достаточно уметь вычислять любую (не обязательно точную) нижнюю грань двух подстановок при условии, что операция вычисления этой нижней грани будет обладать свойством левой дистрибутивности относительно операции композиции подстановок, и для нее будет справедлива лемма 1. В связи с этим целесообразно изучить вопрос о выборе оптимальной по сложности вычисления операции взятия какой-либо нижней грани двух подстановок, которую можно использовать в процедуре глобальной разметки графа логически согласованных трасс для проверки л.-т. эквивалентности программ.
3. Предложенный нами алгоритм л.-т. унификации программ – это многопроходная процедура разметки графов логически согласованных трасс программ. Поэтому одно из направлений повышения эффективности алгоритма л.-т. унификации программ состоит в сокращении числа проходов этой процедуры.

Отметим также, что алгоритмическое решение задач проверки логико-термальной эквивалентности программ и выполнения логико-термальной унификации программ создает хорошие предпосылки для решения задачи, имеющей более важное прикладное значение, – вычисление наиболее специального шаблона двух программ (задачи логико-термальной антиунификации программ).

Список литературы

[1] Lague B., Proulx D., Mayrand J. et al. Assessing the benefits of incorporating function clone detection in a development process // Proceedings of the International Conference on Software Maintenance. — Washington, DC, USA: IEEE Computer Society, 1997.— p. 314–321.

[2] Baker B. S. On finding duplication and near-duplication in large software systems // Proceedings of the Second Working Conference on Reverse Engineering. — Washington, DC, USA: IEEE Computer Society, 1995. — p. 86–95.

[3] Kontogiannis K. A., Demori R., Merlo E. et al. Pattern matching for clone and concept detection // Journal of Automated Software Engineering.— 1996.— Vol. 3.— p. 108 - 125.

[4] Li Z., Lu S., Myagmar S., Zhou Y.. Cp-miner: Finding copy-paste and related bugs in large-scale software code // IEEE Transactions on Software Engineering. — 2006. — v. 32, N 3.— p. 176–192.

[5] Johnson J. H. Identifying redundancy in source code using fingerprints // Proceedings of the Conference Centre for Advanced Studies on Collaborative research (CASCON). — IBM Press, 1993.— p. 171–183.
[6] Фаулер М. Рефакторинг. Улучшение существующего кода. — Символ-Плюс, 2008. — 432 с.
[7] 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.
[8] 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.
[9] Robinson J.A. A machine-oriented logic based on the resolution principle // Journal of the ACM. — 1965. — v. 12, N 1 — p. 23_41.
[10] Baader F., Snyder W. Unification theory. — Handbook of Automated Reasoning, v. I, Elsevier Science Publishers, 2001. — p. 447–533.
[11] Yelick K.A. Generalized approach to equational unification // Technical Report, Massachusetts Institute of Technology Cambridge, MA, USA, 1990.
[12] 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.
[13] Itkin V.E., Zwinogrodski Z. On program schemata equivalence // Journal of Computer and System Science. — 1972. — v. 6, N 1. — p. 88-101.
[14] Иткин В.Э. Логико-термальная эквивалентность схем программ // Кибернетика. — 1972. — N 1. — с. 5-27.
[15] Sabelfeld V.K. The logic-termal equivalence is polynomial-time decidable // Information Processing Letters. — 1980 — v. 10, N 2. — p. 57-62.
[16] Eder E. Properties of substitutions and unifications // Journal of Symbolic Computations. — v. 1. — 1985. — p. 31-46.
[17] Palamidessi C. Algebraic properties of idempotent substitutions // Lecture Notes in Computer Science — v. 443 — 1990. — p. 386-399.
[18] Martelli A., Montanari U. An Efficient Unification Algorithm // Transactions on Programming Languages and Systems (TOPLAS). — 1982. — v.4, N 2. — p. 258-282.
[19] Plotkin G.D. A note on inductive generalization // Machine Intelligence. — 1970. — v. 5, N 1. — p. 153-163.
[20] Sorensen M. H., Gluck. R. An algorithm of generalization in positive supercompilation // Proceedings of the 1995 International Symposium on Logic Programming._MIT Press. — 1995. — p. 465-479.
[21] Bulychev P., Kostylev E., Zakharov V. Anti-unification algorithms and their applications in program analysis // Proceedings of the 7th International Conference "Perspectives of System Informatics", June 15-19, 2009, Novosibirsk. — 2009. — p. 82-89.
[22] Захаров В.А., Костылев Е.В. О сложности задачи антиунификации // Дискретная математика. — 2008. — т. 20, N 1. — с. 131-144.
[23] Zakharov V.A. On the refinement of logic programs by means of anti-unification // Proceedings of the 2nd Panhellenic Logic Symposium, Delphi, Greece. — 1999. — p. 219-224.
[24] Котов В.Е., Сабельфельд В.К. Теория схем программ. — М.:Наука, 1991. — 348 с.

On the application of substitution algebra to program unification

V.A.Zakharov (ISP RAS),
T.A. Novikova (Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University) {zakh@cs.msu.su} {taniaelf@mail.ru}

Abstract. Many problems in software engineering such as program refactoring, deobfuscation, vulnerability detection, require an efficient toolset for detecting pieces of code that have similar behavior. Current state of art in software clone detection makes it possible to find out only those pieces of code which have the same syntactic structure since. A more profound analysis of functional properties of programs encounters with undecidability of equivalence checking problem for Turing-complete models of computation. To overcome this principal deficiency of modern clone detection techniques we suggest to use equivalence relations on programs that are more strong than functional equivalence. One of such equivalences is a so called logic&term program equivalence introduced by V.E. Itkin in 1972 г. In this paper we build a new algorithm for checking logic&term equivalence of programs. This algorithm is based on the operations for computing the least upper bound and the greatest upper bound in the lattice of finite substitutions. Using this algorithm we also develop a procedure for solving the unification problem for sequential programs, i.e. the problem of computing the most general common instance (specialization) of a given pair of programs (pieces of code).

Keywords: program, clone detection, equivalence, substitution, unification, antiunification.

Риски проектирования и производства мобильных программных продуктов

В.В. Липаев
lip@ispras.ru

Аннотация. Вводятся основные понятия и свойства рисков комплексов программ. Рассматриваются факторы и виды рисков комплексов программ и систем. Обсуждается подготовка исходных данных для анализа, прогнозирования и сокращения рисков комплексов программ. Описываются выделение, идентификация, анализ угроз и рисков в комплексах программ. Рассматриваются методы сокращения и ликвидации опасных рисков, регистрации и утверждения допустимого интегрального риска программного продукта.

Ключевые слова: программные продукты, качество и виды рисков программ; дефекты, идентификация, угрозы рисков при производстве и испытаниях программ; регистрация, сокращение и ликвидация рисков программных продуктов.

1. Основные понятия и свойства рисков мобильных комплексов программ

Современные программные продукты для сложных систем активно применяются в критических и ответственных системах динамического управления в ***реальном времени*** объектами, в высокоточном технологическом производстве, в авиации, в управлении космическими аппаратами, атомными электростанциями и оборонной техникой. Они являются одними из наиболее сложных ***интеллектуальных систем высокого качества***, создаваемых человеком и их результатов – программных продуктов [1, 4]. Оценки качества программных продуктов могут проводиться с двух позиций: с ***позиции положительной*** эффективности и адекватности их характеристик назначению, целям создания и применения, а также с ***негативной позиции*** возможного при этом ущерба – риска при их создании и использовании. Характеристики качества и риски объектов и процессов обычно тесно связаны, на них влияют подобные факторы, которые с разных сторон отражаются на свойствах систем или комплексов программ. Показатели качества преимущественно отражают положительный эффект от применения системы или программного продукта и основная задача разработчиков проекта состоит в обеспечении требований заказчика и пользователей заданного качества. Повышению качества проекта обычно ***сопутствует***

снижение его рисков и наоборот сокращение рисков способствует улучшению характеристик качества. Поэтому методы и системы управления качеством близки к методам анализа и управления рисками комплексов программ, они должны их дополнять и совместно способствовать совершенствованию программных продуктов и систем на их основе.

Риски – это негативные события и их последствия, отражающие потери, убытки или ущерб от процессов или продуктов, вызванные реализацией угроз при наличии уязвимости и ***снижения безопасности*** применения системы. Они проявляются при недостатках и дефектах обоснования, проектирования, производства и всего жизненного цикла комплексов программ. Это негативные последствия функционирования и/или применения программных продуктов, в результате отклонения характеристик объектов или процессов от заданных требований заказчика, согласованных с разработчиками, которые способны нарушать безопасность и вызвать ущерб системе, внешней среде или пользователю [3, 7, 10].

Причинами возникновения и проявления рисков могут быть: ***злоумышленные, активные воздействия заинтересованных лиц*** или ***случайные негативные проявления дефектов*** внешней среды, системы, ошибки разработчиков или пользователей. В первом случае риски могут быть обусловлены искажениями программ и информационных ресурсов и их уязвимостью от предумышленных, внешних воздействий (атак) с целью незаконного использования или искажения информации и программ, которые по своему содержанию предназначены для применения ограниченным кругом лиц. При этом подразумевается наличие лиц, заинтересованных в доступе к конфиденциальной или полезной информации в системах, с целью ее использования или разрушения**.**

Риски при случайных, дестабилизирующих воздействиях дефектов и отсутствии предумышленного негативного влияния на системы, программный продукт или информацию баз данных зависят от отказовых ситуаций, отрицательно отражающихся на работоспособности и реализации их основных функций. При этом катастрофически, критически или существенно искажается процесс функционирования программного продукта и системы, что может наносить значительный ***ущерб безопасности*** их применения. Одним из косвенных методов определения ***величины риска*** может быть ***оценка совокупных затрат***, необходимых для ликвидации негативных последствий риска в программном продукте, системе или внешней среде, проявляющихся в результате конкретного рискового события.

Анализы рисков – процессы определения источников и количественного оценивания рисков, угроз, уязвимостей, возможного ущерба, а также контрмер для их уменьшения. Для этого предварительно должны быть определены требования к характеристикам комплекса программ и оценки возможного ущерба при их нарушении. Анализ негативного воздействия, должен устанавливать критерии, используемые для идентификации вторичные

последствия, распространяющиеся на компоненты и системы. Модели последствий требуются для прогнозирования размеров возможных аварий, катастроф и других негативных явлений в различных системах. Они включают идентификацию опасностей, угроз и оценки возможных последствий и ущерба от проявления рисков; проверку достоверности результатов анализа рисков; документальное обоснование возможных рисков. В результате анализа следует создавать план ***отслеживания изменения и сокращения рисков в жизненном цикле комплекса программ***, который должен регулярно рассматриваться и корректироваться. [7, 8, 10].

Управление рисками – процесс идентификации, управления, устранения или уменьшения вероятности событий, которые могут негативно воздействовать на комплекс программ, систему и внешнюю среду, действия, осуществляемые для выполнения решений по мониторингу и сокращению рисков. Процесс включает анализ соотношения стоимости/эффективности контрмер, выбор, построение и испытание подсистемы ***обеспечения безопасности***, и исследование всех аспектов проявления рисков системы. Управления рисками предполагает ясное понимание специалистами внутренних и внешних причин и возможных реальных источников угроз, влияющих на качество программного продукта, которые могут привести к большому ущербу.

Анализом и разработкой методов выявления и устранения рисков систем занимается ***Федерация европейской ассоциации менеджеров рисков*** – FERMA. Разработан и опубликован ряд моделей и стандартов, регламентирующих методы анализа и управления рисками проектирования и производства сложных систем. ***Общие методы анализа рисков в сложных системах***, регламентированы стандартами **ISO/IEC 17799, ГОСТ Р 51901,** Семейством международных стандартов Управления информационной безопасностью **ISO 27000,** стандартом **ISO 15408** – Общие критерии оценки безопасности и рисков информационных технологий. Изложенные в стандартах рекомендации могут быть, в частности, применены при анализе и ликвидации рисков сложных комплексов программ и систем [3, 7, 8, 10].

2. Основные факторы и виды рисков комплексов программ и систем

Необходимым требованием к специалистам для выполнения анализа и управления рисками должно быть ***достоверное знание целей и структуры комплекса программ, исследуемой системы и внешней среды***, а также доступных методов анализа и прогнозирования рисков. Область применения методов анализа и сокращения рисков должна быть определена и документально зафиксирована. Для этого следует составить описание основных проблем, определивших целесообразность проведения исследования рисков. Для выработки плана исследований ***область анализа рисков*** должна быть определена и документально установлена. Она должна включать в себя следующее:

- описание оснований и/или проблем, повлекших необходимость анализа рисков, которое включает: формулировку задач анализа рисков, основанных на идентифицированных потенциальных опасностях, угрозах; определении критериев работоспособности и отказов системы;

- описание исследуемой системы – определение границ и областей интерфейса со смежными системами; описание условий окружающей среды;

- установление возможных источников, предоставляющих подробную информацию о всех технических, связанных с окружающей средой, правовых, организационных и человеческих факторах, имеющих отношение к анализируемым действиям и проблеме; обстоятельства, влияющие на безопасность;

- описание используемых предположений и ограничивающих условий при проведении анализа и прогнозировании рисков;

- формулировка возможных решений по сокращению рисков, которые могут быть приняты, описание требуемых выходных данных, полученных по результатам исследований и от лиц, принимающих решения.

Общей задачей анализа риска является обоснование и подготовка решений, касающихся сокращения рисков критических программных продуктов и систем на двух основных стадиях жизненного цикла.

На стадии проектирования:

- предоставление исходных данных для оценки качества системы в целом;

- выявление главных возможных источников угроз рисков и предполагаемых факторов, существенно влияющих на риски;

- определение и оценка эффективности возможных мер обеспечения безопасности, закладываемых в программный продукт и систему;

- предоставление исходных данных для оценки потенциально опасных действий, компонентов оборудования или системы;

- обеспечение специалистов соответствующей информацией при проведении опытно-конструкторских работ, ориентированных на нормальные и чрезвычайные условия функционирования комплекса программ и системы;

- оценка рисков с учетом регламентов и других требований поддержки применения комплексов программ;

- - оценка альтернативных, конструктивных решений для сокращения

рисков при проектировании комплекса программ.

На стадии производства и эксплуатации комплексов программ:

- контроль и оценка данных эксплуатации с целью сопоставления фактических показателей качества с соответствующими требованиями;

- обеспечение исходными данными процессов производства, методик эксплуатации, технического обслуживания, контроля и действий в чрезвычайных ситуациях проявления рисков;

- корректировка информации об основных источниках угроз, рисков и влияющих факторах;

- предоставление информации по значимости риска для принятия оперативных решений его сокращения;

- определение влияния на риски изменений в организационной структуре, производстве, процедурах эксплуатации и компонентах программного продукта и системы;

- подготовка персонала к применению программного продукта и системы при возможности проявления рисков.

Проявления рисков могут включать взаимосвязанные стоимостные и плановые виды рисков. ***Стоимостные риски*** составляют: нарушения ограничения суммарного бюджета проектирования и производства программного продукта; нарушения заданной ограниченной длительности производства компонентов и комплексов программ. ***Плановые риски*** включают: ущерб от дефектов стратегии и планирования проекта производства комплекса программ; достоверности планов, сроков и этапов жизненного цикла комплекса программ; нарушения требований и стандартов предотвращения, управления и сокращения рисков.

В проектах сложных систем, использующих программные продукты, риски могут быть обусловлены дефектами функциональных характеристик самих программ и их жизненного цикла, а также недостатками систем и внешней среды, в которой они используются. Основной ***ущерб от рисков программных продуктов*** проявляется в последствиях их применения – ***в дефектах и недостатках функционирования систем и внешней среды***. Поэтому анализ и оценка рисков комплексов программ должны быть тесно связаны с исследованием их проявления в системах, где они используются, причинами которых могут быть следующие ***виды рисков***: реализации функциональной пригодности программных продуктов; реализации конструктивных характеристик программного комплекса; ограничений ресурсов на проектирование и производство программного продукта. Результирующий ***ущерб*** в совокупности зависит от величины и вероятности проявления ***каждого вида риска***.

Риски реализации функциональной пригодности программных продуктов включают: назначение; функции; масштаб – размер; сложность программного комплекса.

***Риски реализации конструктивных характеристик программного комплекса составляют*:** корректность программ компонентов и комплекса; способность компонентов к взаимодействию; защищенность – безопасность; надежность – готовность; временная эффективность функционирования; сопровождаемость – изменяемость версий программного продукта.

***Риски ресурсов проектирования и производства программного продукта обусловлены ограничениями*:** экономических и трудовых затрат; квалификации коллектива специалистов; технических, вычислительных ресурсов на проектирование, производство и функционирование программного продукта.

При анализе и управлении сокращением рисков программных продуктов целесообразно выделять наиболее характерные этапы их ЖЦ: технико-экономического обоснование проекта; разработку требований спецификаций; проектирование; кодирование; тестирование; и документирование. При обосновании и реализации комплексов программ, анализ и управление их рисками должны являться частью общей ***проблемы обеспечения высокого качества проекта, предотвращения и сокращения рисков в системе и внешней среде*** [1, 2, 6]. Эти процессы состоят в выявлении возможных негативных отклонений характеристик комплекса программ и систем от требований контракта, технического задания и спецификаций, а также в создании базы для принятия мер по минимизации таких отклонений, с учетом ограниченных ресурсов на их реализацию и других факторов. Для этого при оценивании рисков программных продуктов, их необходимо трансформировать в величины возможных ***рисков для систем, среды и пользователей***, которые являются важнейшими и определяющими при применении программных продуктов. С точки зрения специалистов-разработчиков систем, принимающих решения, к основным ***методам анализа*** относятся:

- систематическая идентификация потенциальных опасностей, угроз и видов возможных отказов конкретной системы;

- количественные оценки или ранжирование опасности – ущерба от возможных рисков;

- оценка надежности и эффективности возможных контрмер и модификаций системы для снижения риска и достижения предпочтительных уровней ее качества;

- выявление факторов, обуславливающих возможный риск, и слабых звеньев в системе;

- сопоставление возможных рисков исследуемой системы с рисками

альтернативных систем или технологий.

Ниже внимание сосредоточено ***на практических задачах и методологии*** анализа, оценивания и сокращения рисков программных продуктов, в процессе их проектирования и производства. Основные результаты обобщения моделей и факторов для сокращения рисков отражены ***этапами проектирования и производства***, которые рекомендуется выполнять при реализации базовых работ жизненного цикла сложных программных комплексов, и могут служить основой для разработки соответствующих планов работ при управлении, прогнозировании и сокращении рисков.

3. Подготовка исходных данных для анализа, прогнозирования и сокращения рисков комплексов программ

Обычно отсутствуют отдельные, предсказуемые факторы или методы, способные существенно изменять основные риски процесса разработки программ. Риски в ЖЦ комплекса программ могут быть обусловлены недостатками или непредумышленными, ***негативными действиями различных лиц***, участвующих в создании или применении системы и программного продукта. Основными ***источниками непредумышленных рисков*** программных продуктов, которые могут приводить к ущербу при их разработке и применении, являются:

- ***заказчики,*** определяющие назначение и функций системы и программного продукта, которые могут задавать некорректные или нереализуемые разработчиками требования к ним, а также ограничивают выделенные и доступные для проекта ресурсы: бюджет, время, затраты на технологию и инструментальные средства;

- ***разработчики*** системы и комплекса программ, обеспечивающие реализацию его ЖЦ, могут допускать дефекты и ошибки при обосновании проекта, не выполнять согласованные с заказчиком требования к характеристикам и качеству комплекса программ, а также превышать допустимое использование выделенных ресурсов, что может отражаться на проявлении и последствиях рисков на различных технологических этапах;

- ***менеджеры и эксперты управления рисками*** – координаторы взаимодействия заказчиков и разработчиков, которые уполномочены принимать решения о необходимости их изменения, путем применения необходимых контрмер, а также о допустимости применения системы и/или программного продукта с прогнозируемыми или достигнутыми, конкретными уровнями рисков.

Источниками и причинами рисков функционирования могут быть также ***пользователи***, некомпетентно применяющие систему или программный

продукт с отклонениями от требований документации по функциональной пригодности или с недопустимым использованием ресурсов при эксплуатации.

Подготовка исходных данных для анализа и управления рисками программного продукта должна ***устанавливать источники*** подробной информацию о всех технических, связанных с окружающей средой, правовых, организационных и человеческих факторах, имеющих отношение к анализируемой проблеме, программному комплексу и системе, ***предположения*** о содержании, месте и условиях возможного проявления рисков. Следует сформулировать ожидаемые и возможные альтернативные решения, которые могут быть приняты, структура и описание предполагаемых рисков по результатам исследований и от лиц, принимающих решения. Исходные данные проекта программного комплекса должны содержать:

- требования к функциям и характеристикам качества комплекса программ;

- описание и графическое представление его архитектуры, базы данных и взаимодействия компонентов;

- предполагаемую модель жизненного цикла комплекса программ;

- предварительные планы последующих этапов проектирования и производства комплекса программ;

- проекты технического задания и контракта на детальное проектирование и весь жизненный цикл комплекса программ [2, 6, 8].

Для этого следует подготовить ***требования к документации*** и обеспечить их реализацию, которая должна быть однозначной – написана в стандартизированных терминах, уточняемых, если необходимо, соответствующими комментариями. Должна быть выполнена первичная идентификация возможных опасностей, угроз и предварительная оценка возможных их последствий, являющихся причиной рисков. Известные потенциальные опасности должны быть четко и точно определены и описаны. Предварительную оценку значения идентифицированных опасностей – угроз необходимо выполнять, основываясь на анализе последствий рисков и изучении их основных причин у аналогичных проектов.

Программные продукты для обработки информации и управления обычно входят компонентами в системы более высокого уровня и зависят от внешней среды и функций системы, в которой они используются. Описания назначения, функций и требований к характеристикам системы, внешней среды и комплекса программ являются исходными данными трех взаимосвязанных технологических процессов:

- базовых, регламентированных ***технологических процессов и инструментария*** для их автоматизации, обеспечивающих

проектирование и производство сложных программных продуктов;

- методов и средств обеспечения, требуемых характеристик комплекса программ на базе ***системы автоматизации контроля и обеспечения качества*** процессов и продуктов в их жизненном цикле;

- процессов и системы автоматизированного ***анализа, управления и сокращения рисков*** при создании и применении комплексов программ и систем в заданной внешней среде.

Для обеспечения высокого качества программного продукта целесообразно формировать группы экспертов для анализа угроз и управления рисками проектирования и производства программного продукта. Их следует ***выделять из основного жизненного цикла комплекса программ*** и поручать экспертам, разработку требований к функциональной пригодности и конструктивным характеристикам программного продукта с учетом возможных рисков[1, 2, 6].

4. Выделение, идентификация, анализ угроз и рисков в комплексах программ

В процессе управления проектом значительное внимание должно уделяться прогнозированию угроз и рисков, имеющих как внешние, так и внутренние причины. Этот ***этап анализа*** включает:

- выделение возможных источников и угроз нарушения требований и ограничений ресурсов в жизненном цикле комплекса программ, определение критериев функциональной работоспособности и/или отказа системы и программного продукта вследствие проявления рисков;

- идентификацию и анализ причин, выделение категорий и возможных последствий проявления рисков функциональной пригодности и конструктивных характеристик программного продукта;

- идентификацию и анализ причин, выделение категорий и возможных последствий рисков нарушения ограничений доступных ресурсов для проекта комплекса программ.

Накопленный опыт и обобщение проведенных исследований позволили выделить основные ***группы факторов*** [1, 5, 6], влияющих на риски при производстве комплексов программ:

- факторы, отражающие особенности создаваемого комплекса программ, как ***объекта*** разработки, требования к функциональным характеристикам и к качеству;

- факторы, определяющие ***организацию процесса*** разработки комплекса программ и его обеспечение квалифицированными

специалистами;

- факторы, характеризующие ***технологическую среду*** и оснащенность инструментальными средствами автоматизации разработки комплекса программ;

- факторы, отражающие оснащенность производственного процесса ***аппаратурными вычислительными средствами***, на которых реализуются комплексы программ и базируются инструментальные системы автоматизации разработки.

Риски функциональной пригодности имеют доминирующее значение и изменения других видов рисков обычно должны быть, в первую очередь, подчинены сокращению этих рисков системы и комплекса программ. Цель и назначение программного продукта детализируются и формализуются в ***требованиях к функциям*** компонентов и всего комплекса программ, способного реализовать декларированные цели системы при отсутствии или допустимых рисках. Поэтому анализ рисков и возможных угроз целесообразно проводить систематизировано, начиная с рисков функциональной пригодности. Ущерб, вследствие ошибок функциональных требований к проекту программного комплекса может проявляться двумя видами рисков: недостатками достигнутых ***характеристик***, и рисками от нарушения ограничений доступных и используемых ***ресурсов*** в жизненном цикле комплекса программ. Предъявление заказчиком необоснованных требований к функциональной пригодности, проявления в них конфликтов и внутренних противоречий в содержании функций и компонентов, при реально доступных ресурсах и возможных условиях внешней среды применения, могут вызывать наиболее существенный ущерб в ЖЦ.

Цель и функции программного продукта реализуются тогда, когда ***выходная информация*** достигает потребителей – системы или операторов-пользователей, с требуемым содержанием и качеством, достаточным для обеспечения их эффективного применения. Степень покрытия всей выходной информацией: целей, назначения и функций для пользователей, следует рассматривать как ***основную меру рисков функциональной пригодности***. Прослеживание и оценивание адекватности и полноты состава выходной информации снизу вверх к назначению программного продукта должны завершать выбор базовых характеристик функциональной пригодности, независимо от сферы применения системы. При этом некоторые характеристики в реальном проекте могут приобретать значения более высокие, чем действительно требуются, на что нерационально расходуются ресурсы, а другие – не удовлетворять требованиям контракта и технического задания. Для разрешения этого противоречия основное значение имеет деятельность менеджера рисков, который должен ***прогнозировать***, проводить поэтапный анализ, контроль, оценивание и мониторинг возможных и реальных отклонений от требуемых характеристик и используемых ресурсов,

управлять контрмерами и последовательно изменять их для сокращения интегрального риска всей системы.

Сокращение рисков конструктивных характеристик сложных комплексов программ целесообразно проводить с учетом их перечня в стандарте **ISO 9126**. Основной эффект по снижению рисков конструктивных характеристик может достигаться на начальных этапах проектирования, когда возможно предотвращение или сокращение многих из них с минимальными затратами времени и других ресурсов. Для этого в ***технологическом процессе производства*** необходимо использовать ***методы, которые включают***:

- систематизацию, документирование и оценивание эффективности доступных методов, средств и ресурсов контрмер для сокращения рисков функциональной пригодности и выделенных конструктивных характеристик;

- определение приоритетов конструктивных характеристик качества, компонентов и этапов ЖЦ, которые могут иметь потенциальные технические, стоимостные или плановые риски;

- оценивание вероятности каждого вида угроз конструктивной характеристики качества, потенциальной величины и вероятности их возможного негативного воздействия на каждую характеристику функциональной пригодности системы и программного продукта;

- оценивание уязвимости и возможных последствий дефектов каждой конструктивной характеристики и затрат ресурсов для восстановления требуемой функциональной пригодности при проявлении рисков;

- планирование и разработку решений по контрмерам для обеспечения допустимого уровня интегрального риска функциональной пригодности и конструктивных характеристик системы, в том числе возможно за счет изменения требований к программному продукту, системе и/или доступных ресурсов.

Риски ограничений доступных и используемых ресурсов в ЖЦ комплексов программ могут включать:

- экономические риски – превышение разработчиком обоснованных, допустимых по контракту размеров стоимости, трудоемкости и эксплуатационных затрат на программные компоненты и комплекс в целом, которые могут также отражаться на их функциональной пригодности и других характеристиках качества;

- плановые риски – нарушение разработчиком допустимых временных затрат в графиках работ, сроков реализации этапов и проекта в целом, а также распределений задач по подрядчикам, подразделениям и специалистам, что может также увеличивать риски характеристик;

- кадровые риски – недостаточная квалификация и число специалистов, отражающаяся на качестве разработки, совершенствования и/или применения программного продукта;

- технические риски – недостаточность вычислительных ресурсов, несогласованность ресурсов внутренней и внешней среды для реализации основных функций программного продукта;

- технологические риски – недостаточное качество инструментария для автоматизации всего ЖЦ и технологических процессов, предназначенных для обеспечения гарантированного сокращения рисков программного продукта;

- распределение ресурсов на контрмеры для сбалансированного сокращения интегрального риска комплекса программ и ответственности специалистов за реализацию сокращения рисков комплекса программ.

Прямые риски, обусловленные ошибками заданных экономических характеристик, могут вызвать ущерб заказчику при ***завышении стоимости проекта*** относительно реально необходимой, или ущерб разработчикам, если ***стоимость оценена недостаточной*** для его успешной реализации. Эти риски могут уменьшаться при последовательном уточнении размера комплекса программ на этапах формирования требований, предварительного и детального проектирования, однако они не полностью учитывают реальное влияние ограничений ресурсов на процессы и риски разработки и конечного программного продукта.

5. Сокращение и ликвидация опасных рисков, регистрация и утверждение допустимого интегрального риска программного продукта

Для выработки плана анализа рисков и применения контрмер их сокращения должна быть определена и документально установлена ***методика применения последовательного анализа угроз, уязвимостей и изменения проявления рисков*** [3, 7, 10]. После того как менеджеры проекта идентифицируют риски в жизненном цикле разработки, а также уточнят тактику итераций применения контрмер по сокращению их влияния, возникает необходимость в ***идентификации уровня допустимости остаточного риска***. В зависимости от требований к характеристикам, уровни допустимости рисков и контрмер могут варьироваться от качественных оценок до итерационных действий, предполагающих использование альтернативных подходов и дополнительных, последовательных разработок программных компонентов контрмер для сокращения рисков.

Совместное влияние на реализацию требований заказчика к комплексу программ, рисков характеристик программ и ресурсов для их реализации, должно быть ***сбалансировано контрмерами допустимого изменения*** тех и

других видов рисков. В крайнем случае, если интегральный риск остается недопустимо большим, возможна по согласованию с заказчиком, ***корректировка требований к программному продукту или выделяемых ресурсов.*** Эти требования и ограничения могут не полностью выполняться на последующих этапах ЖЦ, что приводит к ущербу у заказчиков, разработчиков и пользователей. Причинами такого ущерба могут быть ошибки, а также завышенные заказчиком требования к характеристикам, которые не могут быть реализованы при выделенных ресурсах, или недостаточное качество технологии и квалификация специалистов – разработчиков, исполняющих проект. Если интегральные риски обусловлены недостаточной величиной одного из видов ресурса, то приходится перераспределять доступные ресурсы или искать заказчику способы увеличения некоторого, критического ресурса. В соответствие с их значениями следует откорректировать и утвердить обновленные, экономически и функционально оправданные, требования к характеристикам, используемым ресурсам и технологии. При любых стратегиях ***наиболее стабильными должны быть требования к функциональной пригодности комплекса программ***.

Для ***выбора критического уровня допустимых рисков*** необходимо исследовать особенности системы, последовательность возможного проявления потенциально опасных событий, любые смягчающие факторы и характеристики, а также природу и частоту возможных негативных последствий идентифицированных угроз в программном продукте и в системе. При этом для сбалансированного снижения интегрального риска, может оказаться эффективным сравнительное, количественное или качественное ранжирование рисков (присвоение им приоритетов) специалистами, хорошо информированными ***в проблемной области*** применения соответствующих систем.

Используя ***планы управления рисками***, менеджер должен осуществлять распределение ресурсов контрмер, направленных на преодоление негативных случайностей. На начальных этапах жизненного цикла, инвестиции в проект постепенно растут вплоть до формулирования конкретных требований. Для обеспечения требуемого качества программных комплексов необходима ***организация контрмер*** в процессе управления рисками. Задача менеджера рисков состоит в выявлении и идентификации источников рисков, противоречий требований характеристик и ресурсов для их реализации и в предложении заказчику и разработчикам ***рациональных и возможных контрмер***, обеспечивающих сокращение рисков до допустимых пределов. Контрмеры для сокращения рисков можно разделить на три типа:

- сокращение или исключение ***первичных причин – угроз***, дефектов и ошибок в компонентах и комплексе программ, обусловленных недостатками их проектирования, разработки или модификации, отражающихся на рисках функциональной пригодности или характеристик комплексов программ;

- сокращение или ликвидация ***уязвимости*** компонентов программ и данных при воздействии на них угроз, дефектов и ошибок, путем введения средств защиты для блокирования их возможного негативного воздействия на риски функционирования и применения комплекса программ;

- непосредственное изменение и сокращение ***последствий проявления рисков*** функциональной пригодности, путем их оперативного обнаружения и ликвидации ущерба при сохранении (возможно) вызывающих их первичных источников и причин.

Крупные программные проекты постоянно усложняются, развиваются и модифицируются, вследствие чего весьма затруднительно рассматривать продукт как единое стабильное целое. В некоторых случаях процессы анализа и сокращения рисков могут быть упрощены [8]. Для этого целесообразно выделять и контролировать только отдельные (2 – 3), наибольшие по величине последствий и по вероятности проявления, риски отклонения от требований, и минимизировать возможный в результате ущерб для функциональной пригодности системы.

В процессах устранения рисков сложных комплексов программ, может участвовать большое число специалистов различных направлений и квалификации, которые, при необходимости, могут объединяться в ***службу сопровождения и управления конфигурацией программного продукта*** [4, 6, 10] (стандарт **ISO 15846**). Структура такой службы зависит от сложности и фазы развития проекта, от структуры предприятия, от ее взаимодействия с заказчиком и субподрядчиками и от ряда других факторов. Необходимо установить ***полномочия специалистов*** для выполнения контрмер и изменений рисков на каждом уровне проекта.

Основной задачей управления конфигурацией является документальное оформление и обеспечение полной наглядности выполняемых изменений, текущей конфигурации программ и данных и степени выполнения требований к их функциональной пригодности и конструктивным характеристикам при сокращении рисков. Другая задача заключается в том, чтобы все лица, работающие над проектом, в любой момент его жизненного цикла использовали достоверную и точную информацию о всех единицах конфигурации проекта и их взаимодействии. Изменения конфигурации комплекса и его компонентов при сокращении рисков должны планироваться и предусматривать действия с четкими разделами:

- ***почему*** и с какой целью производится корректировка программ или данных;

- ***кто*** персонально оценивает риски и выполняет контрмеры, санкционирует и утверждает проведение изменений комплекса или компонентов;

- ***какие*** действия и процедуры должны быть выполнены для реализации изменений рисков и конфигурации;

- ***когда*** по срокам и в координации, с какими другими процедурами ЖЦ следует реализовать определенную корректировку рисков и конфигурации комплекса программ;

- ***как*** и с использованием, каких методов, средств и ресурсов должны быть выполнены запланированные изменения комплекса программ и компонентов.

Для конкретного проекта должны быть определены и зафиксированы ***правила управление конфигурацией, документирования и утверждения интегрального риска версий***, применения административных и технологических процедур их мониторинга на всем протяжении жизненного цикла программного продукта. Организация документирования должна определять стратегию, стандарты, процедуры, распределение ресурсов и планы создания, изменения и применения документов на программы и данные. Для этого в общем случае должны быть выделены ***специалисты***, которые обязаны планировать, утверждать, выпускать, распространять и сопровождать комплекты утвержденных документов. Они должны стимулировать разработчиков программных средств, осуществлять непрерывное, регламентированное документирование процессов и результатов своей деятельности, а также контролировать полноту и качество утвержденного итогового отчета по результатам сокращения и ликвидации рисков программного продукта.

Список литературы

[1] Боэм Б.У. Инженерное проектирование программного обеспечения. Пер. с англ. /Под ред. А.А. Красилова. – М.: Радио и связь, 1985.

[2] Кантор М. Управление программными проектами. Практическое руководство по разработке успешного программного обеспечения. Пер. с англ. – М.: Вильямс. 2002.

[3] Липаев В.В. Анализ и сокращение рисков проектов сложных программных средств. – М.: СИНТЕГ. 2005.

[4] Липаев В.В. Отечественная программная инженерия: фрагменты истории и проблемы. – М.: СИНТЕГ. 2004.

[5] Трубачев А.П., Долинин М.Ю., Кобзарь М.Т. и др. Оценка безопасности информационных технологий. Общие критерии. Под ред. В.А. Галатенко. – М.: СИП РИА, 2001.

[6] Фатрелл Р. Т., Шафер Д. Ф., Шафер Л. И. Управление программными проектами: достижение оптимального качества при минимальных затратах. Пер. с англ. – М.: Вильямс. 2003.

[7] Boehm B.W. Software risk management. IEEE Computer Society Press. Washington. 1989.

[8] Charett R. Applications strategies for risk analysis. N.Y.: McGraw – Hill. 1989.

[9] Higuera R., Haimes Y. Software risk management. Pittsburg. Software engineering institute, Cornegie Mellon University. – 1996.
[10] Karolak D. W. Software engineering risk management. IEEE Computer Society Press. Washington. 1996.

Risks of Design and Development of Portable Software Products

V.V. Lipaev
lip@ispras.ru

Abstract. The paper introduces main notions and properties of risks of software complexes. Factors and types of such risks are considered. Preparation of input data for analysis, prediction, and reduction of complex software is discussed. Then the paper describes discovering, identification, and analysis of threads and risks in complex software. Finally, methods of http://www.multitran.ru/c/m.exe?a=110&t=22064_2_1&sc=0dangerous risks reduction and removing, registration and approving acceptable integral risk of software products are considered. водятся основные понятия и свойства рисков комплексов программ.

Keywords: software product, its quality and types of risks; defects of software, threads of risks and their identification during software development and testing; registration, reduction and removing complex software risks.

Решение проблемы NULL в запросах к реляционной базе данных, используя операторы реляционной алгебры A

И.В. Блудов.
ivan.bludov@gmail.com

Аннотация. В операторах ограничения предлагается логические выражения интерпретировать как реляционные. Точнее, считается, что операция реляционного ограничения (*R WHERE b*) над отношением *R* по некоторому логическому выражению *b* может быть представлена как соединение (*R<AND>B*) заданного отношения R с реляционным выражением *B*, полученным из исходного логического выражения *b* заменой логических операторов AND, OR и NOT на соответствующие реляционные операторы <AND>, <OR> и <NOT>. Тогда для некоторого кортежа *T* определим значение атрибута A как отношение с одним кортежем и одним значением интересующего нас атрибута – RELATION<A>{{a}}. Значение атрибута, указанное как NULL, в качестве значение «неизвестно», определим как отношение с заголовком из интересующего нас атрибута и телом, содержащим всевозможные значения типа атрибута *A* – RELATION<A>{…}. Сравнение значений атрибутов на равенство будет выглядеть как соединение таких значений атрибутов, представленных отношениями. Кортеж *T*, который может быть определен как декартовое произведение всех своих атрибутов, будет теперь представлять отношение R_T. Истинность такого кортежа *T*, представленного отношением R_T, по заданному логическому выражению *b*, означает истинность квантора всеобщности над значениями R_T по выражению *b*, что в свою очередь означает равенство соединения (R_T <AND> *B*) и $R_T - (R_T$ <AND> *B*) = R_T.

Ключевые слова: реляционная модель данных; NULL; отсутствующая информация; реляционная алгебра A.

1. Введение

В основу данной статьи легла извечная проблема несостоятельности трехзначной логики, используемой в SQL для поддержки механизма работы с отсутствующей информацией. Кристофер Дейт в работе [1] показывает невозможность использования n-значных логиках. В частности, он критикует предложенные Тэдом Коддом таблицы истинности элементарных логических операторов трехзначной логики, используемых в SQL, а также другие попытки Кодда введения таблиц истинности элементарных логических операторов для четырехзначной логики. Основная причина такой несостоятельности n-значных логиках состоит в том, что множества тождеств

и противоречий (выражения, вычисляемых как *истинна* или *ложь* при любых значениях операнд), используемых в разных логиках, различны. В частности, оптимизаторы SQL-запросов трансформируют выражения, используя некорректное, с точки зрения двухзначной логики (реального мира), множество тождеств и противоречий. В результате чего такие трансформированные реляционные выражения возвращают некорректные, с точки зрения реального мира, результаты.

В качестве одного примера несостоятельности трехзначной логики в SQL Дейт приводит в [2],[3] следующий запрос. Для базы данных

Поставщики S

SNO*	S_CITY
S1	‘London’

Детали P

PNO*	P_CITY
P1	NULL

«получить пары SNO-PNO, для которых города поставщика и детали различны или городом детали не является Париж (либо оба эти условия)». Предлогается следующая «очевидная формулировка этого запроса на языке SQL»: SELECT S.SNO, P.PNO FROM S, P WHERE S.CITY <> P.CITY OR P.CITY <> 'Paris'. В результате даже возник спор между Клодом Рубинсоном [4], с одной стороны, и Дейтом[5] и Джоном Грантом[6], с другой.

Чтобы подчеркнуть «вечность» описанной проблеме, можно повторить мысль Дейта о том, что время от времени люди из сообщества баз данных для решения описанной проблемы пытаются придумать какую-нибудь k-значную логику и определить для нее таблицы истинности основных операторов, попадая при этом в распространённую ловушку. Учитывая это, то есть оставаясь в рамках двухзначной логики, при исследовании проблемы отсутствующей информации я начинаю с понятия истинности логического выражение и того, чем, по сути, является отношение – множеством истинных утверждений [2], [3]. Высказывается идея о том, логические выражения можно рассматривать как подмножество другого класса выражений – реляционных выражений. В то время, как логические выражения оперируют одиночными утверждениями, реляционные выражения оперируют множествами таких утверждений. Используются операторы реляционной алгебры A [7], которые, по сути, являются логическими операторами над множеством истинных утверждений, то есть отношениями. В большей части статьи разбирается пример, который показывает, как можно в операторе реляционного ограничения (*R WHERE b*) логическое выражение *b* представить реляционным выражением, заменяя логические операторы OR, AND, NOT на соответствующие реляционные операторы алгебры A <OR>, <AND>, <NOT>. Далее известное значение атрибута A определяется, как отношение с одним кортежем и интересующим значением (RELATION<A>{{a}}); NULL в качестве значение неизвестно определяется, как отношение, содержащие все возможные значения атрибута *A* (RELATION<A>{…}); кортеж – как

декартовое произведение всех своих атрибутов. Это позволяет получить устойчивое решение проблемы отсутствующей информации.

2. Представление логического выражения в операции ограничения как реляционного выражения

2.1.

Рассмотрим операцию ограничения над отношением REL WHERE $P_{(REL)}$, где $P_{(REL)}$ – некоторое логическое выражение над атрибутами отношения REL. Будем рассматривать только некоторое подмножество из всех возможных логических выражений: пусть $P_{(REL)}$ определяется следующим образом:

<логическое_выражение>::= NOT <логическое_выражение> |

<логическое_выражение> AND <логическое_выражение> |

<логическое_выражение> OR <логическое_выражение> |

<логическое_выражение> = <логическое_выражение> |

<логическое_выражение> <> <логическое_выражение> |

<сравнение_атрибута_и_константы> | <сравнение_двух_атрибутов> |

<логическая_константа>

Отметим, что можно определить операторы сравнения логических выражений:

A = B как *(A and B) or (not (A) and not (B))*;
A <> B как *(not(A) and B) or (A and not (B)).*

Тогда определение $P_{(REL)}$ можно сведено до следующего:

<логическое_выражение>::= NOT <логическое_выражение> |

<логическое_выражение> AND <логическое_выражение> |

(1) <логическое_выражение> OR <логическое_выражение> |
<сравнение_атрибута_и_константы> |<сравнение_двух_атрибутов>|
<логическая_константа>

2.2.

Запишем некоторые общеизвестные правила преобразований в терминах новой реляционной алгебры A:

- *X WHERE (a OR b) =*

  (X WHERE *a*) UNION (X WHERE *b*) = *(X WHERE a) <OR> (X WHERE b)*

- *X WHERE (a AND b) =*

(X WHERE *a*) INTERSECT (X WHERE *b*) = *(X WHERE a) <AND> (X WHERE b)*

- Известно, что любое выражение вида (X WHERE *b*) может быть представлено, как соединение отношения *X* с некоторым другим отношением *B* таким, что заголовки (типы) отношений *X* и *B* одинаковые, а кортежи отношения *B* представляет все возможные истинные значения логического выражения *b*.

  X WHERE b <=> X <AND> B.

- Закон Де Моргана:
  - для логических выражений:

    NOT (*a* AND *b*) = NOT (*a*) OR NOT (*b*);

    NOT (*a* OR *b*) = NOT (*a*) AND NOT (*b*);
  - для реляционных выражений:

    <NOT> (A <AND> B) = (<NOT> A) <OR> (<NOT>B);

    <NOT> (A <OR> B) = (<NOT> A) <AND> (<NOT>B);

- Свойство дистрибутивности:
  - для логических выражений:

    (*a* AND *b*) OR *c* = (*a* OR *c*) AND (*b* OR *c*);

    (*a* OR *b*) AND *c* = (*a* AND *c*) OR (*b* AND *c*);
  - для реляционных выражений:

    (A <AND> B) <OR> C = (A <OR> C) <AND> (B <OR> C);

    (A <OR> B) <AND> C = (A <AND> C) <OR> (B <AND> C);

- *X WHERE (NOT b)* = X MINUS (X WHERE *b*) = X <AND> <NOT>(X WHERE *b*) = X <AND> <NOT> (X <AND> B) = X <AND> ((<NOT>X) <OR> (<NOT>B)) = (X <AND> <NOT>X) <OR> (X <AND> <NOT>B) = *X <AND> <NOT> B*, поскольку (X <AND> <NOT> X) – пустое отношение.

2.3.

Используя перечисленные выше правила (главным образом, правила для *X WHERE (a OR b); X WHERE (a AND b); X WHERE (NOT b)*) для преобразования исходного выражения *REL WHERE* $P_{(REL)}$, мы получим реляционное выражение с соединениями, пересечениями и отрицаниями

простых ограничений вида *REL WHERE P(Atr_i), REL WHERE P(Atr_i, Atr_j), REL WHERE P()*, то есть с простыми логическими выражениями вида:

<логическое_выражение>::= <сравнение_атрибута_и_константы> |

<сравнение_двух_атрибутов> |

<логическая_константа>

Приведем пример:

X WHERE (NOT((A=B)=(B=b)) AND (((C=c) AND FALSE) OR TRUE)) **(2)**

=

X <AND>

<NOT> (

(X WHERE (A=B) <AND> X WHERE (B=b)) <OR> **(3)**

(X <AND> <NOT> (X WHERE (A=B)) <AND> <NOT> (X WHERE (B=b)))

)

<AND> ((X WHERE (C=c) <AND> X WHERE FALSE) <OR> X WHERE TRUE)

2.4.

Далее простые ограничения вида REL WHERE P(Atr_i) будем рассматривать как соединение REL<AND>PAtr_i, где кортежи отношения PAtr_i представляет все возможные истинные значения логического выражения P(Atr_i):

- Сравнение атрибута и константы: (X WHERE A<cmp_op> const) = (X <AND> CONST)
- Сравнение двух атрибутов: (X WHERE A<cmp_op>B) = (X <AND> AB)
- Логическая константа:

  (X WHERE TRUE) = (X <AND> TABLE_DEE) и

  (X WHERE FALSE) = (X <AND> TABLE_DUM)

Тогда реляционное выражение (3) преобразуется к следующему виду:
X <AND>

<NOT>(

(X <AND> AB <AND> X <AND> B) <OR> **(4)**

(X <AND> <NOT> (X <AND> AB) <AND> <NOT> (X <AND> B))

)

<AND> ((X <AND> C <AND> X <AND> TABLE_DUM) <OR> X <AND> TABLE_DEE)

2.5.

Теперь, когда у исходное отношение *REL WHERE* $P_{(REL)}$ представлено в виде (4), «вытащим» (вынесем за скобки) конъюнкт REL. То есть представим выражение (4) в виде соединения REL с выражением, не содержащим отношение REL. Отметим, что это возможно из-за дистрибутивности операторов <AND> и <OR>, а также из-за того, что реляционный оператор <NOT> (используемый в выражение *X <AND> (<NOT> EXP)*) хоть и возвращает дизъюнкцию конъюнктов, содержащих как X, так и <NOT> X, конъюнкты с <NOT> X сократятся, поскольку им всегда будет предшествовать соединение с X.

Тогда реляционное выражение (4) преобразуется к следующему виду:
X <AND>

<NOT>(AB <AND> B <OR> <NOT> AB <AND> <NOT> B) **(5)**

<AND> (C <AND> TABLE_DUM <OR> TABLE_DEE))

Если в исходном ограничении (2) представить равенство
(A=B) = (B=b) как (((A=B) AND (B=b)) OR (NOT (A=B) AND NOT (B=b))),

то получим выражение
X WHERE (NOT (((A=B) AND (B=b)) OR (NOT (A=B) AND NOT (B=b))) AND (((C=c) AND FALSE) OR TRUE)) **(6),**

которое очень схоже с выражением (5):

X WHERE NOT (((A=B) AND (B=b)) OR (NOT (A=B) AND NOT (B=b))) AND ((C=c) AND FALSE OR TRUE)	X <AND> <NOT>(AB <AND> B <OR> <NOT> AB <AND> <NOT> B) <AND> (C <AND> TABLE_DUM <OR> TABLE_DEE))

На схожести выражений (5) и (6), полученных в результате приведенных выше преобразований, основана одна из главных идей данной статьи о ***возможности интерпретации логического выражения как реляционного***

выражения. То есть оператор реляционного ограничения *(X WHERE b)* с некоторым логическим выражением *b* вида (1) может быть представлен как соединение *X* и некоторого реляционного выражения R_b – *X <AND>* R_b, где реляционное выражение R_b получено из логического выражения *b* заменой логических операторов NOT, AND, OR на соответствующие реляционные операторы <NOT>, <AND>, <OR>; сравнения атрибутов и значений/атрибутов на отношения, представляющих данные сравнения на множестве всевозможных значений; логических констант TRUE и FALSE на отношения-константы TABLE_DEE и TABLE_DUM соответственно. Кроме того, поскольку в реляционной алгебры А любое ограничение *(X WHERE b)* может быть представлено как соединение *X* с некоторым отношением *B*, кортежи которого представляет все возможные истинные значения логического выражения *b*, по построению *B* является результатом вычисления R_b. Таким образом, R_b является функций ограничения над всеми возможными значениями *X*, а *X <AND>* R_b является вызовом этой функции для заданного значения отношения *X*.

2.6.

Вернемся к выражению (3) и пункту, для которого выполним подстановку следующего вида:

- Сравнение атрибута и константы:

  (X WHERE A<cmp_op>const) = (X <AND> X{A} <AND> CONST)

- Сравнение двух атрибутов:

  (X WHERE A<cmp_op>B) = (X <AND> X{A,B} <AND> AB)

- Логическая константа:

  (X WHERE TRUE) = (X <AND> TABLE_DEE) и

  (X WHERE FALSE) = (X <AND> TABLE_DUM)

Получим:

X <AND>
<NOT>(
(X WHERE (A=B) <AND> X WHERE (B=b)) <OR>
(X <AND> <NOT> (X WHERE (A=B)) <AND> <NOT> (X WHERE (B=b)))
)
<AND> ((X WHERE (C=c) <AND> X WHERE FALSE) <OR> X WHERE TRUE)
=
X <AND>
<NOT>(
(X <AND> X{A,B} <AND> AB <AND> X <AND> X{B}<AND> B) <OR>
(X <AND> <NOT> (X <AND> X{A,B} <AND> AB) <AND> <NOT> (X <AND> X{B} <AND> B))

)
<AND> ((X <AND> C <AND> X <AND> TABLE_DUM) <OR> X <AND> TABLE_DEE)

Над полученным выражением вновь «вытащим» конъюнкт *X*:

X <AND>

<NOT>(

(X <AND> X{A,B} <AND> AB <AND> X <AND> X{B}<AND> B) <OR>

(X <AND> <NOT> (X <AND> X{A,B} <AND> AB) <AND> <NOT> (X <AND> X{B} <AND> B))

)

<AND> ((X <AND> C <AND> X <AND> TABLE_DUM) <OR> X <AND> TABLE_DEE)

=

X <AND>

<NOT>(

((X{A,B} <AND> AB)<AND>(X{B}<AND> B)) <OR>

(<NOT> (X{A,B} <AND> AB) <AND> <NOT> (X{B} <AND> B)) **(7)**

)

<AND> (C <AND> TABLE_DUM <OR> TABLE_DEE)

Таким образом, выражение (7) в сравнении с выражение (5), отчасти уже содержит вызов функции ограничения. То есть если

<NOT> (AB <AND> B <OR> <NOT> AB <AND> <NOT> B)

<AND> (C <AND> TABLE_DUM <OR> TABLE_DEE)) -

функция ограничения в общем виде, тогда

<NOT>(

((X{A,B} <AND> AB)<AND>(X{B}<AND> B)) <OR>

(<NOT> (X{A,B} <AND> AB) <AND> <NOT> (X{B} <AND> B))

)

<AND> (C <AND> TABLE_DUM <OR> TABLE_DEE)

уже отчасти содержит вызов общей функции ограничения с интересующими нас значениями, и может быть рассмотрен как частный случай функции ограничения.

2.7.

Вернемся к рассмотрению операцию ограничения *REL WHERE P($_{REL}$)*. Отношение REL может быть представлено как объединение множества всех своих кортежей, точнее как объединение отношений, содержащих только один кортеж из REL, для каждого кортежа из REL.

(UNION(R_{Tuple1},R_{Tuple2},…,R_{Tuple_N}))WHERE P($_{REL}$)	(<OR>(R_{Tuple1},R_{Tuple2},…,R_{Tuple_N})) <AND> P_{REL}

Используя свойство дистрибутивности операций ограничения и объединения, получим

UNION(	<OR>(
R_{Tuple1} WHERE P($_{REL}$),	R_{Tuple1} <AND> P_{REL},
R_{Tuple2} WHERE P($_{REL}$),	R_{Tuple2} <AND> P_{REL},
…,	…,
R_{Tuple_N} WHERE P($_{REL}$))	R_{Tuple_N} <AND> P_{REL})

Очевидно, что ограничение *R_{Tuple_K} WHERE P($_{REL}$)* (*R_{Tuple_K} <AND> P_{REL}*) возвращаете либо пустое отношение, если условие ложно; *либо входное отношение, если условие истинно*. (*Примечание*: в дальнейшем будем называть отношения вида *R_{Tuple_K} – отношениями-кортежами*)

2.8.

Теперь применим ограничения из пункта **2.6**, где сравнения атрибут представлены как:

- Сравнение атрибута и константы:
  (X WHERE A<cmp_op>const) = (X <AND> X{A} <AND> CONST)
- Сравнение двух атрибут:
  (X WHERE A<cmp_op>B) = (X <AND> X{A,B} <AND> AB)

индивидуально к отношениям-кортежам X_{Tuple_K} (R_{Tuple_K} из пункта **3.7**).

Таким образом, проекции вида X_{Tuple_K}{A} дадут отношение с одним кортежем, содержащим одно значение для атрибута A – RELATION<A>{{A a}} (*примечание*: в дальнейшем будем называть такие отношения – *отношениями-значениями*). Проекции X_{Tuple_K}{A, B}, поскольку X_{Tuple_K} содержит только один кортеж, можно рассматривать как соединение таких отношений-значений X_{Tuple_K}{A} <AND> X_{Tuple_K}{B}.

Теперь сравнения атрибутов будут выглядеть следующим образом.

- Сравнение атрибута и константы:

X_{Tuple_K} WHERE A<cmp_op>const = X_{Tuple_K} <AND> X_{Tuple_K}{A} <AND> CONST

- Сравнение двух атрибут:

X_{Tuple_K} WHERE A<cmp_op>B = X_{Tuple_K} <AND> X_{Tuple_K}{A} <AND> AB <AND> X_{Tuple_K}{B}

Если вернуться от абстрактных отношений A и AB, содержащих всевозможные значения, используемых в определения сравнений атрибутов, тогда сравнение несут смысл эквисоединения над входными отношениями-значениями (опишем с привлечением терминов SQL):

- Сравнение атрибута и константы:

X_{Tuple_K} WHERE A<cmp_op>const =
X_{Tuple_K} <AND> (X_{Tuple_K}{A} JOIN RELATION<CONST>{{const}} ON (A<cmp_op>CONST)){A}

- Сравнение двух атрибут:

X_{Tuple_K} WHERE A<cmp_op>B =
X_{Tuple_K} <AND> (X_{Tuple_K}{A} JOIN X_{Tuple_K}{B} ON (A<cmp_op>B))

Конечно, иногда может быть удобно для ограничения X_{Tuple_K} *WHERE A<cmp_op>B* использовать отрицание от обратного оператора сравнения X_{Tuple_K} *WHERE NOT(A<inverse_cmp_op>B)*. Который можно упростить:

X_{Tuple_K} WHERE A<cmp_op>B =

X_{Tuple_K} WHERE NOT (A<inverse_cmp_op>B) =

X_{Tuple_K} <AND> <NOT> (X_{Tuple_K} {A} <AND> <INVERSE_AB> <AND> X_{Tuple_K} {B})

2.9.

Все описанные выше утверждения верны для реляционной модели без NULL. Теперь же введем NULL и покажем операции сравнения над ними.

Основная идея состоит в том, чтобы рассматривать NULL для некоторого атрибута $A_{UNKNOWN}$, в случае значение не известно (значение не определено, значение не достоверно, значение не предоставлено), как отношение с заголовком, содержащим только интересующий нас атрибут $A_{UNKNOWN}$, и телом, содержащим всевозможными значениями для заданного типа атрибута $A_{UNKNOWN}$. Более точно, нас в большей мере интересует проекция X_{Tuple_K} { $A_{UNKNOWN}$ } из пункта **2.8**, то есть результатом такой проекции будет отношение с заголовком из атрибута $A_{UNKNOWN}$, и телом со всевозможными значениями для типа атрибута $A_{UNKNOWN}$.

Что касается NULL для атрибута A_{NA}, в качестве значение недопустимо (значение не существует, значением является пустое множество), изначально планировалось определить его как пустое отношение с заголовком из атрибута A_{NA}, однако, как оказалось такое поведение не всегда верно, и даже может быть упрощено. Рассмотрим его чуть позже.

2.10.

С введением NULL, в случае значение не известно, отношение X_{Tuple_K}, изначально содержавший только один кортеж, теперь же содержит множество кортежей, которое получилось как декартовое произведение своих атрибутов. Ограничение X_{Tuple_K} *WHERE* $P(_{REL})$, как говорилось в пункте **3.7** «*возвращаете входное отношение, если условие истинно*», принимает смысл истинности квантора всеобщности для отношения X_{Tuple_K} над выражением $P(_{REL})$.То есть если результат X_{Tuple_K} *WHERE* $P(_{REL})$ равен исходному отношению X_{Tuple_K}, это означает истинность квантора всеобщности, и потому кортеж *Tuple_K* из отношения REL (см. пункт VII), добавляется в результат ограничение *REL WHERE* $P(_{REL})$.

3. Примеры

Рассмотрим приведенный во введении пример Дейта, показывающий несостоятельность трехзначной логики в SQL.

Поставщики и детали

Поставщики S

SNO*	S_CITY
S1	‘London’

Детали P

PNO*	P_CITY
P1	NULL

Запрос

SELECT S.SNO, P.PNO FROM S, P

WHERE S.CITY <> P.CITY OR P.CITY <> ’Paris’

Представим соединение кортежей S и P как отношение TUPLE_S_JOIN_P, и далее будем использовать операторы реляционной алгебры, а не SQL.

TUPLE_S_JOIN_P WHERE (S_CITY<>P_CITY) OR (P_CITY<>’Paris’)

=

TUPLE_S_JOIN_P <AND>

(<NOT> ((TUPLE_S_JOIN_P {S_CITY} RENAME S_CITY AS P_CITY) <AND> TUPLE_S_JOIN_P {P_CITY})

<OR>

<NOT> (TUPLE_S_JOIN_P {P_CITY} <AND> RELATION< P_CITY> {{‘Paris’}}))

(см. Преобразование 3)

Выполним подстановку значений P_CITY

3.1. Значение P_CITY определено

Отношение TUPLE_S_JOIN_P:

SNO	S_CITY	PNO	P_CITY
S1	‘London’	P1	‘Paris’

TUPLE_S_JOIN_P =
RELATION<SNO> {{S1}} <AND> RELATION<S_CITY> {{‘London’}} <AND>
REALTION {{P1}} <AND> RELATION<P_CITY>{{‘Paris’}}

TUPLE_S_JOIN_P WHERE (S_CITY<>P_CITY) OR (P_CITY<>’Paris’)
=
TUPLE_S_JOIN_P <AND>
(<NOT>(RELATION<P_CITY>{{‘London’}} <AND> RELATION<
P_CITY>{{‘Paris’}})
<OR>
<NOT>(RELATION<P_CITY>{{‘Paris’}} <AND>
RELATION<P_CITY>{{‘Paris’}}))
=
TUPLE_S_JOIN_P <AND>
(<NOT> RELATION<P_CITY>{} <OR>
<NOT>RELATION<P_CITY>{{‘Paris’}})
=
TUPLE_S_JOIN_P <AND>
(RELATION<P_CITY>{…} <OR> RELATION<P_CITY>{…/{‘Paris’}})

где RELATION< P_CITY>{…} означает отношение, содержащее всевозможные кортежи; а RELATION<P_CITY>{…/{‘Paris’}} означает отношение, содержащее всевозможные кортежи за исключением кортежа {‘Paris’}

=
TUPLE_S_JOIN_P <AND> RELATION<P_CITY>{…}

Очевидно, что рассмотренное ограничение вернет отношение TUPLE_S_JOIN_P, что будет означать истинность квантора всеобщности – то есть для всех кортежей отношения TUPLE_S_JOIN_P верно выражение (*S_CITY <> P_CITY OR P_CITY <> ’Paris’)*

3.2. Значение P_CITY неопределенно

Отношение TUPLE_S_JOIN_P:

SNO	S_CITY	PNO	P_CITY
S1	‘London’	P1	UNKNOWN

TUPLE_S_JOIN_P =
RELATION<SNO> {{S1}} <AND> RELATION<S_CITY> {{‘London’}} <AND> REALTION {{P1}} <AND> RELATION<P_CITY>{…}

TUPLE_S_JOIN_P <AND>
(<NOT>(RELATION< P_CITY>{{‘London’ }} <AND> RELATION<P_CITY>{…})
<OR>
<NOT>(RELATION<P_CITY>{…} <AND> RELATION<P_CITY>{{‘Paris’}}))

=
TUPLE_S_JOIN_P <AND>
(<NOT> RELATION<P_CITY>{{‘London’}} <OR> <NOT>RELATION<P_CITY>{{‘Paris’}})
=
TUPLE_S_JOIN_P <AND>
(RELATION<P_CITY>{…/’London’} <OR> RELATION<P_CITY>{…/{‘Paris’}})
=
RELATION<SNO, S_CITY, PNO, P_CITY>{{S1, ’London’, P1, …}} <AND> RELATION<P_CITY>{…}

Данное ограничение также вернет отношение TUPLE_S_JOIN_P, то есть истинность квантора всеобщности для значений RELATION<SNO, S_CITY, PNO, P_CITY>{{S1, ’London’, P1, …}} над выражением (S_CITY <> P_CITY OR P_CITY <> ’Paris’)

3.3. Значение P_CITY недопустимо

Отношение TUPLE_S_JOIN_P:

SNO	S_CITY	PNO	P_CITY
S1	‘London’	P1	N/A

Как было упомянуто ранее в пункте **2.10**, изначально планировалось определить NULL, в качестве значение недопустимо, как пустое отношение, с

заголовком соответствующего атрибута. Поскольку сравнение на равенство атрибутов представляется как соединение *отношений-значений* сравниваемых атрибутов, а соединение с пустым отношением всегда даст пустое отношение, что означало бы ложь. Однако сравнение значения двух атрибутов с недопустимым значением, тоже даст пустое отношение, хотя ожидалась истина. Кроме того, не очевидно как представлять TUPLE_S_JOIN_P. Поскольку TUPLE_S_JOIN_P = RELATION<SNO> {{S1}} <AND> RELATION<S_CITY> {{'London'}} <AND> REALTION {{P1}} <AND> RELATION<P_CITY>{}, естественно даст пустое отношение, а для пустого отношения квантор всеобщности всегда истинен, даже если условие ложно. Однако в действительности случай NULL, как значение недопустимо, более вырожденный, чем случай NULL, как значение неизвестно, поскольку возвращает истину, только если оба значения недопустимы. Потому результат его сравнение может быть просто представлен логическими константами, или же TABLE_DEE и TABLE_DUM, если мы работаем с отношениями. Остается открытым вопрос из пункта **2.10**, как определить добавлять ли кортеж *Tuple_K* из отношения *REL* в результат ограничение *REL WHERE* $P(_{REL})$, где равенство отношения X_{Tuple_K} и ограничения X_{Tuple_K} *WHERE* $P(_{REL})$ означает истинность квантора всеобщности для отношения-кортежа X_{Tuple_K}. Истинность же квантора всеобщности в случае NULL, значение недопустимо, предлагается ввести как истинность равенство проекции $(X_{Tuple\_K}$ *ALL BUT* $(A_{NA}, \ldots))$ и ограничения $(X_{Tuple\_K}$ *ALL BUT* $(A_{NA}, \ldots)$ *WHERE* $P(_{REL}))$.

Так результат (TUPLE_S_JOIN_P {S_CITY} RENAME S_CITY AS P_CITY) <AND> TUPLE_S_JOIN_P {P_CITY} определим как TABLE_DUM, результат (TUPLE_S_JOIN_P {P_CITY} <AND> RELATION< P_CITY> {{'Paris'}}) также определим как TABLE_DUM. Тогда

(TUPLE_S_JOIN_P ALL BUT P_CITY) <AND>

(<NOT>TABLE_DEE <OR> <NOT>TABLE_DUM)

=

(TUPLE_S_JOIN_P ALL BUT P_CITY) <AND> TABLE_DEE

=

(TUPLE_S_JOIN_P ALL BUT P_CITY)

Очевидно что, квантор всеобщности истинен.

(см. Примеры)

4. Смежные вопросы

4.1. Сравнения атрибута на равенство больше/меньше

Рассмотрим вопросы сравнения атрибута и константы на равенство больше/меньше. Из пункта **2.8** операторы сравнения атрибута и константы определены следующим образом:

X_{Tuple_K} WHERE A<cmp_op>const = X_{Tuple_K} <AND> X_{Tuple_K}{A} <AND> CONST, или же
X_{Tuple_K} WHERE A<cmp_op>const =
X_{Tuple_K} <AND> (X_{Tuple_K}{A} JOIN RELATION<CONST>{{const}} ON (A<cmp_op>CONST)){A}

Пусть атрибут A имеет тип T, для которого определены операции сравнения на больше/меньше. По определению каждый тип определяет множеством значений, принадлежащих данному типу. Операция ограничения в реляционной алгебре A предполагает существование отношения CONST, кортежами которого являются все значения T, удовлетворяющие условию *t<cmp>const*. Результатом соединение $X_{Tuple\_K}\{A\}$ *<AND> CONST*, поскольку $X_{Tuple\_K}\{A\}$ представляет отношение с всевозможными значениями над типом T, очевидно будет отношение *CONST*.

Запрос 1

Rel:

A REAL
UNKNOWN

Rel *WHERE* A>3 OR A<4 =
Rel <AND> ((Rel{A} <AND> $REAL_{A>3}$) <OR> (Rel{A} <AND> $REAL_{A<4}$))
где $REAL_{A>3}$ означает отношение RELATION<A REAL>{A|A>3} =
Rel <AND> (RELATION<A>{…} <AND> $REAL_{A>3}$
)<OR>(RELATION<A>{…}<AND>$REAL_{A<4}$)
=
Rel <AND> ($REAL_{A>3}$ <OR> $REAL_{A<4}$) =
Rel <AND> REALITON<A > {…}

Квантор всеобщности истинен.

Запрос 2

TYPE REAL_1_TO_10 IS REAL CONSTRAINT R>=1 AND R<=10;
Rel:

A REAL_1_TO_10
UNKNOWN

Rel WHERE A>0 =

Rel <AND> Rel{A} <AND> $REAL_{A>0}$ =
Rel <AND> RELATION<A REAL_1_TO_10>{…} <AND> $REAL_{A>0}$=
Rel <AND> RELATION<A REAL_1_TO_10>{…}

Квантор всеобщности истинен

4.2. Сравнение двух неопределённых значений

Рассмотрим сравнение двух неопределённых значений. Очевидно, что сравнения двух атрибутов, описанных в пункте **2.8** как:

X_{Tuple_K} WHERE A<cmp_op>B = X_{Tuple_K} <AND> X_{Tuple_K}{A} <AND> AB <AND> X_{Tuple_K}{B},

или же

X_{Tuple_K} WHERE A<cmp_op>B = X_{Tuple_K} <AND> (X_{Tuple_K}{A} JOIN X_{Tuple_K}{B} ON (A<cmp_op>B)),

В случае, когда атрибуты представлены неопределённым значением, проекции X_{Tuple_K}{A} и X_{Tuple_K}{B} представляют множество всевозможных значений, тогда возникает сложность в представлении отношения AB, не говоря о вычислительных сложностях. Поскольку нам к тому же следует проверять истинность квантора всеобщности: «каждый человек выше любого другого» – то предполагается, что такое сравнение неопределённых значений атрибутов будет ложью. То есть соединения: (*X_{Tuple_K}{A} <AND> AB <AND> X_{Tuple_K}{B}*) или же (*X_{Tuple_K}{A} JOIN X_{Tuple_K}{B} ON (A<cmp_op>B)*) – будут возвращать пустые отношения. Конечно, это не позволит показать истинность выражения: «Все либо выше других, либо ниже других, либо одинокого роста». Также следует помнить о типах атрибутов, поскольку неопределённые значения определяются как множество всевозможных значений над заданным типом: так очевидно, что какой-либо человек выше, скажем, к примеру, кого-либо муравья.

Также высказывается идея, что при добавлении неопределённого значения в базу данных, в системе автоматически вводится некий уникальный идентификатор для нового неопределённого значения. Тогда становится возможным сравнение неопределенного значения с самим собой, что дает истину. Более того, если такое соотношение уникального идентификатора и неопределённого значения существует, тогда система базы данных сможет всегда отслеживать миграцию такого неопределенного значения, и выполнять корректное сравнивание их, даже если они находятся в разных отношениях.

5. Заключение

В данной статье я попытался доказать, что ограничении (*R WHERE B*) некоторого отношения R по логическому выражению b может быть записано

как соединение R<AND>B, где B реляционное выражение, полеченное из логического выражения b заменой, логических операторов AND, OR, NOT на соответствующие операторы новой реляционной алгебры <AND>, <OR>, <NOT>, а сравнение атрибутов, представленных отношением, представляется эквисоединением. Тогда я определяю NULL, в качестве значение неизвестно, как отношение со всевозможными значениями над соответствующим типом атрибута. Кортежа T из R, содержащего неопределённое значение для некоторого атрибута, представляется как декартовое произведение своих атрибутов – R_T. Истинность такого кортежа T, представленного отношением R_T, по заданному логическому выражению b, означает истинность квантора всеобщности над значениями R_T по выражению b, что означает равенство соединения (R_T <AND> B) и R_T. Также я рассмотрел некоторые практические вопросы сравнения неопределённого значения атрибута, представленного отношением со всевозможными значениями.

Список литературы

[1] C. J. Date. “Why Three- and Four-Valued Logic Don’t Work” in “Date on Database. Writings 2000–2006”. Apress, 2006. Впервые опубликована на сайте http://www.dbdebunk.com (February 2006).

[2] C. J. Date. Database in Depth: Relational Theory for Practitioners. Sebastopol, Calif.: O’Reilly Media, Inc. (2005).

[3] C. J. Date. SQL and Relational Theory: How to Write Accurate SQL Code. O’Reilly Media, Inc. (2009).

[4] Claude Rubinson. Nulls, Three-Valued Logic, and Ambiguity in SQL : Critiquing Date’s Critique. SIGMOD Record, December 2007 (Vol. 36, No. 4). См. также перевод: http://citforum.ru/database/articles/nulls/

[5] C. J. Date. A Critique of Claude Rubinson’s Paper Nulls, Three - Valued Logic, and Ambiguity in SQL: Critiquing Date’s Critique. SIGMOD Record, Vol. 37, No. 3, September 2008. См. также перевод: http://citforum.ru/database/articles/date_vs_rubinson/

[6] John Grant. Null Values in SQL. SIGMOD Record, Vol. 37, No. 3, September 2008. См. также перевод: http://citforum.ru/database/articles/grant_vs_rubinson/

[7] C. J. Date, Hugh Darwen. “Foundation for Future Database Systems: The Third Manifesto”, Addison-Wesley Pub Co; 2nd edition (2000).
Имеется перевод: Дейт К., Дарвен Х. Основы будущих систем баз данных. Третий манифест. 2-е изд. (под ред. С. Д. Кузнецова). М.: Янус-К, 2004.

Solving NULL issue in queries to a relational database using the operators of relational algebra A.

Ivan. V. Bludov
ivan.bludov@gmail.com

Annotation. Suggest interpreting logical expressions in the relational restriction operator as relational expressions. To be more precise, the relational restriction operator (*R WHERE b*) over the relation *R* on some logical expression *b* can be represented as join (*R<AND>B*) of given relation *R* and relational expression *B*, which is derived from original logical expression b replacing of logical operators AND, OR, NOT for corresponding relational operators <AND>, <OR>, <NOT>. Then for some tuple T we define a value of attribute A as the relation with the single tuple with the single value of concerned attribute – RELATION<A> {{a}}. The value of attribute marked as NULL meaning the unknown value we define as the relation with a head including concerned attribute and the body including all possible values over the type of attribute A – REALTIONAL<A>{…}. The equality values of attributes are definite as equijoin of these attributes represented as relation. The tuple *T*, which could be presented as Cartesian product over all its attributes, is also definite as a relation R_T. The validity of this tuple *T* represented as relation R_T over the original logical expression b means the truth of universal quantifier over the tuples of R_T on expression b, which means equality of join (R_T <AND> *B*) and R_T – (R_T <AND> *B*) = R_T.

Keys: relational model, NULL, missing information, relational algebra A

Обзор моделей данных объектно-ориентированных СУБД

А.М. Эльдарханов
033555@gmail.com

Аннотация. Объектно-ориентированные СУБД – одно из наиболее перспективных направлений развития современной теории баз данных, наряду с дедуктивными и темпоральными СУБД. Тем не менее, серьёзным препятствием к построению теоретических основ ООСУБД и внедрению действующих ООСУБД является большая разрозненность подходов и отсутствие единого стандарта как в области теории (исчисление объектов, концепции моделей данных), так и в области практики (язык запросов, API для ОО-языков…). Целью данной статьи является анализ существующих на сегодняшний день концепций формального устройства объектно-ориентированных СУБД, начиная с моделей данных и далее переходя к формальным математическим моделям (исчислениям объектов, формализациям объектных языков запросов). В завершение делается заключение о наиболее актуальных проблемах моделирования ООСУБД.

Ключевые слова: модель данных, объектно-ориентированные СУБД

1. Введение. Историческая справка.

Современная концепция объектно-ориентированных СУБД возникла как результат развития двух поколений СУБД.

Первое поколение, представленное иерархическими и сетевыми СУБД 70-ых годов, ввело такие фундаментальные элементы СУБД, как схема базы данных, язык определения данных, язык запросов, выборка (select), кортеж и другие. Модель данных в этих системах являлась деревом (иерархические СУБД) или произвольным орграфом (сетевые СУБД). Вершина графа означала семейство однотипных кортежей (аналог таблицы в реляционной БД); кортежи, как ясно из вида модели, были связаны отношением родитель-потомок, никаких ограничений на связи не налагалось – эта задача ложилась непосредственно на программиста. Важным значением этих моделей было то, что они заложили принцип декомпозиции реальных данных на связанные между собой однотипные кортежи, который с тех пор по сути применяется практически во всех СУБД вне зависимости от их класса. Тем не менее, реализации СУБД с иерархической и сетевой моделью обладали значительным недостатком: отсутствием декларативного языка запроса к данным. Пользователи этих

СУБД были вынуждены пошагово программировать процесс того, как из хранилища будут извлечены данные, вместо того чтобы просто указать в запросе декларативным образом, какие данные нужны приложению в данный момент. Революция произошла с появлением реляционных СУБД, в которых императивные языки создания запросов были заменены декларативным языком SQL. Это стало возможно благодаря появлению алгебры Кодда, формализующей соотношения между кортежами и, в частности, предоставившей концепцию нормальной формы БД. Всё это обеспечило реляционным СУБД настолько высокую эффективность создания приложений, что РСУБД по сей день сохранили безусловное лидерство на рынке СУБД.

Тем не менее, уже в середине 80-ых годов в области систем управления данными возник ряд задач, для которых и реляционная модель была недостаточно эффективна. Круг их был достаточно широк, так что в СУБД сформировалось несколько направлений, значительно различающихся целевыми проблемами и подходом к модели БД – объектно-ориентированные СУБД, дедуктивные (логические) СУБД, активные СУБД, темпоральные СУБД. Все они в совокупности образовали **третье поколение** СУБД.

Класс СУБД третьего поколения, о котором идёт речь в этой статье, – **объектно-ориентированные СУБД** – возник как реакция на неоптимальность (impedance mismatch) реляционных СУБД для применения в таких областях, как автоматизированное проектирование (computer aided design, CAD); автоматизированное производство (computer aided manufacturing, CAM); автоматизированные системы управления предприятием (ERP); системы, основанные на знаниях (knowledge-based systems), мультимедийные системы. Проблемы РСУБД, выявленные в этих областях, составляют (приблизительно) следующий список:

– Ограниченность набора простых типов и отношений между сущностями. (Последнее фактически представлено всего одним экземпляром – связь через ключ.) При наличии в предметной области сложных специфических типов данных и стандартных операций над ними их приходится эмулировать созданием составных структур над разрешёнными простыми типами данных/одномерными таблицами и написанием сложных триггеров на таблицах с этими структурами, что является прежде всего крайне громоздкой задачей.

– Отсутствие object persistence, т.е. возможности постоянного хранения в БД составного объекта как единого целого. В реляционных СУБД составные сущности не хранятся как отдельный объект, а собираются в run-time с помощью операции JOIN из таблиц. В том числе это приводит к необходимости дополнительных вычислительных затрат на JOIN; или – если с БД работает некоторое клиентское приложение и изменяет программные объекты, агрегируемые из нескольких таблиц – к вычислительным затратам на разнесение изменений этого объекта на составляющие его таблицы.

– Отсутствие идентификации сущностей. В основе реляционных СУБД лежит ассоциативный доступ к данным, то есть доступ к записям на основе их значений (SELECT FROM… WHERE field=”value”), в отличие от адресного доступа, где обращение к элементу данных происходит по id, имени или адресу элемента. Это приводит к ряду проблем – к примеру, к несоответствию между механизмом доступа к данными с точки зрения SQL (ассоциативный механизм) и механизмом доступа к данным на носителе (адресный механизм, для которого РСУБД вынуждена выполнять преобразование в ассоциативный).

– Вычислительная неполнота. Чистый SQL, без процедурных расширений, не способен решить некоторые <разрешимые с теоретико-множественной точки зрения> задачи выборки данных из реляционной БД. Пример: пусть в БД хранится информация об отношении "родитель-ребёнок" среди некоторого множества людей. Для простоты, пусть люди задаются только своими id, а база данных задана 1 таблицей с 2 неключевыми полями: parent_id и child_id. Требуется вывести на печать всех потомков человека с данным id (т. н. задача вычисления транзитивного замыкания отношения родитель-ребёнок). Легко написать SQL-запрос, выводящий для данного id всех его потомков I поколения (детей), всех потомков II поколения (внуков) ит.д., но вывести потомков всех поколений (искомое множество) нельзя, т.к. заранее не известно нужное число шагов – JOIN'ов таблицы с собой. Даже отвлекаясь от SQL, в реляционной алгебре задача построения такого объекта, как замыкание отношения, алгоритмически неразрешима. В связи с этой проблемой в некоторые РСУБД вводят дополнительные средства, обеспечивающие работу с иерархиями (рекурсивные временные таблицы Common Table Expressions, рекурсивный оператор CONNECT BY). Их принцип действия основан на разрешении в ходе запроса создавать именованные промежуточные выборки данных с возможностью JOIN-а их с собой – что приводит к возможности вычисления результата такого запроса, если он выполняется на иерархической структуре данных.

– Непрозрачность интерфейса реляционной БД для приложений на ОО-языках. Множества примитивов модели данных ОО-языков программирования (объекты и литералы) и реляционной СУБД (поля, кортежи, таблицы) пересекаются лишь по представителям простых типов данных, таким как целым числам или символьным строкам, в остальном они не совпадают. Помимо данных, сильно различаются также функциональные части.

Концепция объектно-ориентированной СУБД возникла как реализация принципов объектно-ориентированного программирования в теории баз данных и позиционировала себя как средство разрешения многих указанных выше задач. В частности, она позволила разработчикам гибко реализовывать всю специфику предметной области через механизм классов, предоставляя программисту возможность в том числе задавать собственные составные типы данных (точнее, классы, т. к. класс и тип – не во всех ООСУБД одно и то же)

вместе с неотделимыми от них базовыми операциями (методами классов), как и в "обычном" ООП.

Теоретические основания ООСУБД с самого начала развивались практически независимыми группами разработчиков, в основном как математические модели и модели данных под конкретные ООСУБД. Поначалу основными представителями таких ООСУБД являлись исследовательские проекты, такие как IRIS (by Hewlett-Packard) [1] или ORION (by Microelectronics and Computer Technology Corporation) [2]; далее появились и коммерческие продукты – Gemstone [3], Jasmine [4], O2 [5]. Негативным следствием ситуации стало отсутствие единого стандарта на объектную модель данных, объектный язык запросов и т.д., что актуально и на сегодняшний день; впрочем, реляционные СУБД также проходили через эту стадию.

Некими попытками приведения ООСУБД к стандартам стали такие публикации, как "Манифест ООСУБД" [6], "Манифест систем баз данных третьего поколения" [7], "Третий манифест" [8], стандарты ODMG 1.0-3.0 (консорциум ODMG, 1991-2001, [9]) , о которых ещё пойдёт речь далее в статье. Несмотря на то, что некоторые из этих стандартов возымели успех (к примеру, ODMG 3.0 [10] был формально принят как стандарт многими вендорами, входящими в ODMG), единства всё равно не возникло. Следует отметить, что одним из основных спорных моментов в положениях этих стандартов является возможность выразить объектную модель реляционными средствами: позиции стандартов варьируются от отказа от реляционной модели (Первый манифест) до противоположной позиции, что объектная модель целиком и полностью реализуема в ОРСУБД (Третий манифест). Так или иначе, со временем происходило сближение объектной и реляционной концепций: с одной стороны, в стандарт языка SQL 1999 года [11] – SQL-3 – были включены объектные средства, позже реализованные в том числе в таких крупных представителях рынка СУБД, как Oracle и Postgres; а с другой стороны, поздние стандарты со стороны ООСУБД-вендоров, тот же ODMG 3.0, включали в том числе стандарты на объектно-реляционную привязку (mapping).

Важным этапом развития ООСУБД стала разработка к середине 2000-х гг. альтернативы механизму API ООСУБД для взаимодействия с приложениями, а именно включение средств конструирования запросов непосредственно в данный язык программирования. Это выразилось в создании таких средств, как Native Queries (для Java и C#) [12] и Microsoft LINQ [13] (для .NET).

Также в середине 2000-х гг. наблюдался рост популярности ООСУБД в связи с появлением и доступностью open-source ООСУБД, например db4o [14].

В 2006 году консорциум OMG объявил о возможности выпуска четвёртой редакции стандарта ODMG [15].

2. Общее устройство ООСУБД.

Задачу создания любой отдельно взятой ООСУБД можно рассматривать с точки зрения следующих уровней представления данных. (Деление достаточно условное, т. к. при желании каждый из уровней можно подразделить на более подробные.) Уровни указаны в порядке убывания абстракции. Первым двум из них далее посвящено по отдельной главе статьи.

Уровень формальной математической модели. На нём необходимо определить данные как формулы некоторой формальной теории, известной в математике или сконструированной самостоятельно. Вообще говоря, для построения модели ООСУБД достаточно лишь аппарата построения формул (являющегося лишь частью соответствующей ему формальной теории, наряду с аксиоматикой и правилами вывода формул), но для логического подхода к созданию ООСУБД характерно полное применение атрибутов формальной теории, в том числе логического вывода. К задаче формализации данных тесно примыкает задача формализации декларативного языка запросов: в первой задаче в виде формул строятся сами структуры данных, во второй задаче – пути обхода этих структур.

В будущем необходимо также определить, каким сущностям следующего уровня представления – **уровня модели данных**, см. следующий пункт – эти объекты соответствуют, иначе мат.модель будет изолированной "вещью в себе". Но пока формально на уровне мат.модели лишь нужно определить некую (алгебраическую?) структуру, не задумываясь о её интерпретации.

К примеру, реляционная алгебра имеет основой формальную Канторовскую теорию множеств, дополненную рядом операций над одним из её элементов – бинарными отношениями. Изначально определяются такие объекты, как множество и его элемент, затем декартово произведение множеств, затем некий предикат на этом произведении (бинарное отношение), и т. д. Интерпретациями этих объектов в реляционной модели данных являются, соответственно, поле <у таблицы> и его домен, затем кортеж и т.д.

Достоинство математической модели в её формальности – строго определены разрешённые объекты и действия над ними.

Уровень модели данных. СУБД на уровне модели данных – это также формально-математическая модель, но в качестве её элементов берутся не математические примитивы, а примитивы некоторого языка манипулирования данными. К примеру, в случае реляционной модели это тип данных, кортеж, таблица, ключ, индекс, связь по ключу, запрос. В объектном случае, примитивами выступают класс, объект, опять тип данных, указатель, uid и прочее, а также всевозможные производные от них; соответственно, требуется определить допустимые значения примитивов и в каких соотношениях могут состоять эти примитивы – система типов, вид типизации, как соотносятся типы и классы, как устроено наследование, ит.д. На этом же уровне задаются методы классов, синтаксис языка запросов, язык управления данными. Это

основной уровень разработки СУБД, большая часть статьи посвящена ему. Следует отличать разработку СУБД от разработки конкретной БД под неё; первая задача неизмеримо сложнее, т. к. вторая состоит лишь в создании конкретной реализации модели данных – т.е. схемы БД – и наполнения её данными и средствами их контроля, пользуясь уже созданным инструментарием СУБД.

Физический уровень. На этом уровне задаётся прямое взаимодействие СУБД с ЭВМ – действия с оперативной памятью, физическим хранилищем данных, процессорными инструкциями, а также всевозможные обслуживающие элементы СУБД, не относящиеся напрямую к моделированию данных – контроль совместного доступа к данным, система бэкапов, индексирование. В число задач этого этапа входит разработка механизма выполнения языка запросов (в отличие от предыдущего уровня, где задавался лишь его синтаксис), способа распределения данных по физическим хранилищам, системы кэширования запросов.

Стоит указать базовые для многих ООСУБД свойства, не относя их к какому-либо из уровней представления:

– Использование объектов. Объект – базовый элемент модели данных ООСУБД, позволяющий применять объектно-ориентированный подход при программировании этой СУБД. Он характеризуется набором своих атрибутов и методов. В отличие от реляционной модели, где в основном ведётся работа с кортежами атрибутов, атрибуты объектов принято скрывать от внешнего доступа, реализуя доступ к ним через методы. Если все поля являются скрытыми (т.е. открытыми только для методов класса), то говорят о "чистой объектной системе" [16], или, что то же самое, что объекты в такой системе являются скалярами.

Наличие объектов позволяет не только применять ООП, но и делает ООСУБД более прозрачной при доступе к ней из приложений на ОО-языках, т.к. объекты в таком случае являются примитивами как языка, так и БД, упрощая задачу создания API.

– Использование идентификаторов объектов (oid) и построение с их помощью составных объектов. Последнее означает, что атрибут объекта может быть oid-ом, то есть один объект может быть структурным подобъектом другого (или, в терминах классов, класс может иметь атрибут, принадлежащий не к простому типу данных, а к другому классу):

```
class TDept:{          // Класс "отдел компании"
char* name;            // Название отдела
TEmployee chief;       // Начальник отдела
…}
```

Такой подход, несмотря на свои преимущества (гибкость, компактность данных, простота представления иерархических структур), имеет и свои

недостатки, такие как, например, однонаправленность ссылок по oid-у: из объекта класса TDept можно непосредственно адресовать его начальника, но не наоборот. Плюс, появляется необходимость сборки мусора – удаления объектов, на которые не осталось ссылок.

– Возможность перегрузки методов, то есть введения нескольких одноимённых методов с разными наборами входных параметров. Это ведёт к невозможности определения типа входных параметров в некоторых точках вызова метода на этапе компиляции. Возникшая проблема разрешается, как и в ОО-языках, введением возможности позднего связывания, т.е. определения типа входных параметров на этапе выполнения программы (в данном случае в момент вызова метода в объектном запросе к БД).

– Сохраняемость объектов (object persistence). Определение этого свойства дано выше. Для реализации сохраняемости в ООСУБД применяется следующий приём: в пуле объектов выделяется некоторое множество корневых (root) объектов, которые хранятся как единое целое (персистентны) всегда. Затем любой составной объект, в структурном графе которого есть корневые объекты, сам объявляется персистентным. Задача удаления объектов, к которым не осталось ни одного структурного пути от корневых объектов, решается, как и в случае со ссылками по oid, сборщиком мусора.

3. Уровень модели данных ООСУБД.

Представление ООБД на уровне модели данных подразделяется на **структурную** и **поведенческую** части. Первая представляет данные и их домены, вторая – функциональную составляющую, то есть функции, методы и исключения.

Структурная часть, в свою очередь, подразделяется на **уровень данных** (data level) и **уровень схемы** (schema level). Первый представляет собственно данные и соотношения между ними, второй – "искусственные" сущности, непосредственно в данные не заложенные, такие как классы, типы, роли и отношения.

Уровень данных

Наиболее универсальное описание модели данных ООБД приводится в статье [2]. В этой статье объекты считаются атомарными элементами БД и им не даётся никакого внутреннего определения.

(На первый взгляд, это отличает данную модель от модели, принятой в большинстве ОО-языков программирования, где объект содержит атрибуты и методы. Тем не менее, забегая вперёд, скажем, что на самом деле модель Беери реализует ту же самую идеологию, лишь выражая её в терминах теории графов – объекты есть вершины некоторого орграфа, а его рёбра есть отношения наподобие "быть атрибутом".)

Каждый объект характеризуется своим уникальным идентификатором (oid, object identifier). Это важнейший примитив любой ООСУБД, аналогичный понятию ключа в реляционной СУБД.

Далее, если мы хотим ввести в ООСУБД какие-либо простые типы данных и экземпляры этих типов, мы можем выделить некоторые объекты из пула объектов и поставить каждому из них в соответствие целое число, символьную строку или любое другое значение-представитель простого типа. Такие объекты назовём *объектами-значениями*[1], остальные – *абстрактными объектами*. У объектов-значений отпадает необходимость иметь oid, так как вместо этого такие объекты можно заменить на сущности, поставленные им в соответствие: ясно, что, к примеру, число "7" или строка "abcd" однозначно и полностью будут определять соответствующий им объект.

Далее определим над объектами составные операции, такие как set, array, и tuple. (Набор составных операций для различных ООСУБД может варьироваться). Любая составная операция ставит в соответствие списку объектов ещё один объект.

array – создаёт массив однотипных объектов. Отношение однотипности задаётся снаружи операции array с помощью объявления принадлежности объектов к типам и классам, см. ниже.

set – создаёт множество однотипных объектов. То же, что и array, но порядок элементов отсутствует. Альтернативой является операция *isa*, задающая, наоборот, отношение принадлежности или инстанцирования. Традиционно в языках определения данных обозначается фигурными скобками {}.

tuple – создаёт кортеж <возможно, разнородных> объектов. Традиционно в языках определения данных обозначается квадратными скобками [].

В зависимости от конкретной ООСУБД, можно задать и другие операции над объектами, например:

state – ставит в соответствие объекту другой объект, понимаемый как его *состояние*. К примеру, состоянием объекта oid1, адресующего работника предприятия, может служить объект oid2, являющийся результатом применения операции tuple к тройке ("Smith", "$1000", oid3). Необходимость этой операции является предметом дискуссий (дилемма "attributes vs tuple-based states"); среди её преимуществ – возможность явно разрешить важную проблему отделения задачи сравнения объектов от задачи сравнения состояний объектов, среди недостатков – возможная избыточность.

После этих соглашений базу данных можно представить в виде графа, вершинами которого являются объекты, а рёбрами – отношения вида "быть i-ым операндом операции с данным результатом":

1 В ODMG 3.0 используется термин "литерал".

oid1 и oid2 – идентификаторы 2 студентов, студент задаётся кортежем из имени и года рождения. Набор (set) этих 2 студентов составляет объект oid3. Объект oid4 – вуз, являющийся кортежем из объекта-названия и объекта-множества своих студентов. Номер после "tuple" означает, какой это по счёту операнд операции составления кортежа.

Уровень схемы

На уровне схемы, как уже сказано, задаются **типы**, **классы**, **отношения** и **роли**, если они есть. Всех их предлагается считать опять же неопределяемыми "атомами" схемы данных, то есть задаваемыми исключительно через их соотношения друг с другом и с объектами предыдущего уровня.

Как и на предыдущем уровне, схема данных понимается как некоторый (возможно несвязный) орграф, вершины которого являются указанными выше атомами, а рёбра означают операции создания составных объектов (tuple, set, array...) и прочие специфические операции, к примеру:

state – операция сопоставления состояния, аналогично *state* на уровне данных. А именно, если ООСУБД поддерживает состояния, то вершины-типы в ней мы разбиваем на две разновидности: структурные типы (объекты которых служат как состояния) и абстрактные типы. Далее стрелка *state* соединяет абстрактный тип с его типом-состоянием.

inherits (isa) – операция наследования типов или классов, о ней идёт речь ниже.

Следует заметить, что граф схемы отличается от графа объектов по использованию операций создания однородных наборов (set,array). В графе объектов все вершины-наборы создаются явно указанием всех стрелок *set* или *array* от вершины-набора к вершинам-элементам. В графе схемы же ставится единственная стрелка *set* или *array* от вершины-типа к вершине, означающей однородный набор элементов этого типа.

```
type TStudent = [name: string, birth_year: integer]
type TStudents={student:TStudent}
type TInst=[name: string, students: TStudents].
```

Диаграмма типов для БД студентов и вузов, упомянутой выше, и её задание на гипотетическом DDL

Типы и классы: Существенной чертой ООСУБД является выделение разницы между типами и классами. В терминах графа схемы разница следующая: вначале в пуле объектов выделяются все однородные наборы,

состав которых задан заранее (к примеру, integer, или перечислимый тип, или множество структурно созданных объектов с одной и той же структурой). В графе схемы на каждый такой набор создаётся вершина, называемая **типом**, затем такие вершины-типы соединяются стрелками структурных операторов, если какие-то из этих типов структурно образованы из других. В противоположность этому, **классом** называют вершину, обозначающую абстрактный, заранее не заданный однородный набор объектов данного типа, что реализуется введением ещё одной особой разновидности стрелки, задающей принадлежность класса к типу.

Иными словами, для вершины-типа заранее можно указать её расширение (список объектов данного типа), а для вершины-класса нельзя.

Соотношение между примитивами тип и класс является достаточно спорным вопросом в различных ООСУБД. Существуют как ООСУБД, в схемах которых используется только один из этих примитивов (к примеру, в системах, написанных на CLOS, часто есть только классы и нет типов), так и ООСУБД, поддерживающие оба примитива, тем более что последнего требует ODMG.

Отношения (есть не во всех моделях ООСУБД): Отношение с точки зрения математической логики – это функция, отображающая некоторое множество однотипных объектов на множество {false,true}.

В ООСУБД отношения задаются классами. Класс определяет некоторое множество однотипных объектов. Все объекты, которые входят в данный класс (потенциально могут существовать другие объекты этого же типа с другими значениями атрибутов, не входящие в класс), формируют отношение так же, как формируют отношения кортежи, находящиеся в таблицах РСУБД. Таким образом, в ООСУБД класс задает некоторое отношение так же, как в РСУБД отношение задается таблицей.

Распространённый пример отношения – бинарное отношение, задаваемое на декартовом произведении 2 множеств и означающее, что если для данной пары объектов оно истинно, то эти 2 объекта состоят в некотором "физическом" отношении, например отношении родства между объектами-людьми, а если ложно – то не состоят. Тем самым для задания отношения достаточно на области определения отношения задать подмножество, на котором и только на котором функция истинна.

В ООСУБД для задания отношения между двумя объектами могут применяться ссылки по oid. Если объект o_employee значением своего атрибута works_in_dept имеет ссылку на объект o_dept, то можно говорить что этот объект служащего o_employee находится в отношении works_in_dept с объектом отдела o_dept, т.е. служащий o_employee работает в отделе o_dept. При этом нужно обратить внимание на то, что ссылки в ООСУБД однонаправлены и в отличие от РСУБД нельзя говорить о том что отдел o_dept находится в симметричном отношении "содержит_работника" к служащему o_employee. Это связанно с техническими сложностями

реализации в ООСУБД поиска всех объектов, ссылающиеся на некоторый заданный объект. При необходимости такой функционал может быть реализован аналогично тому, как реализован поиск по индексу в современных РСУБД. В качестве примера такого расширения трактовки ссылок можно привести механизм запросов интегрированных в язык программирования (LINQ) разработки компании Microsoft. Этот механизм позволяет найти всех работников заданного отдела:

```
from emp in employees where emp.works_in_dept ==
o_dept select emp;
```

В графе схемы ООСУБД отношения задаются так же, как и классы, то есть введением вершин-отношений и стрелок *rel*, указывающих на вершину-тип.

```
RInversion i1 = TRational2.Create ((2,5),(5,2));
RInversion i2 = TRational2.Create ((1,3),(3,1));
```

Отношение обратности двух рациональных чисел и добавление в него пар (2/5, 5/2) и (1/3, 3), в предположении наличия C++-подобного языка определения данных и класса TRational

Тем самым введение отношений выглядит избыточным, дублирующим функциональность классов, но при введении стандартных операций над отношениями позволяет фактически создавать в рассматриваемом "графовом фреймворке" реляционные модели данных.

Роли (есть не во всех моделях ООСУБД): Основная задача ролей – реализовать возможность динамического наследования.

Рассмотрим распространённую проблему [18]: пусть в ООБД есть классы "работник" и "человек" – cEmployee и cPerson, и в некоторый момент человек, занесённый в БД как объект класса cPerson, был взят на работу, т.е. возникла необходимость <временно> придать ему поля и методы класса cEmployee, а в будущем, возможно, лишить его присвоенных атрибутов. Это может быть реализовано созданием нового класса, унаследованного от двух названных (если имеется возможность множественного наследования и в классах нет одинаковых полей наподобие "имя"), либо созданием cEmployee как унаследованного от cPerson. Данные способы представляются неудобными: первый из них нарушает принцип замещения Лисков ("представитель типа должен безопасно заменяться на представителя унаследованного типа"), т.к. классы cEmployee и cPerson лишаются естественного отношения вложенности, второй же приводит к необходимости удалить прежний объект и пересоздать его как объект нового класса.

Указанная проблема называется задачей реализации динамического наследования и может быть разрешена следующим образом. Среди объектов-

представителей классов выделим некоторые и назовём их **ролями**. Далее введём в граф объектов особый тип стрелки *has_role* между обычным объектом и объектом-ролью. Смысл такой конструкции – обычному объекту присваивается временная принадлежность к объектам данного класса, возможность "сыграть роль" объекта данного класса.

Возвращаясь к задаче о cPerson и cEmployee, роль можно назначить так:

Далее при необходимости для данного объекта присваиваются или отнимаются роли.

Поведенческий аспект.

Помимо структурной части, хранящей собственно данные, модель всякой ООБД также обладает т.н. поведенческой частью, у которой нет аналога в реляционной модели. Основными элементами поведенческой части являются: метаклассы, классовые атрибуты, операция наследования, методы и функции. [17]

Метаклассы. Для введения метаклассов предлагается несколько изменить структуру данных, переместив вершины-классы из графа схемы в граф объектов (сделав их абстрактными объектами). Для каждого класса, по определению класса выше, при этом необходимо ввести ещё одну вершину, обозначающую набор объектов данного класса, и соединить их стрелкой *set*. Это даёт возможность производить структурные операции над классами, работая с ними как с объектами. Для организации коллекций объектов-классов в граф схемы вводятся новые вершины, называемые **метаклассами**, с которыми объекты-классы можно связывать стрелкой *has_class*.

Классовые атрибуты. Объектные модели с метаклассами позволяют реализовать известные по ООП **классовые атрибуты**, то есть возможность наличия у класса атрибута, значение которого одно и то же для всех объектов данного класса. Более точно, так как классы теперь являются объектами, классовый атрибут есть не что иное, как вершина-объект (обычный), соединённая стрелкой структурной операции с вершиной-классом.

Наследование. Выше указывалось, что существует стрелка *inherits* между типами или классами, обозначающая наследование. В данной объектной модели **наследованием типов** называют операцию создания одного типа из другого с помощью структурного добавления вершин. Это объясняет отождествление в [17] операций *inherits* и *isa*, так как такой подход есть реализация вложенности типов: представители типа-потомка гарантированно имеют в своём составе те же поля, что и представители типа-предка (и, может быть, ещё какие-то), так что могут использоваться вместо представителей типа-предка согласно принципу замещения Лисков.

```
TEmployee=[name:string, salary:integer]
```

```
TEmployeeChief  inherits TEmployee add dept:TDept
```

Наследование классов есть наследование соответствующих им типов плюс требование унаследовать классовые атрибуты на графе объектов (если модель с метаклассами и классовыми атрибутами), а также унаследовать методы родительского класса.

Методы и функции. Это фундаментальные составляющие ООСУБД, неясным может вначале показаться их место в графе объектов и графе схемы, так как они не являются данными или доменами. Существует подход, при котором они считаются не входящими в объектную модель и внешним образом манипулируют данными. Такая практика используется в ООСУБД, в которых для модификации данных используется непосредственно язык программирования БД, а не язык управления данными, например Caché [18], db4o (хотя они ограниченно поддерживают также управление данными через SQL). Этот подход гораздо проще прочих, но он никак не принимает во внимание целостность данных и контроль целостности.

Существует возможность включить методы и функции в объектную модель, для чего применяется тот факт, что любая математическая функция представима на теоретико-множественном языке как выделенное множество пар вида [входная переменная, выходная переменная]. Иными словами, функция есть некое отношение, а отношения входят в объектную модель. Следует заметить, что представление в виде отношения для функций нескольких переменных неоднозначно: например, функцию, возвращающую для данных двух родителей множество их детей, можно представить как минимум 2 способами.

1. Тернарное отношение: (o_person, o_person, {o_person}).
2. Вложение бинарных отношений: (o_person, (o_person, {o_person})).

Для полного включения функций в объектную модель следует добавить в неё структурные операции над функциями, хорошо известные из функционального программирования: композиция, apply-to-all и другие (см. ниже о функциональном подходе в математической модели).

Остаётся вопрос, как с помощью такого подхода реализовано включение методов классов (а не абстрактных функций) в граф ООСУБД, но это скорее технический вопрос, решение которого схоже с решением задачи включения отношений, метаклассов ит.д. в объектную модель и поэтому не будет здесь приводиться.

4. Уровень математической модели ООСУБД.

Как уже сказано выше, задача формализации данных ООСУБД схожа с задачей формализации декларативного языка определения данных: обе они предполагают построение объектов или запросов к ним в виде формул

некоторой формальной теории. Существует два подхода к созданию такой формальной теории – **функциональный** и **логический (дедуктивный)**.

При функциональном подходе из всей формальной теории задаются только структурные правила создания формул и, быть может, оценки их истинности, то есть семантика теории. Логический вывод не применяется. Подход получил своё название от функционального программирования, основой которого является построение результата работы программы как результата последовательного вычисления некоторой математической функции, что схоже с последовательным построением формул.

В [17] выделяется два основных функциональных подхода: алгебра **select-project-join** (SPJ) и многосортное исчисление предикатов высшего порядка (называемое у него **полным исчислением объектов**).

Основой функционального подхода является задание объектов, базовых функций над ними и структурных операций над функциями. В алгебре SPJ (идеологию которой использует SQL) объектами являются: атомы объектной модели, наборы однородных элементов (set, array), кортежи. Базовыми функциями являются:

1. разнообразные элементарные операции над числами и строками (сумма, конкатенация итд);
2. σ (pred)(set) – предикат pred на множестве set (aka выборка);
3. p (fun)(set) – применение одноместной функции fun к каждому элементу множества set (aka apply-to-all из ФП).

Структурные операции – композиция, кортеж, агрегация (на вход получает однородный набор и бинарную функцию f, например сложение, на выходе выдаёт результат $f(x_1,f(x_2,f(x_3...)...)))$, операция if-then-else и т. д.

Приведём несколько примеров правильно построенных запросов в алгебре SPJ.

Каждый из них допускает 2 формы записи: выражение на некотором SQL-подобном языке и выражение на языке лямбда-исчисления. Кроме того, добавим ещё, забегая вперёд, запись в виде формулы на языке полного исчисления, чтобы провести сравнение. Очевидно, третья запись не есть альтернативная форма записи двух предыдущих, т. к. полное исчисление устроено иначе, чем SPJ-алгебра.

Пусть база данных задана так (пример из [17], но без состояний):

```
o_type o_person = [name:string, b-date:date]
o_type o_empl inherits o_person add [sal:int,
children:{c_persons}]
o_type o_dept=[dname:string, budget:int, mgr:c_empls,
employees:{c_empls}]
```

```
class c_persons: o_person
class c_empls: o_empl
class c_depts: o_dept
…
//Заполнение базы
dept1: o_dept = (какой-то вызов конструктора)
dept2: o_dept = (какой-то вызов конструктора)
…
depts: {dept1,dept2,...}
```

Запрос 1: Найти всех начальников отделов, бюджет которых <отделов> более 1000.

```
SQL:            select   mgr   from   depts   where
budget>1000
Лямбда-исчисление:    p (mgr)(σ (budget>1000)(depts))
Формула:              d.mgr: d∈ depts & d.budget>1000
```

Запрос 2: Для каждого отдела, вывести его название, имена сотрудников с >3 детьми, и имена их детей.

```
SQL:            select  dname,  (select  name,  (select
name from children)
      from employees
      where count(children)>3)
Лямбда-исчисление:    p(dname, p(name,
p(name)(children))
      (σ (aggr(+,p(f_1,children))>3)(employees)))(depts)
Формула:  e.dname, {d.name, {c.name:    c∈d.children}:
      count(d.children)>3}:d∈e.employees
```

(где f_1 – базовая функция, принимающая на вход string и на выходе выдающая тождественную 1; допустим, она объявлена выше)

Видно, что полное исчисление объектов отличается от алгебры SPJ в первую очередь тем, что оно допускает операцию принадлежности к типам, то есть является многосортным, а также что оно является исчислением предикатов высшего (выше первого) порядка, так как допускает квантификаторы не только над атомарными объектами. Это достаточно неудобная черта, так как известно, что теории высшего порядка не аксиоматизируемы (не существует набора формул такого, что все остальные правильно построенные формулы из них выводятся), что в свою очередь ведёт к большой вычислительной сложности задачи проверки истинности формул (считаем, что задана семантика формул, то есть отображение множества формул на {true,false}), в то время как в случае аксиоматизируемой разрешимой теории для проверки истинности формулы достаточно было бы проверить её на разрешимость. Для сравнения, реляционная модель основана на исчислении предикатов первого порядка, допускающем аксиоматизацию.

Дедуктивный подход. Этот подход представляет целое направление в современной теории баз данных, причём, что характерно, объектно-ориентированные системы применяют некоторые дедуктивные механизмы – к примеру, в создании языков запросов (т. н. языки, основанные на правилах) или в задачах контроля целостности.

Общая идея этого подхода – представление запросов СУБД в виде формул некоторой формальной теории, получаемых с помощью логического вывода в этой теории. Логический вывод над множеством формул представляет из себя заранее выделенный набор формул, называемых аксиомами, и правил (схем) вывода, позволяющим ставить в соответствие набору формул ещё одну формулу, называемую синтаксическим следствием из данных формул. За аксиомы обычно берутся базовые предикаты принадлежности литералов к простым типам и заранее определённые отношения. Пример правила [16]:

MOTHER_OF (X,Y), MOTHER_OF (Y,Z) → GRANDMOTHER_OF(X,Z)

Здесь MOTHER_OF и GRANDMOTHER_OF – имена предикатов, означающих соответствующие формы родства.

Ясно, что если в логической БД будет задана принадлежность некоторых людей к предикатам MOTHER_OF и GRANDMOTHER_OF в виде аксиом, то база будет иметь возможность обрабатывать запросы вида, к примеру, SELECT * FROM persons p WHERE GRANDMOTHER_OF(p,p0), вычисляя истинность предикатов для разных p с помощью или аксиом непосредственной принадлежности человека к GRANDMOTHER_OF(·,p0), или данного правила вывода.

Значительным плюсом такого подхода является поддержка рекурсивных запросов. Часто приводимый пример – запись транзитивного замыкания.

Пусть есть граф и на множестве его вершин есть отношение child (вторая вершина есть прямой потомок первой), требуется определить его транзитивное замыкание, то есть отношение path_exists (вторая вершина достижима из первой). Для этого можно использовать пару дедуктивных аксиом:

child(x,y) → path_exists (x,y)
path_exists(x,z), child (z,y) → path_exists (x,y).

В такой аксиоматике формула path_exists (x,y) будет разрешима тогда и только тогда, когда у достижим из x, что и требовалось от БД.

Кроме того, представление БД в дедуктивной форме позволяет единообразно, в виде аксиом, задавать ограничения целостности (например, в виде предикатов принадлежности)

Тем не менее, в применении логического подхода к моделированию ООСУБД есть несколько трудностей. Одна из основных проблем связана с наличием в ООСУБД, возможно, неразрешимой рекурсии в правиле, к примеру если при

вычислении тела правила необходимо перебрать элементы отношения, модифицируемого в выводе правила, например:

$$p_1(X,Y) \rightarrow p_2(X,\{Y\})$$

Это правило реализует операцию group by по первому аргументу: для каждого фиксированного объекта X берутся все объекты Y, которые удовлетворяют в паре с этим X отношению p_1, и объединяются в <зависящее от X> множество {Y}, после чего пара из этого X и соответствующего ему объекта-множества {Y} включается в новое отношение p_2, которое следует назвать отношением группировки. Но если положить $p_1=p_2$, то получается, что для обхода множества пар {X,Y} в том числе необходимо знать, входит ли создаваемая пара в отношение, а это неизвестно, т.к. правило ещё не сработало.

Методом борьбы с этой проблемой является введение введение т. н. ограничений стратификации на правила, запрещающих предикату в выводе правила зависеть от себя самого в теле правила.

5. Выводы и основные проблемы.

Представленная структурная модель данных является представлением на языке теории графов целого класса структурных объектных моделей и достаточно точно реализует как основные принципы ООП (oids, структурные объекты, персистентность объектов, инкапсуляцию, наследование, полиморфизм), так и абстракции группировки данных – абстрактные множества, типы, классы. Модель состоит из двух уровней, структурного и поведенческого, первый из которых есть усложнённая версия реляционной модели данных, второй представляет агрегирующие объекты – типы, классы, отношения, функции. Право агрегирующих объектов в данной модели участвовать в тех же операциях, что и индивидуальные объекты, даёт в качестве математической формализации этой структурной модели достаточно выразительную систему – полное объектное исчисление, являющееся разновидностью логики второго порядка, но одновременно это право (наряду с разнообразием базовых элементов ООСУБД и связей между ними) приводит к проблеме невозможности создать единообразное средство проверки корректности объекта теории (чем бы он ни был – формулой структурного типа, предикатом, правилом или чем-либо другим), выраженное на языке этой теории. Иными словами, можно сказать, что ограничения на корректность объектов теории являются пока на сегодняшний день внешними средствами по отношению к теории, в основном выраженными в форме указания конкретных классов некорректных объектов с обозначением пути разрешения проблемы.

Кроме указанной (кратко называемой в [17] "higher-orderness problem") важной является также проблема реализации в данной модели сочетания функционального и логического программирования. Первое представляет

аппарат для работы с функциями как с данными, второе – с отношениями как с данными (операции с отношениями также предоставляет, очевидно, реляционное исчисление; известно, что реляционная СУБД является частным случаем логической). Оба аппарата необходимы, т.к. хотя функции и представимы, как сказано выше, в виде отношений, но это представление неудобно. Полной формализации включения функций и отношений в объектную модель пока не создано.

Список литературы

[1] D. H. Fishman et al. Overview of the Iris DBMS. In Object-oriented concepts, databases, and applications, ACM Press. New York, NY, 1989, 219-250

[2] Won Kim, Jorge F. Garza, Nathaniel Ballou, Darrell Woelk. Architecture of the ORION Next-Generation Database System // IEEE Trans. Knowledge and Data Eng.- 2, N 1.- 1990.- 109-124

[3] Tomothy Andrews, Craig Harris. Combining Language and Database Advances in an Object-Oriented Development Environment GemStone Object-Oriented DBMS // Proc. OOPCLA'87, Orlando, Fla, USA, Oct. 4-8, 1987.- 430-440

[4] Hiroshi Ishikawa et al. An Object-Oriented Database System Jasmine: Implementation, Application, and Extension. IEEE Transactions on Knowledge and Data Engineering, Volume 8 Issue 2, April 1996, 285 - 304

[5] Christophe Lecluse, Philippe Richard, Fernando Velez. O2, an Object-Oriented Data Model // Proc. ACM SIGMOD Int. Conf. Manag. Data, Chicago, Ill, USA, June 1-3, 1988, ACM SIGMOD Record.- 17, N 3.- 1988.- 424-433

[6] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik: "The Object-Oriented Database System Manifesto", Proc. 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan (1989). New York, N.Y.: Elsevier Science (1990). (Имеется русский перевод: М. Аткинсон и др. "Манифест систем объектно-ориентированных баз данных", *СУБД,* No. 4, 1995, http://citforum.ru/database/classics/oo_manifesto/)

[7] M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, Ph. Bernstein, D. Beech. "Third-Generation Data Base System Manifesto". Proc. IFIP WG 2.6 Conf. on Object-Oriented Databases, July 1990, *ACM SIGMOD Record 19,* No. 3 (September 1990). (Имеется русский перевод: Стоунбрейкер М. и др. "Системы баз данных третьего поколения: Манифест", *СУБД,* No. 2, 1996, http://citforum.ru/database/classics/manifest/)

[8] Hugh Darwen and C. J. Date: *The Third Manifesto. ACM SIGMOD Record 24,* No. 1 (March 1995). (Имеется русский перевод: Х. Дарвин, К. Дейт. "Третий манифест", *СУБД,* No. 1, 1996, http://citforum.ru/database/classics/third_manifesto/)

[9] Стандарты ODMG. http://www.odbms.org/ODMG/OG/

[10] "The Object Data Standard: ODMG 3.0". Edited by R.G.G. Cattel, Douglas K. Barry. Morgan Kauffmann Publishers, 2000

[11] Jim Melton, Alan R. Symon. "SQL:1999. Understanding Relational Language Components". Morgan Kaufmann Publishers, 2002

[12] William R. Cook and Carl Rosenberger. Native Queries for Persistent Objects. February 01, 2006, http://drdobbs.com/windows/184406432

[13] Elisa Flasko. Introducing LINQ to Relational Data. Microsoft Data Platform Development Technical Article, January 2008, http://msdn.microsoft.com/en-us/library/cc161164.aspx

[14] db4o: http://www.db4o.com/
[15] 4th Generation Standard for Object Databases on its Way, NEWS 2/18/2006, http://www.odbms.org/About/News/20060218.aspx
[16] К. Дж. Дейт – Введение в системы баз данных, 8-е издание.: Пер. с англ. – М.: Издательский дом "Вильяме", 2005
[17] C.Beeri – A formal approach to object-oriented databases. – Data & Knowledge Engineering, vol 5 (1990), p. 353-382
[18] InterSystems Caché, http://www.intersystems.com/cache/

An Overview of Data Models of Object-Oriented DBMSs

A.M. Eldarkhanov
033555@gmail.com

Abstract: Object-oriented DBMS theory is one of the most prospective areas of modern database theory, along with deductive and temporal DBMS. Nevertheless, a high variety of different approaches and absence of a uniform standard, both in theoretical OODBMS area (object calculus, data models) and in practical area (query languages, database API for programming languages), seriously hinder creation of firm OODBMS basics. The purpose of this article is to analyze the existing concepts of OODBMS structure, observing object data models and formal mathematical models; finally, a number of primary actual problems of OODBMS theory is formulated.

Keywords: data model, object-oriented DBMSs

Snapshot Isolation Protocol Performance Evaluation

D. Vasilik
Saint Petersburg State University
Dmitri.Vasilik@gmail.com

Abstract. Snapshot Isolation (SI) level is extensively used in commercial database systems. We developed a simple SI implementation protocol for distributed DBMS and implemented it in the Apache HBase. This work presents the performance evaluation of the protocol in HBase cluster for the OLAP-like workload. We have measured the performance of a single-node system and validated the model using the obtained results.

Key Words: Snapshot Isolation; distributed systems; performance modelling; HBase.

1. Introduction

In the last ten years the special class of data-intensive web-application has emerged. Youtube, Google Maps, social networks, etc. represent the applications of the class. These applications can be characterized by the following features: they work with large volumes of data and typically do not need ACID transactions.

The choice of the concurrency control to employ in the application is of great importance, since it affects the performance of the application as well as the complexity and cost of development. For example, choice of eventual consistency (EC) as an isolation level may result in increased complexity of programming model. Thus, application designer has to carefully look for the most suitable compromise between level of guarantees he gets and performance of isolation level protocols.

The Snapshot Isolation (SI) isolation level was introduced in [2]. It is widely adopted by the commercial and open-source systems thanks to its ability to cope with read intensive workloads and high degree of consistency guarantees. It is even used instead of serializable isolation level in Oracle and PostgreSQL. This decision can be justified by level of guarantees SI actually offers.

The transaction executed under SI reads data from a snapshot of the data as of the time the transaction started. Two transactions are called competitive if they have overlapping execution intervals and they have written the same data item. When the transaction is ready to commit, it is assigned the commit timestamp and in case if

there is no competing transactions in the system it commits. This feature is called "First-Commiter-Wins".

We developed SI protocol for distributed DBMS. We have presented it in [12] along with its implementation prototype. We have chosen HBase to implement it in because it is open source distributed data storage which has natively supported only EC1. In [12] we have measured the performance of the prototype in a single-node mode and compared it with that of HBase, but we had not an opportunity to run the system in a distributed mode. Compared to [12], in this work we present performance model of the protocol and the comparison of the protocol simulation results with HBase simulation results for the distributed mode.

The remainder of this paper is organized as follows. In Section 2 we discuss literature works on the DBMS performance evaluation with concurrency control performance comparison being in focus. Section 3 contains the description of the protocol and its implementation. Performance model is provided in Section 4. Section 5 presents results of conducted simulation experiments. Section 6 contains discussion of protocol implementation performance along with model validation issues. In section 7 the results are summed up.

2. Related Work

A lot of concurrency control algorithms was introduced in the last 30 years. Considerable research was devoted to evaluating the performance of concurrency control algorithm in the 80s and the earlier 90s.

In [1] authors has examined the assumption set employed in concurrency control performance (CC) modelling. Authors has investigated the reasons of apparent contradictions in performance results of earlier studies. In particular, the common assumption of infinite resources was critically examined. The paper [11] lists the CC methods along with results gained from analytical and simulation studies. For a given CC method the models employed for its performance evaluation are briefly described. In [10] issues in modeling performance are discussed. The paper contains tens of questions to be solved by the researcher before he starts to implement his model along with some critique of using one's intuition in performance modelling.

Works [7, 6, 5] are focused on the distributed systems performance evaluation, however, this studies do not have the CC methods in the main focus. In [6] performance models classification based on the model assumption set is provided. The paper [7] contains a detailed comparison of Shared Disk and Shared Nothing (SN) architectures. In particular, the performance of Scan and Join operators is discussed. The paper [5] contains a comprehensive SN DBMS performance study, performance is modeled using three different workloads, one of them being generated from database traces.

[1] Optimistic concurrency control was the only available alternative for HBase by the moment we started implementation.

The work [8] presents an analytical SI performance model for a standalone server machine.

Recently there was a number of studies related to SI implementation in distributed systems [14, 4, 13]. The work [4] presents a technique to preserve the behaviour of centralized SI system in guaranteeing global SI in distributed, lazily replicated system. The technique leverages local SI concurrency control. A simulation model was developed to study the cost of using the technique. This model is rather simple in aspects of single-node performance, e.g. the service time per operation is specified as model parameter.

The article [13] presents a cloud storage system named ecStore. The article contains a brief description of concurrency control protocols used, but it does not contain CC performance comparison.

In the paper [14] an approach to use HBase as a cloud database solution with global SI is described. The paper presents an optimistic protocol, which uses 4 special tables and several queries to provide SI gaurentees. The protocol is implemented on top of bare-bones HBase on the client side, it's imlementation is rather high-level, it does not introduce any changes to server code. Authors have conducted several experiments in a distributed environment (3-machine cluster) to evaluate the cost of employing the proposed protocol.

In this work we present performance models for our protocol and HBase and the comparison of simulation results. To the best of our knowledge the performance model of Key Value store like HBase has not been presented yet.

3. Implementation Protocol

3.1. Protocol Description

Let us suppose that current transaction state, its start timestamp and end timestamp (if a transaction has been already committed or aborted) are available at the execution time.

Read operation execution is straightforward. To read a data object x the transaction t performs a selection of all versions of x, committed before start time of t. Then it traverses obtained list of versions to find the latest committed version. If the list contains version written by t, this version is read instead of the latest committed.

Before the value of x can be updated the transaction has to ensure, that no competitive transaction had already written x. If x was updated by transaction, which was committed after t had started or by active transaction, than t aborts.

We have formally proved the correctness of the protocol in [12], but the prove is not in the scope of this work.

3.2. Data Storage Design Issues

We have implemented our protocol in HBase. HBase is an open-source distributed data storage written in Java. It was made after Google Bigtable. In [3] Bigtable is presented along with design and implementation decisions made by developers.

Authors has called Bigtable "a sparse distributed persistent multidimensional sorted map". Bigtable API is very simple. HBase API is also restricted to get, put, delete and scan operations. The data processing necessary for a query can not be pushed as close as possible to data, as it is usually done in distributed RDBMS. If a client wants to update data according to a special rule, it requests the data, updates it locally and puts the updated data to the storage. That is, complex query execution is decomposed into two or more phases. As it was said before, initially only eventual consistency isolation guarantees were provided by HBase. HBase/Bigtable users who needed transactional execution, were suggested to implement it themselves using data timestamps.

HBase cluster consists of servers of two types: master nodes and region nodes. Every region server manages a number of regions. HBase uses horizontal partitioning, each region contains data for an interval of keys.

HBase is column oriented storage, that is, data is stored on a per column basis in main memory and in the file system. Every column store has its own set of files and a per file buffer pool employing the classical LRU discipline to manage pages. HBase use immutable files to store data. When the total space occupied by updates kept in main memory exceeds specified threshold a new file is written to the file system. This is done in a special thread. HBase uses write ahead log (WAL), that is, every update is written to log before it gets to main memory store.

3.3. Protocol Performance Discussion

The "First-Commiter-Wins" feature has an optimistic nature, thus SI definition naturally accepts the optimistic protocol. Our approach is not optimistic, it should be rather called "First-Updater-Wins".

Although the algorithm is nonlocking, it is more close to restart oriented locking methods such as immediate-restart. Immediate-restart algorithm description, its performance modelling and comparison with blocking and optimistic methods can be found in [1]. More sophisticated restart oriented methods exist such as wound-wait or wait-die, which could outperform the immediate-restart method in most cases ([11]). We did not attempt to build a protocol on top of these ideas.

Before a data object is written the version check is performed. All data object versions made after the transaction had started must be selected and traversed. It may take a considerable amount of time, because some of committed versions of data object may have been already written to the file system. In this case a number of page reads is to be performed.

Some tricks could be used to avoid unnecessary disk accesses or at least reduce their probability. The main memory store could be assigned a timestamp each time its

contents are flushed to a new file. The version check performed in main memory is sufficient to ensure "First-Writer-Wins" feature if the transaction started after the main memory store had been written to disk last time. The second trick which may be used is to write only the oldest versions to file, keeping new ones in memory. Given average transaction execution time t_{avg}, we can keep versions with timestamps greater than $now()-t_{avg}$.

To ensure transactional execution of queries in distributed data storage we used the distributed commit protocol. We have chosen a protocol similar to Two Phase Commit (2PC) for its implementation simplicity. The classical 2PC successfully commit scenario consists of the following steps. The transaction initiator waits for READY message from all the participated servers. When all the participants voted to commit, it sends them COMMIT message and waits for ACK messages. After all ACK messages were received, initiator finishes the transaction.

Since HBase uses WAL, when subtransaction sends READY message to the transaction home node (client machine), it has no additional work to do before it can commit. It can not be aborted by other local subtransaction. When the transaction initiator receives READY message from all the execution region servers, it just sends them COMMIT message and commits the transaction locally. It does not wait for ACK messages from executor servers as in the classical 2PC.

Let us summarize the overheads introduced by the protocol.

- Employing distributed commit protocol leads to additional communication overhead.
- Version check may lead to additional (local) disk accesses and may consume CPU.
- Certain resources are wasted due to additional aborts by protocol reasons

4. Simulation Performance Model

The simulation model was written in Java using Simkit simulation package [9]. Our model is rather hardware resource bounded than data contention bounded. We believe that HBase was not intended to be used for the workloads with high data contention. Our model was not aided to solve some fundamental questions, it concentrates on answering the engineering questions only. We have modeled the particular distributed data storage performance, so the model is much less universal than models employed in mentioned related works. However, the model is rather simple. It abstracts from some details. For example, the primary copy replication used in HBase, may affect the performance, but it is not reflected in the model.

The model will be described in terms of queries, we will use the term transaction when we need to outline the differences.

Parameters		Settings
CPU:	cores per PE processor capacity (MIPS)	2 9000
#avg instr.:	BOE and EOE steps per object reference for message for IO operation	5000 25000 5000 3000
Disks:	count per PE access time mean (ms) page transmission time (ms)	1 8 0.2
Network: Database:	throughput (Mbit/sec) min packet size page size (Kb) count of pages per PE count of pages in buffer pool	100 540 4 53000000 500000

Table 1: System parameter settings.

4.1. Modelling Assumptions

We use a close system model. There is always the same number of queries in the system. After one query had completed its execution, the new one immediately replaces it. Since the concurrency control performance hardly depends on the active query count in the system, we decided to keep these number constant.

The cluster is modelled as a queueing network, each node is itself modeled as a queueing network.

We use quite common assumption called “homogeneity assumption” in [6].

- All database sites have the same structure and the same service capacity.
- The amounts of data allocated to each unit are equal.
- Data accesses are spread among nodes in uniform manner.

We do not use a sophisticated model of a communication network as well, because we believe that network should not be a bottleneck. In our model there is a virtual connection between every query initiator (client) and executor node with 100Mbit/sec throughput.

Table 1 contains the system settings used for experiments.

Parameters	Settings
M N_r N_w N_{vc}	7 1000 10 10
data access pattern percent of sequential IO	99%-1% 5%

Table 2: Workload parameter settings.

4.2. Workload Model

The workload model is homogeneous, i.e. it consists of queries of one type. As was mentioned above, the complex query is executed in a number of steps. We evaluate the performance for the query which has two execution steps.

- Several regions are scanned. After the scan results are send to the client (query initiator machine).
- Some of the objects obtained on the first step are updated. Updates take place in the same regions, and thus are performed by the same region servers.

The number of nodes participating in query execution is distributed according to Poisson distribution with mean M. The number of data objects query reads per region server is exponentially distributed with the mean N_r, writes count is distributed in a similar way with the mean N_w. Reads and writes are distributed among the nodes uniformly.

The 99%-1% data access pattern is used. 99% of transactions references 1% of data objects. In our model the data access pattern influences both the buffer pool hit rate and the data contention.

For the model simplicity we assumed that the data object version has a page size. It is reasonable assumption, because HBase was not designated to work with objects of small size. Thus, the granule for CC algorithm is represented by a page in our model.

Workload parameters can be found in the Table 2.

4.3. Query Execution Model

Each query has a home node. The home node is the client machine it has arrived to. After a query has arrived, it is split to a number of subqueries, each of which is local

to one of execution nodes. Each subquery is modeled as a list of data references (pages) to be processed. For each subquery the message is sent to the execution site to start up the execution. Each query joins the Net queue and consumes CPU time to send a message to execution sites.

When the subquery start message arrives to the execution site, the subquery is created and it joins the Net queue on the execution site and consumes CPU time needed to read a message. Than it consumes CPU to begin execution.

The subquery execution consists of Begin-Of-Execution (BOE) step, a number of data reference processing iterations and End-Of-Execution (EOE) step.

For each data object reference it goes to the buffer pool to get the data. If the required page is not already in memory, subquery consumes processor time to start IO and joins the disk queue. Than the subquery goes to CPU queue and consumes time to process data reference (to find appropriate version). In case there are some more references the subquery repeats a similar data reference processing iterations.

After the subquery has successfully finished its execution it sends requested data (if present) and the READY message to the home node. After that, the subquery is finished, so it goes to CPU queue and consumes time for EOE step.

The client receives data from all the participated servers, process it and sends a message to each participant to perform updates of the second execution stage.

A write subquery is executed in the similar way with a read subquery. Its behaviour differs in data processing iteration. The write subquery firstly consumes CPU for reference processing and after that writes the log page to the file system and enqueues the page write (it does not wait until the page will be written).

4.4. Transaction Execution Model

Read subtransactions do not differ from read subqueries in their execution model. Although they may consume more CPU time for a data reference processing to find appropriate version, the performed steps are same.

Write subtransaction performs version check before each write. If the version check failed, the transaction is to be aborted, and subtransaction goes to the Net node to send an ABORT message. The probability of version check success will be discussed later.

The probability that versions needed for version check were not flushed to a file since the transaction had started is quite high for not write intensive workload. Therefore, the version check step is modeled as processing of N_{vc} pages, each of which is already in buffer pool, so disk IO is not needed.

After the write subtransaction has finished execution it sends READY to the home node and waits for COMMIT or ABORT message in reply. The transaction initiator waits until all READY messages (or one ABORT message) will be received. Than it sends the COMMIT (ABORT) message to all executors and the transaction is

finished. After the subquery received the COMMIT (ABORT) message, it writes a log page and goes to the CPU queue to perform the EOE step.

4.5. Abort Probability

Let us consider the probability of version check failure. Suppose the transaction t which was started at the moment u0 is going to update the value of data object x at the moment u.

The failure of version check may be caused by two types of transactions that had updated x recently.

- x may been written by the transaction which was committed after t had started. Given the throughput of the server $T(u)$ for the moment u, the number of transactions committed in the time interval $[u_0, u]$ can be approximated by $K(u,u_0) = N_w * \frac{A(u) + 2*T(u)*(u-u_0)}{2}$.
- x may been written by the transaction, that had not already committed. Let $A(u)$ denote the number of write subtransaction active at the server as for the moment u. The average number of writes made by active subtransactions may be approximated by $N_w / 2$. The number of writes made by all active transaction as for a moment u may be estimated with $N_w * A(u) / 2$.

The count of updates written by transactions which execution interval overlaps with execution interval of t may be approximated by

$$K(u,u_0) = N_w * \frac{A(u) + 2*T(u)*(u-u_0)}{2}.$$

Let us consider the $a-b$ data access pattern and the case when t belongs to majority of transactions. The number of updates made by transactions of major class is $a*K(u,u_0)$, while the number of data objects t can access is $D*b$. The probability of observing update made by a transaction of the same class is

$$\frac{a*K(u,u_0)}{D*b}.$$

The probability of observing updates made by transactions of the other class is

$$\frac{(1-a)*K(u,u_0)}{D}.$$

Thus, for a transaction of the first type the probability of version check failure can be estimated by

$$P_1(u,u_0)=\frac{b*(1-a)+a}{b}*\frac{K(u,u_0)}{D}$$

and for a transaction of the second type by

$$P_2(u,u_0)=\frac{K(u,u_0)}{D}.$$

5. Simulation Results

The main goal of our performance study consists in evaluation of performance degradation caused by additional overheads introduced by the protocol. We measured the performance of the protocol in terms of the system throughput and average response time. We evaluated the throughput degradation for a particular parameter settings as the difference between the EC and SI throughputs divided by EC throughput.

In this section we use the abbreviation EC to denote HBase version, which uses eventual consistency, and SI for HBase, which uses our protocol. For most of the experiments we present two diagrams: one depicts normal workload results, while the other reflects the situation when all the available resources are already saturated. We have payed a special attention to the saturated system performance to examine the influence of transaction aborts on the throughput.

Fig. 1 and 2 plot throughput results against the active queries count in the system. The maximum degradation obtained for normal workload was 3.3% and 7.7% for saturated system. In the latter case transaction aborts make significant contribution to the throughput degradation. As the number of queries in the system increase, the throughput degradation raises to the mentioned maximum of 7.7% for 1400 queries. When the number of queries is equal to 100, aborts number per second is less than 30% of difference between EC and SI throughputs. The rate of aborts count per second to difference of throughputs reaches the maximum of 98.8% at the point of 700 queries.

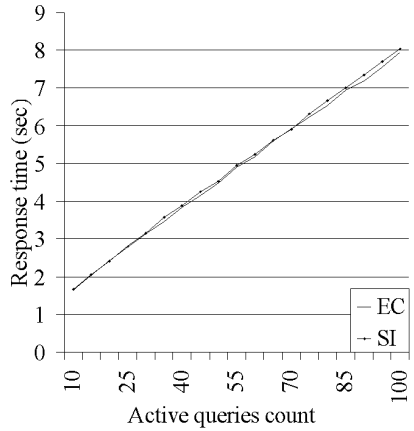


Figure 1: Normal workload throughput against the active queries count.

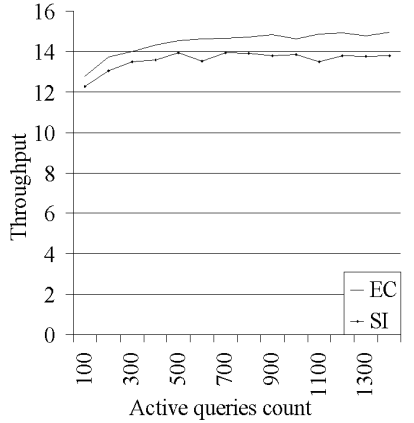


Figure 2: Saturated workload throughput against the active queries count.

Fig. 3 and 4 give the average response time against the active queries count in the system. Relative response time increase is quite small, it does not exceed 3%. After the queries count reaches 35 the relative response time increase does not change significantly. It remains about 3% for the highly saturated workload too.

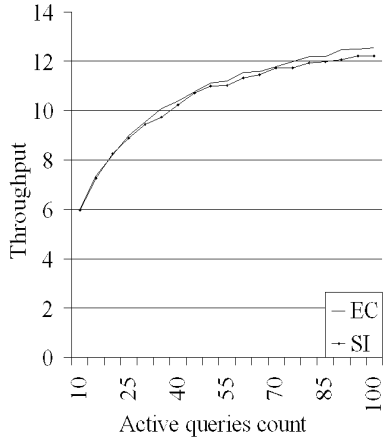


Figure 3: Normal workload response time against the active queries count.

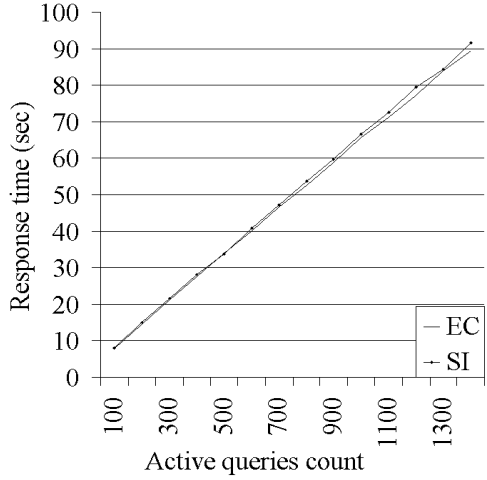


Figure 4: Saturated workload response time against the active queries count.

As expected the use of distributed commit protocol does not make considerable contribution to the average response time for the long-running transactions. The disk system appears to be the bottleneck for this kind of workload. The performance of the system is substantially influenced by the buffer pool hit rate.

Fig. 5 and 6 shows the throughput results for EC and SI against buffer pool hit rate. These experiments were conducted with the systems running 70 (Fig. 5) and 400 (Fig. 6) transactions. The partition size was varied to provide needed buffer pool hit rate. The results obtained from both experiments are similar. In both experiments systems show significant speedup with buffer pool hit rate verging towards 1. EC outperforms SI by 0.51-6.3% in case of normal workload and by 2.9-6.7% otherwise.

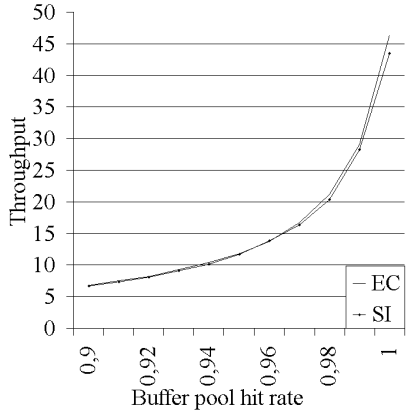


Figure 5: Normal workload throughput against buffer pool hit rate.

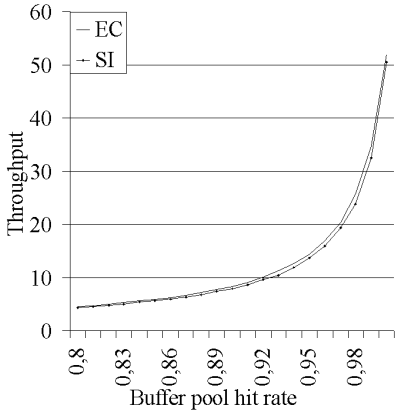


Figure 6: Saturated workload throughput against buffer pool hit rate.

Fig. 7 gives response time against buffer pool hit rate for the system running 70 queries. Response time is not as sensible to buffer pool hit rate as throughput. The graph is close to linear. However, relative response time increase grows rapidly with buffer pool hit rate getting closer to 1 and reaches the maximum of 6.25% for normal workload and 3.87% for saturated workload.

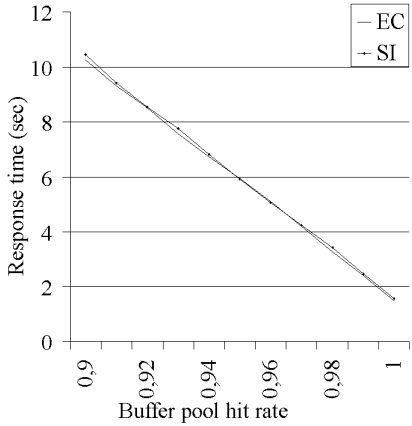


Figure 7: Normal workload response time against buffer pool hit rate.

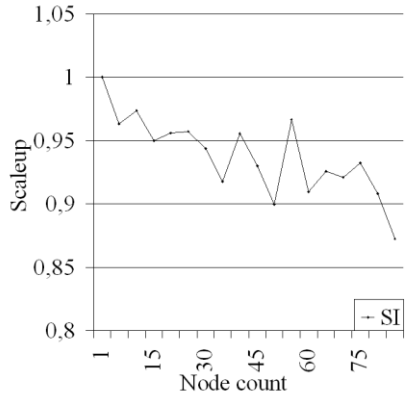


Figure 8: SI scaleup.

Fig. 8 presents SI scaleup against the node count in the system. Scaleup characterizes the ability of the N-times larger system to perform N-times larger job. Scaleup is defined ([15]) as a rate of small system elapse time on small problem to large system elapse time on large problem.

To evaluate the scaleup we slightly changed the workload in such a way that every query spans all region servers. We varied the number of nodes in the cluster from 10 to 85, while the number of transactions in the system remained constant. Experiments were conducted with 10 simultaneously executed transactions.

In our model the scaleup decrease main reason is the overhead introduced by distributed commit protocol. This overhead increases linearly with the increasing of the number of nodes the transaction spans.

Scaleup value on the graph is in 0.87-0.97 bounds, the minimum of 0.87 is reached for the cluster with 85 machines. Scaleup is not significantly lower than linear, so the distributed commit protocol does not decrease the performance dramatically.

6. Model Validation

We have implemented the protocol in HBase. Initially we have evaluated the protocol performance using the prototype. We have measured the prototype performance for a workload, which consisted of short-running queries. Each query read and updated 3 values of double type. The prototype has shown relatively good results for such workload: the throughput was 3 times lower than that of initial HBase version.

To validate our model mentioned test set was changed. If each query writes 3 double values, size of log entries to be flushed to disk before query finishes is small. So we replaced double values with 4Kb sized byte arrays. After the changes had been made we discovered HBase throughput being ten times greater than that of the prototype.

We suppose the prototype implementation showed such a poor performance, because it copies data objects for internal purposes. For every single get and put operation at least one data object is copied. We suppose that despite the fact that main memory reads and writes are extremely fast, copying of data objects may affect the performance in case of no disk accesses.

The values of model parameters for EC were selected in such a way, that EC throughput was as close as possible to HBase throughput. The experiment conducted with the same parameters using SI model has shown the throughput 1.5 times lower than for EC (1200 vs. 1800).

To have the SI model performance close to that of the prototype, we adjusted the mean page processing time parameter to be 270000 instruction instead of default 25000. The primary goal of this experiment was to validate the abort probability model. Fig. 8 shows the prototype throughput and SI model throughput, and 9 shows abort rates obtained. The difference between the aborts rates obtained from

the model and the prototype has not exceeded 3% being about 2% at average. We conclude that our model captures the system behavior adequate.

7. Summary

In this work we presented SI protocol and evaluated its performance. The advantage of the protocol under EC is obvious: the data storage user is provided with SI consistency guarantees. However, the protocol introduces several additional overheads, which may affect the performance. We have modeled the performance for the particular kind of workload and validated the model using the prototype we have presented in [12]. Although, the protocol implementation prototype has shown quite poor performance, it does not imply that the protocol is ineffective itself.

We have payed a special attention to the performance degradation caused by overheads introduced by distributed commits and aborts by protocol reasons. The ability of the protocol to scale was examined as well. The simulation results has shown that use of the protocol does not lead to significant performance degradation for this workload type.

References

[1] R. Agrawal, M.J. Carey and M. Livny. Concurrency control performance modeling: Alternatives and implications. ACM Transactions on Database Systems, 12(4):609–654, 1987.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In Proceedings of the 1995 ACM SIGMOD international conference on Management of data, volume 24, pages 1–10, New York, 1995.

[3] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes and R.E. Gruber. Bigtable: A distributed storage system for structured data. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, volume 7, pages 295–310, 2006.

[4] K. Daudjee and K. Salem. Lazy database replication with Snapshot Isolation. In Proceedings of the 32nd international conference on Very Large Databases, 2006.

[5] R. Marek and E. Rahm. Performance evaluation of parallel transaction processing in Shared Nothing database systems. In Proceedings of the 4th International PARLE Conference on Parallel Architectures and Languages Europe, pages 295–310, Paris, 1992.

[6] M. Nicola and M. Jarke. Performance modeling of distributed and replicated databases. IEEE Transactions on Knowledge and Data Engineering, 12(4), 2000.

[7] E. Rahm. Parallel query processing in Shared Disk database systems. In Proceedings of 5th Int. Workshop on High Performance Transaction Systems, Asilomar, 1993.

[8] P. Di Sanzo, B. Ciciani and F. Quaglia Sapienza. A performance model of multi-version concurrency control. In Proceedings of IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems, pages 1–10, 2008.

[9] Naval Postgraduate School, March 2010. Simkit.

[10] Y.C. Tay. Issues in modelling locking performance. In Hideaki Takagi, editor, Stochastic Analysis of Computer and Communication. Elsevier Science Publishers B.V. (North-Holland), New York, 1990.

[11] A. Thomasian. Concurrency control: Methods, performance, and analysis. ACM Computing Surveys, 30(1), 1998.
[12] D. Vasilik. Implementing Snapshot Isolation in HBase. Diploma thesis, Saint Petersburg State University, 2010.
[13] H. Tam Vo, C. Chen, and B. Chin Ooi. Towards elastic transactional cloud storage with range query support. Proceedings of the VLDB Endowment, 3(1-2), 2010.
[14] C. Zhang and H. De Sterck. Supporting multi-row distributed transactions with Global Snapshot Isolation using bare-bones HBase. In Proceedings of the 11th ACM/IEEE International Conference on Grid Computing (Grid 2010), pages 295–310, Brussels, 2010.
[15] D.J. DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Processing. Communications of the ACM, Vol. 36, No. 6, June 1992.

Оценка Производительности Протокола Реализации Snapshot Isolation

Д.Н. Василик
СПбГУ
Dmitri.Vasilik@gmail.com

Аннотация. Уровень изоляции Snapshot Isolation (SI) широко используется в коммерческих системах баз данных. Мы разработали простой прокол реализации SI для распределенных СУБД и реализовали его в Apache HBase, распределенном хранилище данных с открытым исходным кодом. В данной работе представлена оценка его производительности в OLAP задачах в распределенном кластере HBase. Для валидации модели были использованы результаты измерений на одно-серверной конфигурации.

Ключевые слова: Snapshot Isolation; распределенные системы; моделирование производительности; HBase.

Empirical study of parallel SQL query execution

Kirill Smirnov, George Chernishev
Kirill.k.smirnov@math.spbu.ru, Chernishev@gmail.com

Abstract. In this paper we experiment with two major types of query parallelization techniques - intra and inter operator parallelism and their combinations. We evaluate these techniques applied to a query tree with a number of join operators in the multithreaded environment. In our experiments we vary thread count, buffer size, workload characteristics, measure the relative performance of several distinct join algorithms and observe several modes which appear in the series of experiments.

Keywords: database, experimentation, parallelism, multithreading, query execution

1. Introduction

Distributed database systems (DDBS) had appeared more than three decades ago and since that time there is constant demand in DDBS technologies. The driving force of this demand is the ever-growing volume of data which require processing. The scalability of distributed databases gives us a promise to handle this mass of data. The present demands new effective algorithms, models, architectural solutions so the field remains active and relevant.

In early 90s a substantial portion of research effort was dedicated to ascertainment of how we should divide the work (speaking more precisely a query execution plan or a part of it) between processing elements. This research effort was not specific to a distributed, but aimed for a wide range of database systems. Extensive experiments were conducted regarding the query plan execution techniques, and a pool of knowledge was formed. Now, query execution parallelism is the commonplace and is employed by a various commercial systems.

Despite these experiments, nowadays, the same questions had become relevant again due to the following reasons:

1) Shortage of research concerning utilization of threads and processes. Research papers usually had following distinct attitudes: use one process/thread per processing element, allocate much higher number of threads than processing elements [1] or not to touch this issue in detail.
2) The advancement of hardware – a few generations of CPUs had been developed. The improvements were not just clock speeds and cache size

growth, but also more sophisticated ones. Notable examples are multithreading and multicore processing.

3) Advancement of software – mainly thread/process scheduling software involved.

Even alone items two and three can alter or invalidate principles of thread usage in database system.

The goal of this study is to determine how we should utilize threads designing a distributed database which runs on a modern hardware. We re-evaluate a few classical approaches to process joins in a distributed databases taking into consideration threading aspect. There are two ideas being considered: intra and inter operation parallelism and their combinations. The first one is how we can divide our work processing inside one operator treating our cores as separate processing elements. The latter covers matters of executing several operators in different threads.

This paper is organized as follows: in section 2 we briefly review some crucial aspects of DDBS which are related to our study; in section 3 we present our prototype of DDBS and provide some insights into implementation details. Section 4 reviews several possible ways of thread usage for parallelizing join operation in our system. Sections 5 and 6 describe our experiments and their interpretation results respectively.

2. Related work and DDBS technology overview

2.1. Overview of important technology concepts of DDBS

Extensive research in this field had been conducted for decades and a lot of knowledge had been accumulated. In this section we will review some part of it (by giving definitions at least), which will be referred to in this study.

The first aspect to consider designing a distributed database is its architecture. Here we imply the hardware architecture, how the data is distributed, stored and processed on the lowest level. Most popular architectures are shared-nothing, shared-disk and shared memory [1]. This choice impacts all further decisions regarding all constituent parts the design.

The second aspect is the system processing model. There are two basic alternatives and a lot of different combinations. This choice is the choice between pipelined execution and a materialized one. The former is the policy when a tuple passes through all operators of query tree without “stops”. The latter approach means that some “stops” are allowed. The stop is essentially a blocking operator [2], e.g. an operator which requires all tuples to carry out its task. The sort operator is the blocking one, while predicate filtering operator is the non-blocking. The pipelined execution has a huge benefit over materialization-based, namely all operations can be done in memory. While materialization often requires extensive disk operations which are caused by intermediate result being too large to fit into memory.

Another important aspect to consider is the tuple routing mechanisms. There are a few approaches, namely each tuple passes the operators in the same order, or not. The latter is known as adaptive execution [2]. The next major choice is the choice between data-driven (also called producer-consumer) and demand-driven (known as workflow) execution. Demand-driven execution works by "pumping" tuples from the leaves of the tree to the root, through the operators involved. In this scheme, each operator of a higher rank sends a notification request to its children to get tuples. Consequently, this request propagates to the leaf nodes, which usually deal with disk-based operations. The data-driven approach is the vice-versa: the leaf nodes generate intermediate result, notify its parents and submit the result. The data-driven approach is the more appealing one in case of distributed database system, because it has great potential for parallelization. Also, it offers more freedom and convenience implementing leaf operators. However, this approach requires great care programming it, due to synchronization and work distribution issues.

Yet another pair of aspects to consider designing a distributed database system is the operator parallelization methods. Considering a single query optimization task one can name the following ways: inter- and intra-operator parallelism. Intra-operation parallelism refers to the parallel execution of a single operator and inter-operation parallelism means executing a multiple operators in parallel. The first heavily relies on the data partitioning techniques and the latter is more architectural-dependent.

2.2. Threads and processes in databases

Threading and processing aspects are of critical importance and should be taken into consideration building high performance database engine. However, there is shortage of studies related to threading and its impact on operator parallelism. With the increasing popularity of multicore processors the demand for the reevaluation of these threading technologies had risen greatly. Let's briefly review the threading in databases.

One of the first most prominent database systems which implemented threading was Volcano [3]. The following conclusions were presented [3]: it is beneficial to use multiple threads running concurrently on the same processor. However, running each operator in its own dedicated thread is unfeasible: synchronization and thread switching will eat up all the benefits. The same results presented here [4], also another important question is mentioned: how much memory we should provide to the thread. This question affects the number of threads which could be run effectively. In this paper [2] an adaptive query processing is considered, and shown that each adaptive symmetric hash join operator can work effectively in the separate thread.

2.3. Current state of parallelism in contemporary DBMS

In this section we deal not with the idea of running several transactions concurrently, but consider the possibility of executing several select queries in parallel. The idea of executing a single query in parallel, which in most cases

implies inter or intra operator parallelism had become widely applied since 90. Now, almost all major DBMS have capabilities for query parallelism. In most systems the parallelism is controlled using the Degree Of Parallelism (DOP) mechanism. The DOP is the value which describes how much of logical processing elements we would like to dedicate to a query. User may specify the minimum and maximum values or use default settings. Then, the DBMS schedules the work and maps logical processing element into real ones – processors or cores. DBMS can also be tuned to cancel query execution if there are not enough free processing elements. The resulting performance heavily depends on the number of available processors, the data and query specifics and so on. Thus, it requires careful engine tuning to achieve good performance.

We can name Microsoft SQL Server [5], Oracle [6], IBM DB2 [7] products among the commercial systems that offer mechanisms for such parallelism. At the same time, to the best of our knowledge, neither MySQL, nor PostgreSQL have this feature. For these products, there are several third party frameworks exist. These frameworks offer a substitute to query parallelism which can be described as a layer which allows parallelization of a query via horizontal partitioning of the underlying data in a shared-nothing architecture.

Unfortunately, the control interfaces of contemporary DBMS supporting query parallelism do not provide sufficient degree of freedom. We can't have much influence on how the threads would be used. Thus, we can't use these systems to perform our experiments.

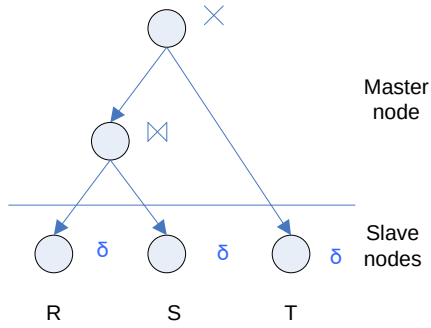
3. Our system

The experiments which are described in this work were conducted using our prototype of distributed query engine developed by authors. This system is devised for simple SPJ conjunctive queries [2] and is capable of serving simultaneously a substantial number of clients. The architecture of the system is described on a Pic. 1 and it is essentially a shared-nothing design with a number of processing nodes distributed among network. We partition nodes into two types: master and slave. There is only one master node per setup and it is responsible for handling incoming queries. For purpose of paper's experiments we run slave processes on the same computer.

Our system is based on assumption that data are pre-distributed and reside on slave nodes. The data are collection of relations, where each attribute may be either integer value or string. Each relation is partitioned horizontally and may be replicated.

The overall design of the system resembles a classical Volcano [4], where QEP (query execution plan) is defined as a tree of various operators and candidate tuples have to pass them in order for result to be produced. In our system there are the following operators: selection (actually consisting of both selection and projection), join and cross-product. Details regarding our implementation of join operator can be

found in part 4, but it is essential to note that the overall architecture was designed for non-blocking operations, so the join operator should be non-blocking too.

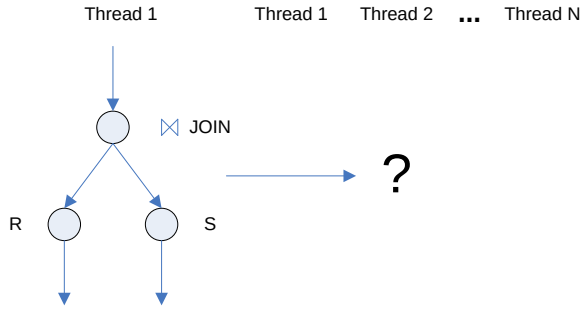


Picture 1: architecture

The QEP generation procedure also is a very unpretentious and is based entirely on heuristics, which resemble classical ones described [8]. The plan generation doesn't affect our benchmarks much because we use very specific loads (described in part 5) to assess the performance. Some details regarding implementation can be found in [9]. To conclude, we should mention that despite its simplicity our system is sufficient for use in our benchmarks to reach our goal.

4. Considered approaches

We considered the following approaches to incorporation of threading into join processing. Our initial algorithm was essentially nested loop join algorithm [4] which was modified to cope with the networking issues. The main difference is the bulk processing technique which provides bufferization to lessen the communication burden (as opposed to per record processing). When the both sets of tuples are received, we sort one of them and use the second as the probing one. Also, we employ in-operation caching to lessen the network stress further. This caching works as follows: if one of the participating relations is small enough to fit in memory, it is put into inner operation cache. This baseline method is working inside a single thread (all joins and cross-products are working inside one thread), Pic. 2.

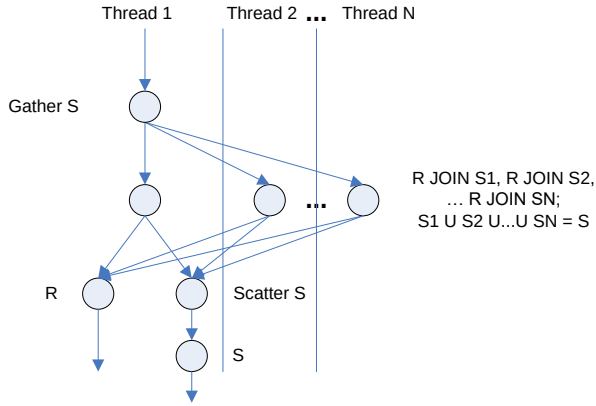


Picture 2: initial join design

To introduce parallelism we consider the technique which employs a number of threads for data processing. The core idea is to partition data supplied by one of the child nodes, while leaving the other one intact, then use different threads to process it (shown on Pic. 3).

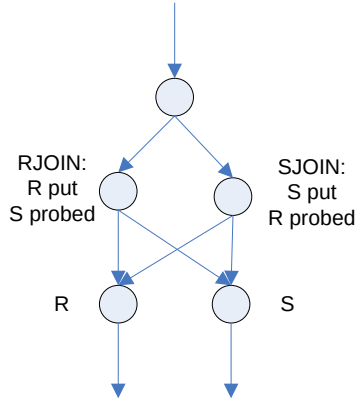
Also, we consider a hash join operator. In this case, the processing of a join operation is separated into two separate phases: a build phase and a probe phase. During the build phase, a table which was chosen as a build one is used in building a hash-table. Then, the next phase is essentially a traversal of a probe table to locate the match. The evident drawback of this technique is the temporary "freeze" of the system. This is caused by the build stage, e.g. the dependent operators should wait until the build phase is finished. This drawback is a critical issue, because it breaks [2] the pipelining execution, or this join operator becomes a blocking operator. If we are dealing with the QEP of a sufficient depth the blocking operators can become the source of a major slowdown in processing of the whole query. Also, this blocking especially adversely affects inter-operation parallelism (threads will wait for tuples to arrive). Thus, blocking operations should be taken into consideration.

To parallelize this join we employ the similar technique: we partition the build relation into a few fragments and each fragment is processed inside its dedicated thread.



Picture 3: join parallelization

To overcome the blocking nature of the previous approach we would consider a symmetric hash-join operator [2]. The core idea is to have two join nodes, which have different building relations. The processing happens as follows: given a tuple from relation R we probe it against build relation of S join operator and at the same time, we use this tuple in building relation of R join operator. Likewise, given a tuple from relation S we use it to probe it in R and build it in S operators. This join is correct due to the fact that if a tuple arrives too early (e.g. no relevant tuples from other relation had been built) it would be probed later, because it is built in other join operator. This type of hash join operator is a non-blocking operator. However, this advantage comes at a price – additional memory and time constraints (twice as much as a simple hash join) and more sophisticated programming techniques are required.



Picture 4: symmetric hash join operator design

To implement data partitioning in our system we introduced two new operators (much alike exchange operator in volcano system), for distribution and collection of the results. These operators are inserted in QEP around join operator which is going to be parallelized. We call helper threads the threads which hold the additional join nodes which will appear as the result of our parallelization.

5. Experiments

5.1. Parameters used in experiments

We implemented the proposed approaches and examined them using our system on a variety of workloads. First we characterize the data used in our experiments:

1) Size of the tables are 150Mb (first type of workload)
2) Each table contained three attributes: primary key, integer attribute used for joining and predicate which defines selectivity. We used only 4 byte integer data in our experiments and it is uniformly distributed.

The processing details:

1) Number of helper threads per processing node (QEP one): 1, 2, 5, 10
2) Methods (node types) of join:
    a. Hash join
    b. Block nested loop (sort)
    c. Block multithreaded nested loop (sort)
    d. Block multithreaded nested loop (hash)
    e. Symmetric hash join

3) Operation cache (also, size of bulk which is transferred from one QEP node to another): 2, 5, 10 MB
4) Join operators use results of a sequential scan over table (no index is used)

Query type details:

1) Query to evaluate intra-operation parallelism: a single join of two tables
2) Query to evaluate inter-operation parallelism: a QEP which contains 2 joins and 3 tables

5.2. Hardware and software

The following setup was used: Intel(R) Pentium(R) D CPU 3.00GHz, RAM 3Gb, which run GNU/Linux 2.6.35.10 x86, and we used gcc 4.5.2.

NPTL threads are used for multithreading, threads communicate via chunk of shared memory. Notifications are implemented using POSIX pipes along with select core. For hashes we used c++0x unordered map STL container.

5.3. Experiments performed

We had evaluated our algorithms on two distinct groups of workloads, namely workloads which evaluate intra-operator parallelism and inter-operator parallelism. The first ones evaluate our algorithms using QEP which contains only one join operation, which is fueled by two scan operations. The Table 1 contains summarized results over all runs in this group of experiments (here we measured execution time in seconds). Each experiment was repeated 100 times and the average value was used. Detailed results are presented in diagrams 1-3, where each histogram shows results for a fixed cache size.

Method/Cache size	2Mb	5Mb	10Mb
Block NL Hash	50.69	50.77	56.60
Block NL Sort	57.50	56.56	57.45
Symmetric HJ	67.60	66.98	68.16
Block NL Hash MT 2	47.96	45.28	44.78
Block NL Sort MT 2	48.36	45.74	45.43
Block NL Hash MT 5	46.75	45.56	44.32
Block NL Sort MT 5	47.31	45.05	47.64
Block NL Hash MT 10	45.34	45.19	45.94
Block NL Sort MT 10	51.62	44.23	46.47

Table 1: Intra-operation parallelism

In this experiment we varied the following parameters: operation cache and number of helper threads. The last six methods describe parallelized join methods (for threads amount of 2, 5, 10 respectively) of two distinct methods: sort and hash. Sort method is a plain block nested loop, where one relation's block is sorted before probing by another.

In multithreaded tests we observed several modes. This phenomenon can be explained by system thread scheduler fluctuations. The distributions of measured values for block nested loop are presented in histograms 1-2, bin size is determined by Sturges' formula in each case. The Table 1 presents main mode value.

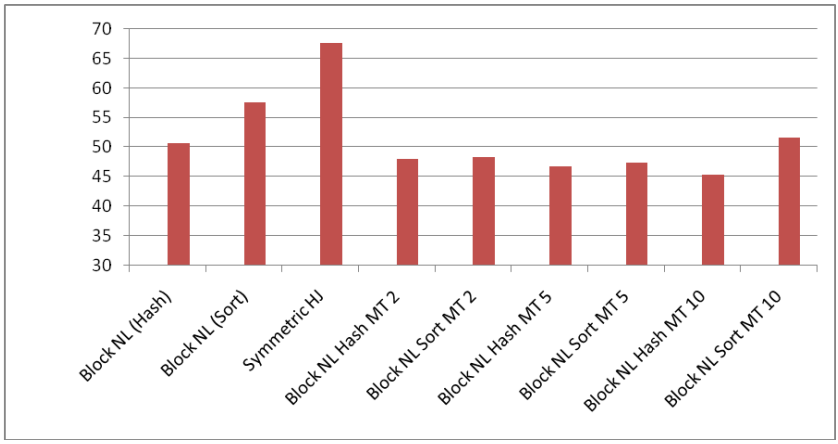


Diagram 1: 2 Mb cache

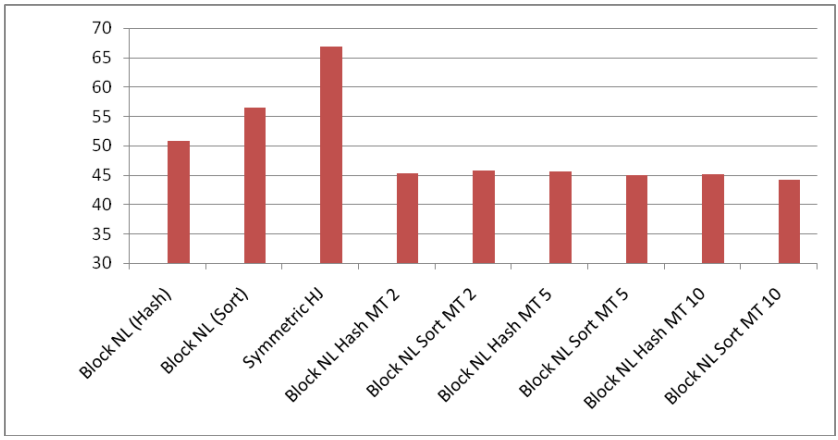


Diagram 2: 5 Mb cache

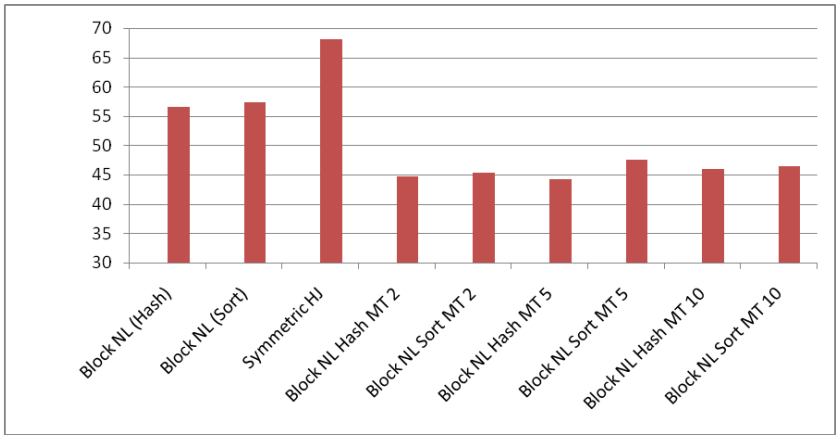
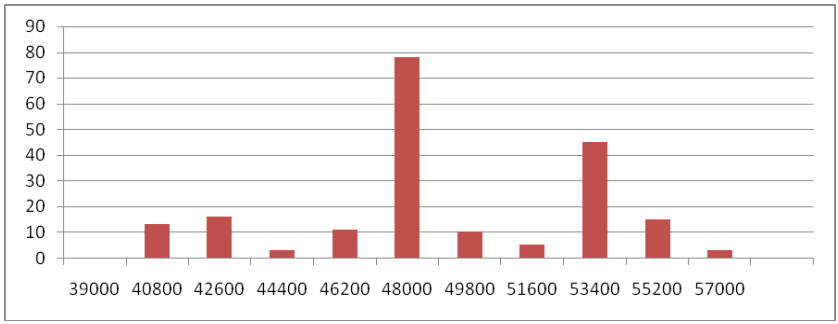
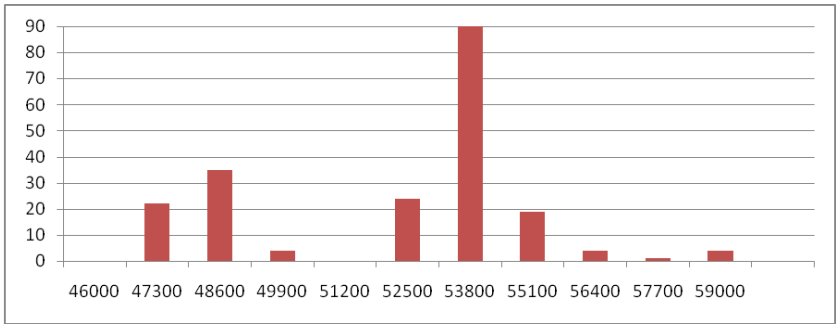


Diagram 3: 10 Mb cache



Histogram 1: 2 threads



Histogram 2: 10 threads

The Block NL Hash MT version of the operator is constructed in such way, that hash relation is always hashed and fully resides in the memory. So, it can be considered a true hash join. We can draw the following conclusions:

1) Parallelization does help to speed-up join processing on modern hardware, under several conditions. Our initial experiments were conducted with bulk size of 1Mb and parallelized joins had shown worse results that single-threaded ones.
2) Increasing bulk size does increase performance, up to some point. Our assumption is the following: there is a tradeoff between a number of data transfers and degree of thread utilization, e.g. increasing amount of work one may ensure that communication does not happens too often, but we have to communicate frequently enough in order to have a benefits of parallelism and pipelining.
3) Increasing number of threads further does not improve performance on dual core CPU.
4) Parallelization leads to multi-mode distribution of measured execution time. This behavior may affect estimation of parallel plan execution time.
5) Examining single-threaded join methods (first three mentioned operators) had shown us the following:
    a. Hash join better than nested loop
    b. Symmetric hash join is actually worse than nested loop
6) Other results (not shown here) state the following: hash-based parallel algorithms a bit worse than sort-based ones on workload which converges to cross-product and vice versa (join which has both tables filtered on the same field and value). Our hypothesis is the following: this behavior could be attributed to our implementation of sort-based algorithm (we implemented a quicksort algorithm) which is tightly coupled with pointers for efficiency and to a large amount of collisions which arise considering this workload.

Another group of experiments we considered was the group which tested the QEP containing 2 join nodes. The goal was to run each join in separate threads. We restrict ourselves to having both join operators of the same type. The results of these experiments were the following:

1) There are not much intermediate results of the first join processing to load second one in such way to see different results. So is unfeasible to use the original dataset in order to evaluate these approaches. To cope with this problem we raised scan selectivity 5 times.
2) Under the new conditions we got the following results:
    a. Difference between non-threaded hash join and symmetric hash join is about 5% (hash join is better). Overall performance of QEP had increased over the same case of the first group (8% loss). We presume that this happens due to enabling of pipelining.

b. Threaded versions worse 2.5 times compared to single-threaded hash join on the smallest buffer size (2MB). Unfortunately, now, it is unclear whether our programming technique was sufficient to implement this threading correctly or it is the hardware processing specifics guarantee us these bad results (different operators cause constant cache resets). We can also say for sure, that it is not thread scheduler problem, because adding more threads does not have any effect.
c. Bulk size has a huge impact on performance in such case also. It is becoming even more evident here. Increasing bulk size increases performance of threaded joins almost 2 times (single-threaded ones get approximately 10%), but still threaded joins lose to single threaded ones.

6. Conclusions and future work

We presented results of evaluation of a set of join algorithms. The results imply that thread usage is beneficial for join processing and can give up to 50% increase in performance on simple queries. However, these approaches require extra programming effort, extensive parameter tuning and heavily rely on thread scheduler. Symmetric hash join has a promise for deep QEPs over threaded joins on such hardware.

This is a relatively small piece of work for this vast and evolved topic. A lot of aspects require our attention and they hadn't been touched at all in this paper. The future work will include parallelization of symmetric hash join algorithm, deeper join trees for more thorough analysis of threading effects to pipelining, proper evaluation of scale-up, and possibly evaluation on more productive equipment.

References

[1] M. Tamer Özsu and Patrick Valduriez. 1999. Principles of Distributed Database Systems (2nd Ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[2] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive query processing. Found. Trends databases 1, 1 (January 2007), 1-140.

[3] Donald Kossmann. 2000. The state of the art in distributed query processing. ACM Comput. Surv. 32, 4 (December 2000), 422-469.

[4] Goetz Graefe. 1993. Query evaluation techniques for large databases. ACM Comput. Surv. 25, 2 (June 1993), 73-169.

[5] http://msdn.microsoft.com/en-us/library/ms188611.aspx Degree of parallelism SQL Server 2008 R2, access date 14.06.2011

[6] Oracle (R) Database Datawarehousing Guide 10g Release 2 (10.2)

[7] Yun Wang. 1995. DB2 Query Parallelism: Staging and Implementation. In Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95), Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 686-691.

[8] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In Proceedings of the 1979 ACM SIGMOD international conference on Management of data (SIGMOD '79). ACM, New York, NY, USA, 23-34.
[9] Kirill Smirnov and George Chernishev. 2011. Networking and multithreading architectural aspects of distributed dbms (in Russian), Software and Systems 1(93) (March 2011), 164-168

Экспериментальное исследование параллельного исполнения SQL запросов

К. К. Смирнов, Г.А. Чернышев
Kirill.k.smirnov@math.spbu.ru, Chernishev@gmail.com

Аннотация. В данной работе мы экспериментально изучаем два основных типа параллельного исполнения запросов – интер и интра операционный параллелизм и их комбинации. Мы рассматриваем эти техники в применении к дереву запроса, содержащего несколько операторов соединения, в условиях многопоточности. В наших экспериментах мы варьируем количество потоков, размер буфера, характеристики тестовых данных для того, чтобы сравнить производительность различных алгоритмов соединения. В ходе экспериментов было выявлено мультимодовое поведение системы.

Ключевые слова: базы данных, эксперименты, параллелизм, многопоточность, исполнение запросов

Integrating Fuzzy *c*-Means Clustering with PostgreSQL[1]

R. M. Miniakhmetov
tavein@gmail.com

Abstract. Many data sets to be clustered are stored in relational databases. Having a clusterization algorithm implemented in SQL provides easier clusterization inside a relational DBMS than outside with some alternative tools. In this paper we propose Fuzzy *c*-Means clustering algorithm adapted for PostgreSQL open-source relational DBMS.

Keywords: fuzzy c-means; fcm; fuzzy clustering; postgresql; integrating clustering; relational dbms.

1. Introduction

Integrating clustering algorithms is a topic issue for database programmers [1]. Such an approach, on the one hand, encapsulates DBMS internal details from application programmer. On the other hand, it allows to avoid overhead connected with export data outside a relational DBMS. The *Fuzzy c-Means (FCM)* [2], [3], [4] clustering algorithm provides a fuzzy clustering of data. Currently this algorithm have many implementations on a high-level programming languages [5], [6]. For implementation the FCM algorithm in SQL we choose an open-source PostgreSQL DBMS [7].

The paper is organized as follows. Section 2 introduces basic definitions and an overview of the FCM algorithm. Section 3 proposes implementation of the FCM in SQL called pgFCM. Section 4 briefly discusses related work. Section 5 contains conclusion remarks and directions for future work.

2. The Fuzzy c-Means Algorithm

The K-Means [8] is one of the most popular clustering algorithms, it is a simple and fairly fast [9]. The FCM algorithm generalizes K-Means to provide fuzzy clustering, where data vectors can belong to several partitions (*clusters*) at the same time with a

[1] This paper is supported by the Russian Foundation for Basic Research (grant No. 09-07-00241-a) and the Ministry of Education and Science of the Russian Federation (state contract No. 07.514.11.4036).

given weight (*membership degree*). To describe FCM we use the following notation:

- $d \in \mathbb{N}$ – dimensionality of a data vectors (or data items) space;
- $l \in \mathbb{N}: 1 \le l \le d$ – subscript of the vector's coordinate;
- $n \in \mathbb{N}$ – cardinal number of training set;
- $X \subset \mathbb{R}^d$ – training set for data vectors;
- $i \in \mathbb{N}: 1 \le i \le n$ – vector subscript in a training set;
- $x_i \in X$ – the i-th vector in the sample;
- $k \in \mathbb{N}$ – number of clusters;
- $j \in \mathbb{N}: 1 \le j \le k$ – cluster number;
- $C \subset \mathbb{R}^{k \times d}$ – matrix with clusters' centers (*centroids*);
- $c_j \in \mathbb{R}^d$ – center of cluster j, d-dimensional vector;
- $x_{il}, c_{jl} \in \mathbb{R}$ – l-s coordinates of vectors x_i and c_j respectively;
- $U \subset \mathbb{R}^{n \times k}$ – matrix with membership degrees, where $u_{ij} \in \mathbb{R}: 0 \le u_{ij} \le 1$ is a membership degree between vector x_i and cluster j;
- $\rho(x_i, c_j)$ – distance function, defines a membership degree between vector x_i and cluster j;
- $m \in \mathbb{R}: m > 1$ – the fuzzyfication degree of objective function;
- J_{FCM} – objective function.

The FCM is based on minimization of the *objective function* J_{FCM}:

$$J_{FCM}(X, k, m) = \sum_{i=1}^{N} \sum_{j=1}^{k} u_{ij}^m \rho^2(x_i, c_j) \qquad (1)$$

Fuzzy clusterization is carried out through an iterative optimization of the objective function (1). Membership matrix *U* and centroids c_{ij} are updated using the following formulas:

$$u_{ij} = \sum_{t=1}^{k} \left(\frac{\rho(x_i, c_j)}{\rho(x_i, c_t)} \right)^{\frac{2}{1-m}} \quad (2)$$

$$\forall j, l \quad c_{jl} = \frac{\sum_{i=1}^{n} u_{ij}^{m} \cdot x_{il}}{\sum_{i=1}^{n} u_{ij}^{m}} \quad (3)$$

Let s is a number of iteration, $u_{ij}^{(s)}$ and $u_{ij}^{(s+1)}$ are elements of matrix U on steps s and $s+1$ respectively, and $\varepsilon \in (0,1) \subset \mathbb{R}$ is a termination criterion. Then the termination condition can be written as follows:

$$\max_{ij} \{ | u_{ij}^{(s+1)} - u_{ij}^{(s)} | \} < \varepsilon \quad (4)$$

Objective function (1) converges to a local minimum (or a saddle point) [10].

Input: X, k, m, ε

Ouput: U

1. $s := 0, U^{(0)} := (u_{ij})$ {initialization}
2. **repeat**
   {computation of new centroids' coordinates}
3. Compute $C^{(s)} := (c_j)$ using formula (3) where $u_{ij} \in U^{(s)}$
   {update matrixes values}
4. Compute $U^{(s)}$ and $U^{(s+1)}$ using formula (2)
5. $s := s+1$
6. **until** $\max_{ij} \{ | u_{ij}^{(s)} - u_{ij}^{(s-1)} | \} \leq \varepsilon$

Fig. 1. The Fuzzy c-Means Algorithm.

Algorithm showed on Fig. 1 presents the basic FCM. The input of algorithm receives a set of data vectors $X = (x_1, x_2, \ldots, x_n)$, number of clusters k,

fuzzyfication degree *m*, and termination criterion ε. The output is a matrix of membership degrees *U*.

3. Implementation of Fuzzy c-Means Algorithm using PostgreSQL

In this section we suggest pgFCM algorithm as a way to integrate FCM algorithm with PostgreSQL DBMS.

3.1. General Definitions

To integrate FCM algorithm with a relational DBMS it is necessary to perform matrixes U and X as relational tables. Subscripts for identification elements of relational tables are presented in Table 1 (numbers n, k, d are defined above in a section 2).

Subscript	Range	Semantics
i	$\overline{1,n}$	vector subscript
j	$\overline{1,k}$	cluster subscript
l	$\overline{1,d}$	vector's coordinate subscript

Table 1. Data Elements Subscripts.

As a function of distance $\rho(x_i, c_j)$, without loss of generality, we use the Euclidean metric:

$$\rho(x_i, c_j) = \sqrt{\sum_{l=1}^{d} (x_{il} - c_{jl})^2} \qquad (5)$$

To compute the termination criterion (4) we introduce the function δ as follows:

$$\delta = \max_{ij} \{| u_{ij}^{(s+1)} - u_{ij}^{(s)} |\} \qquad (6)$$

3.2. Database Scheme

Table 2 summarizes database scheme of pgFCM algorithm (underlined columns are primary keys).

No.	Table	Semantics	Columns	Number of rows
1	SH	training set for data vectors (horizontal form)	$\underline{i}, x1, x2, \ldots, xd$	n
2	SV	training set for data vectors (vertical form)	$\underline{i,l}, val$	$n \cdot d$
3	C	centroids' coordinates	$\underline{j,l}, val$	$k \cdot d$
4	SD	distances between x_i and c_j	$\underline{i, j}, dist$	$n \cdot k$
5	U	degree of membership vector x_i to a cluster c_j on step s	$\underline{i, j}, val$	$n \cdot k$
6	UT	degree of membership vector x_i to a cluster c_j on step $s+1$	$\underline{i, j}, val$	$n \cdot k$
7	P	result of computation function δ (6) on step s	$\underline{d, k, n}, s, delta$	s

Table 2. Relational Tables of pgFCM Algorithm

In order to store sample of a data vectors from set X it is necessary to define table $SH(\underline{i}, x1, x2, \ldots, xd)$. Each row of sample stores vector of data with dimension d and subscript i. Table SH has n rows and column i as a primary key. FCM steps demand aggregation of vector coordinates (sum, maximum, etc.) from set X. However, because of its definition, table SH does not allow using SQL aggregation functions. To avoid this obstacle we define a table $SV(\underline{i,l}, val)$, which contains $n \cdot d$ rows and have a composite primary key (i,l). Table SV represents a data sample from table SH ans supports SQL aggregation functions max and sum. Due to store coordinates of cluster centroids temporary table $C(\underline{j,l}, val)$ is defined. Table C has $k \cdot d$ rows and the composite primary key (j,l). Like the table SV, structure of table C allows to use aggregation functions. In order to store distances $\rho(x_i, c_j)$ table $SD(\underline{i, j}, dist)$ is used. This table has $n \cdot k$ rows and the composite primary key (i, j). Table $U(\underline{i, j}, val)$ stores membership degrees, calculated on *s*-th step. To store membership degrees on $s+1$ step similar table $UT(\underline{i, j}, val)$ is used. Both tables have $n \cdot k$ rows and the composite primary key (i, j). Finally, table $P(\underline{d, k, n}, s, delta)$ stores iteration

number s and the result of computation function (6) *delta* for this iteration number. Number of rows in table P depends on the number of iterations.

3.3. The pgFCM Algorithm

The pgFCM algorithm is implemented by means of a stored function in *PL/pgSQL* language. Fig. 2 shows the main steps of the pgFCM algorithm.

Input: SH, k, m, eps

Ouput: U

{initialization}

1. Create and initialize temporary tables (U, P, SV, etc.)
2. **repeat**

{computations}

3. Compute centroids coordinates. Update table *C*.
4. Compute distances $\rho(x_i, c_j)$. Update table *SD*.
5. Compute membership degrees $UT = (ut_{ij})$.

{update}

6. Update tables *P* and *U*.

{check for termination}

7. **until** $P.delta \geq eps$

Fig. 2. The pgFCM Algorithm.

The input set of data vectors X stored in table SH. Fuzzyfication degree m, termination criterion ε, and number of clusters k are function parameters. The table U contains a result of pgFCM work.

3.4. Initialization

Initialization of tables SV, U, and P provided by SQL-code `I1`, `I2`, and `I3` respectively. Table SV is formed by sampling records from the table SH.

```
I1: INSERT INTO SV
      SELECT SH.i, 1, x1
      FROM SH;
    ...
    INSERT INTO SV
      SELECT SH.i, d, xd
      FROM SH;
```

For table U a membership degree between data vector x_i and cluster j takes 1 for all $i = j$.

```
I2: INSERT INTO U (i, j, val)
      VALUES (1, 1, 0);
    ...
    INSERT INTO U (i, j, val)
      VALUES (j, j, 1);
    ...
    INSERT INTO U (i, j, val)
      VALUES (n, k, 0);
```

In other words, as a start coordinates of centroids, first d data vectors from sample X are used.

$$\forall i = j \quad u_{ij} = 1 \Rightarrow c_j = x_i$$

When initializing the table P, the number of points k is taken as a parameter of the function $pgFCM$. A data vectors space dimensionality d and a cardinal number of the training set n also provided by function $pgFCM$ parameters. The iteration number *s* and *delta* initializes as zeros.

```
I3: INSERT INTO P(d, k, n, s, delta)
      VALUES (d, k, n, 0, 0.0);
```

3.5. Computations

According to Fig. 2, the computation stage is splitted to the following three sub-steps: computation coordinates of centroids, computation of distances, and computation membership degrees, marked as `C1`, `C2`, and `C3` respectively.

```
C1: INSERT INTO C
      SELECT R1.j, R1.l, R1.s1 / R2.s2 AS val
      FROM (SELECT j, l, sum(U.val^m * SV.val) AS s1
            FROM U, SV
            WHERE U.i = SV.i
            GROUP BY j, l) AS R1,
           (SELECT j, sum(U.val^m) AS s2
            FROM U
            GROUP BY j) AS R2
      WHERE R1.j = R2.j;
C2: INSERT INTO SD
      SELECT i, j, sqrt(sum((SV.val - C.val)^2))) AS dist
```

```
      FROM SV, C
      WHERE SV.l = C.l
      GROUP BY i, j;
```

Through the FCM, computations of the distances provide by formula (2). In formula (3) the fraction's numerator does not depend on t, then we can rewrite this formula as follows:

$$u_{ij} = \rho^{\frac{2}{1-m}}(x_i, c_j) \cdot \left(\sum_{t=1}^{k} \rho^{\frac{2}{m-1}}(x_i, c_t) \right)^{-1} \qquad (7)$$

Thus, the computation of membership degrees can be written as follows:

```
C3: INSERT INTO UT
      SELECT  i,  j,  SD.dist^(2.0^(1.0-m))  *  SD1.den  AS
val
      FROM (SELECT i, 1.0 / sum(dist^(2.0^(m-1.0))) AS den
            FROM SD
            GROUP BY i) AS SD1, SD
      WHERE SD.i = SD1.i;
```

3.6. Update

Update stage of the pgFCM modifies P and U tables as shown below in `U1` and `U2` respectively.

```
U1: INSERT INTO P
      SELECT L.d, L.k, L.n, L.s + 1 AS s, E.delta
      FROM (SELECT i, j, max(abs(UT.val - U.val)) AS delta
            FROM U, UT
            GROUP BY i, j) AS E,
           (SELECT d, k, n, max(s) AS s
            FROM P
            GROUP BY d, k, n) AS L) AS R
```

Table UT stores temporary membership degrees to be inserted into table U. To provide the rapid removal all the table U rows, obtained at the previous iteration, we use the truncate operator.

```
U2: TRUNCATE U;
    INSERT INTO U SELECT * FROM UT;
```

3.7. Check

This stage is the final for the algorithm pgFCM. On each iteration the termination condition (4) must be checked.

To implement the check, the result *delta* of the function (6) from table P is stored in the temporary variable *tmp*.

```
CH1: SELECT delta INTO tmp
     FROM P,(SELECT d, k, n, max(s) AS max_s
             FROM P
             GROUP BY d, k, n) AS L
     WHERE P.s = L.max_s AND P.d = L.d AND P.k = L.k AND
P.n =L.n;
```

After selecting the *delta*, we need to check the condition $\delta < \varepsilon$. Then if this condition is true we should stop, otherwise, work will be continued.

```
CH2: IF (tmp < eps) THEN
       RETURN;
     END IF;
```

The final result of the algorithm pgFCM will be stored in table U.

4. Related Work

Research on integrating data mining algorithms with relational DBMS includes the following. Association rules mining is explored in [11]. General data mining primitives are suggested in [12]. Primitives for decision trees mining are proposed in [13].

Our research was inspired by papers [1], [14], where integrating K-Means clustering algorithm with relational DBMS, was carried out. The way we exploit is similar to mentioned above. The main contribution of the paper is an extension of results presented in [1], [14] for the case where data vectors may belong to several clusters. Such a case is very important in problems connected with medicine data analysis [15], [16]. To the best of our knowledge there are no papers devoted to implementing fuzzy clustering with relational DBMS.

5. Conclusion

In this paper we have proposed the pgFCM algorithm. pgFCM implements Fuzzy *c*-Means clustering algorithm and processes data stored in relational tables using PostgreSQL open-source DBMS. There are following issues to continue our research. Firstly, we plan to investigate pgFCM scalability using both synthetical and real data sets. The second direction of our research is developing a parallel version of pgFCM for distribution memory multiprocessors.

References

[1] C. Ordonez. Programming the K-means clustering algorithm in SQL. Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2004, pp. 823–828.

[2] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. ACM Computing Surveys, 1999, Vol. 31, Iss. 3, pp. 264–323.

[3] J. C. Dunn. A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters. Journal of Cybernetics, 1973, Vol. 3, Iss. 3, pp. 32–57.

[4] J. C. Bezdek. Pattern Recognition with Fuzzy Objective Function Algorithms. Kluwer Academic Publishers, Norwell, USA, 1981, p. 256.

[5] E. Dimitriadou, K. Hornik, F. Leisch, D. Meyer, and Weingessel A. Machine Learning Open-Source Package 'r-cran-e1071', 2010. http://cran.r-project.org/web/packages/e1071/index.html. Reference date: 13.06.2011.

[6] I. Drost, T. Dunning, J. Eastman, O. Gospodnetic, G. Ingersoll, J. Mannix, S. Owen, and K. Wettin. Apache Mahout, 2010. https://cwiki.apache.org/confluence/display/MAHOUT/Fuzzy+K-Means. Reference date: 13.06.2011.

[7] M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of POSTGRES. IEEE Transactions on Knowledge and Data Engineering, March 1990, Vol. 2, Iss. 1, pp. 125–142.

[8] J. B. MacQueen. Some Methods for Classification and Analysis of MultiVariate Observations. Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability, 1967, Vol. 1, pp. 281–297.

[9] P. S. Bradley, U. M. Fayyad, and C. Reina. Scaling Clustering Algorithms to Large Databases. Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining, 1998, pp. 9–15.

[10] J. Bezdek, R. Hathaway, M. Sobin, and W. Tucker. Convergence Theory for Fuzzy *c*-Means: Counterexamples and Repairs. IEEE Transactions on Systems, Man and Cybernetics, 1987, Vol. 17, Iss. 5, pp. 873–877.

[11] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, 1998, pp. 343–354.

[12] J. Clear, D. Dunn, B. Harvey, M. Heytens, P. Lohman, A. Mehta, M. Melton, L. Rohrberg, A. Savasere, R. Wehrmeister, and M. Xu. NonStop SQL/MX primitives for knowledge discovery. Proceedings of the 5th ACM SIGKDD international conference on Knowledge discovery and data mining, 1999, pp. 425–429.

[13] G. Graefe, U. M. Fayyad, and S. Chaudhuri. On the Efficient Gathering of Sufficient Statistics for Classification from Large SQL Databases. Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining, 1998, pp. 204–208.

[14] C. Ordonez. Integrating K-Means Clustering with a Relational DBMS Using SQL IEEE Transactions on Knowledge and Data Engineering, 2006, Vol. 18, Iss. 2, pp. 188–201.

[15] A. I. Shihab. Fuzzy Clustering Algorithms and their Applications to Medical Image Analysis. PhD thesis, University of London, 2000.

[16] D. Zhang and S. Chen. A Novel Kernelized Fuzzy *c*-Means Algorithm with Application in Medical Image Segmentation. Artificial Intelligence in Medicine, 2004, Vol. 32, pp. 37–50.

Интеграция алгоритма кластеризации Fuzzy *c*-Means в PostgreSQL

Р. М. Миниахметов
tavein@gmail.com

Аннотация. Большие объемы данных, которые могут быть кластеризованы, хранятся в реляционных базах данных. Алгоритм кластеризации, реализованный на языке SQL, обеспечивает более легкий процесс кластеризации, по сравнению с использованием внешних утилит. В данной статье предложена реализация алгоритма Fuzzy *c*-Means, адаптированного для реляционной СУБД с открытым исходным кодом PostgreSQL.

Ключевые слова: fuzzy c-means; fcm; нечеткая кластеризация; postgresql; интеграция алгоритма кластеризации; реляционные субд.

Detecting Content Spam on the Web through Text Diversity Analysis

Anton Pavlov, Boris Dobrov
pavvloff@yandex.ru, dobroff@mail.cir.ru

Abstract. Web spam is considered to be one of the greatest threats to modern search engines. Spammers use a wide range of content generation techniques known as content spam to fill search results with low quality pages. We argue that content spam must be tackled using a wide range of content quality features. In this paper we propose a set of content diversity features based on frequency rank distributions for terms and topics. We combine them with a wide range of other content features to produce a content spam classifier that outperforms existing results.

Keywords: search spam; feature analysis; topical diversity.

1. Introduction

Web spam or spamdexing is defined as "any deliberate action that is meant to trigger an unjustifiably favorable relevance or importance for some Web page, considering the page's true value" [1]. Studies show that at least 20 percent of hosts on the Web are spam [2]. Web spam is widely acknowledged as one of the most important challenges to web search engines [3].

There is a wide range of spamming techniques usually aimed at different algorithms used in search engines. This article is dedicated to content spam detection algorithms. Content spamming or term spamming refers to "techniques that tailor the contents of text fields in order to make spam pages relevant for some queries" [1]. We argue that content spam can be detected using a combination of text quality features that cover multiple characteristics of natural texts. In this work we introduce several novel features based on frequency rank distributions for terms and topics that substantially improve content spam classification.

In Section 2 we provide basic assumptions behind our research. In Section 3 we describe the content spam detection framework. Section 4 contains evaluation results. Section 5 is dedicated to future work and conclusions.

1.1. Related Work

Many spam detection techniques have been proposed in recent years during the Web Spam Challenge [4]. Some content features we used were proposed by Ntoulas et. al. [5]. This work showed that compressibility of text and some HTML-related

characteristics distinguish content spam from normal pages. A large amount of linguistic features were explored in a work by Piskorski et. al. [6]. Latent Dirichlet Allocation [7] is known to perform well in text classification tasks. Biro et al. did a lot of research on modifying the LDA model to suit Web spam detection. They developed the multi-corpus LDA [8] and linked LDA [9] models. The former builds separate LDA models for spam and ham and uses topic weights as classification features. The later incorporates the link data into LDA model to spam classification.

Web spam is also aimed at web graph features used by search engines so many researchers focused on detecting link spam. Techniques like TrustRank [10] minimize the impact spam pages in ranking. Much attention has been focused on fighting link farms – web graph structures designed to accumulate PageRank and affect other pages rankings [11]. Finally more and more researchers combine the link and content data to improve classification results [12, 9]. In this work we didn't use any link spam detection techniques as we focused on content spam.

Fetterly et. al. proposed using duplicates analysis to detect web spam [13]. They measured phrase-level duplication of content across the web and found that spam tends to have greater number of popular shingles per document.

2. Understanding Content Spam

We believe that tackling Web spam is impossible without understanding how it works. Content spamming is aimed at text relevance algorithms, such as BM25 and tf.idf [1]. These algorithms are particularly vulnerable to content spam as there is a strong correlation between document relevance and amount of query terms found in the text.

Content spam is often used in doorways – pages and sites designed specifically to attract and redirect traffic. Doorways are only efficient if they reach the top of search results. Spammers prefer to generate thousands of doorway pages, each optimized for a specific query, to maximize amount of traffic collected.

This leads to several requirements that content spam must satisfy to be efficient:

- It must be generated in thousands of pages;
- Each page must maximize text relevance for some search query.
- Thus spammers have very little options of generating content for their doorways:
- They may generate content automatically;
- They may duplicate texts from other web sites;
- Or they may use a combination of both techniques.

Automatic text generation is a difficult task that does not have a satisfactory solution yet. Natural texts have multiple levels of consistency that are extremely hard to emulate all at once. In text generation tasks such as automatic document

summarization researchers distinguish multiple qualities of natural texts. Experiments show that even specialized text generation algorithms score low in most of these measures [14].

The levels of consistency include local coherence, style and authorship consistency, topical consistency, logical structure of the document etc. In this setting the uniqueness of text is just another type of constraint that is inherent for natural texts. Our approach is based on controlling as much natural text constraints as possible, making it harder for spammers to conceal low quality content.

There is a wide range of text generation techniques that generate locally coherent yet unreadable texts. Techniques like Markovian text generators are often used by web spammers to generate unique texts in great numbers. We were especially interested in designing text quality analyzer that would detect such advanced types of web spam.

3. Content Spam Detection Framework

Our work was based on assumption that spammers cannot emulate all aspects of natural texts. Our goal was to address as many domains of consistency as possible, by using various features. We measured multiple aspects of text quality and used supervised learning to combine them into content spam classifier. Despite a popular trend of combining link and content detection methods we focused solely on content analysis.

The basic natural language characteristics such as readability and POS ratios are overviewed in Section 3.1. The novel part of our spam detection framework is a set of text diversity features. We designed a range diversity features based on frequency rank distributions for different aspects of text diversity. The description and analysis of these features are provided in Section 3.2. Topical classification and topical diversity features based on LDA statistical model are presented in Section 3.3.

All statistics on described features were collected on WEBSPAM-UK2007 dataset [15]. The spam prevalence histograms provided in this section were generated on the set of 3995 labeled hosts from the training part of the dataset.

3.1. Statistical Features

The benefit of using wide range of linguistic features has been shown before by Piskorski et. al [6]. These features are commonly used in stylometry and authorship identification. We used POS tagger to tag every word in the dataset. We also substantially elaborated linguistic features by implementing a set of style-related diversity features that are described in Section 3.2.

In order to extract maximum information from POS tagging we calculated ratios of different parts of speech in words and ratios of different grammatical categories:

- POS ratios:
    - Adjectives;

    - Nouns;
    - Pronouns;
    - Verbs;
    - Numerals;
    - Particles;
    - Conjunctions;
    - Articles;
- Grammatical categories:
    - Number;
    - Tense;
    - Aspect;
    - Mood.

Combinations of different parts of speech and categories resulted in 82 distinct grammatical forms. We calculated the ratio of each grammatical form:

$$Ratio(form) = \frac{\#\, form\_occurences}{\#\, words}.$$

We also measured ratios of grammatical categories for specific parts of speech, e.g. ratio of verbs in past tense compared to all verbs:

$$Ratio_{verbs}(past\_tense) = \frac{\#\, verb\_in\_past\_tense}{\#\, verbs}.$$

As a result, we used a total of 145 POS-related statistical features.

Another domain we took features from was text readability research. Readability metrics were developed for military and educational purposes to measure how hard the text is to understand. Such features are helpful as automatically generated texts are usually unreadable. Some readability features have already been investigated by Ntoulas et. al. [5]. We implemented a set of readability features:

- Average word length;
- Average sentence length;
- Average number of punctuation symbols per sentence;
- Ratio of words longer than 7 symbols;
- Ratio of words shorter than 3 symbols;
- Maximum sentence length;
- Minimum sentence length.

The set of 152 statistical features described above allows detecting simple anomalies in text, such as query dumping, but it is still inadequate to fight advanced types of spam.

3.2. Text Diversity Features

Many researchers noticed that entropy and compressibility distinguishes content spam from normal texts [5]. We argue that this trait stems from auto generated nature of content spam. Currently no text generation algorithm can repeat the variety of natural language.

Some diversity-related features are easily faked by spammers. It is not uncommon for content spammers to use garbage text to decrease compressibility of texts in attempt to foil spam detection algorithms. To overcome these limitations we propose measuring variety of content in multiple aspects.

3.2.1. Character-Level Diversity

Compressibility is a well-known text variety feature. This characteristic has been used in both e-mail [16] and web spam detection [5]. Some content spamming techniques such as keyword stuffing produce texts with large number of repetitions. We use gzip and bz2 compression algorithms to measure compressibility of a document.

3.2.2. Term-Level Diversity

Compressibility is known to work well, when repeated keywords are located nearby in text. Spammers often dilute normal texts with keywords, thus making them harder to detect. Such subtle statistical violations can be detected by analyzing word frequency distributions.

Words in natural texts are known to obey power law frequency distributions. The most notable is Zipf law [17] that states that the frequency of any term is inversely proportional to its rank. Given a word *w*, with a frequency rank of *rank(w)*, its frequency may be estimated using the following formula:

$$freq(w) \approx \frac{const}{rank(w)^s}.$$

Parameter *s* characterizes variety of words in the given corpus of texts. We will refer to this value as to uniformity of terms. Greater uniformity leads to greater frequency of the most probable words, and lower frequencies of other words. The easiest way to calculate uniformity for a document is to convert the Zipf law to logarithmic scale:

$$\log(freq(w)) \approx s \log(rank(w)) + const.$$

Using this equation uniformity can be estimated using linear least squares. Let n be the number of different words in text, then:

$$f_w = \log(freq(w));$$
$$r_w = \log(rank(w));$$
(*)

$$s = -\frac{n\sum_w r_w f_w - \sum_w r_w \sum_w f_w}{n\sum_w (r_w)^2 - \left(\sum_w r_w\right)^2}.$$

We estimated terms uniformity to detect texts that contain multiple repeating keywords. In order to reduce the effect of stopwords we also calculated uniformity for nouns.

We also used a simpler approximation of term-level diversity by calculating the average number of terms that are repeated in neighbor sentences.

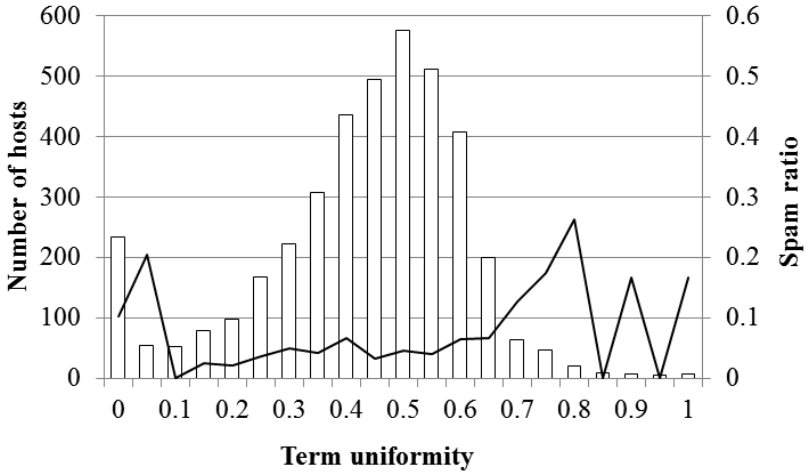


Figure 1. Prevalence of spam relative to term uniformity

The prevalence of spam relative to term uniformity is shown in Figure 1. In this figure the horizontal axis corresponds to different levels of term uniformity. The white bars correspond to number of hosts from the WEBSPAM-UK2007 training set with a given level of term uniformity and the black line corresponds to ratio of spam among those hosts. The figure shows that content spam tends to have greater uniformity, as spammers often repeat search keywords.

3.2.3. Sentence Structure Diversity

Most of content spam generation techniques produce new unique texts from a set of natural samples. Spammers may use Markovian text generator, which is trained on a set of natural documents, or they may simply take sentences from different texts, to form a single page content. These techniques often yield locally coherent texts that are hard to detect. To fight these types of spam we developed a set of features to measure the diversity of styles used in text.

We elaborated POS features described in Section 3.1 by adding a wide range of linguistic diversity features to detect style anomalies in texts. For each one of 145 POS ratio features we calculated its variance across sentences of text. A distribution of variances of adjectives ratio is shown in Figure 2, similar distributions work for other parts of speech and different grammatical categories. The graph confirms our hypotheses that content spam tends to mix styles from different texts, resulting in higher variances.

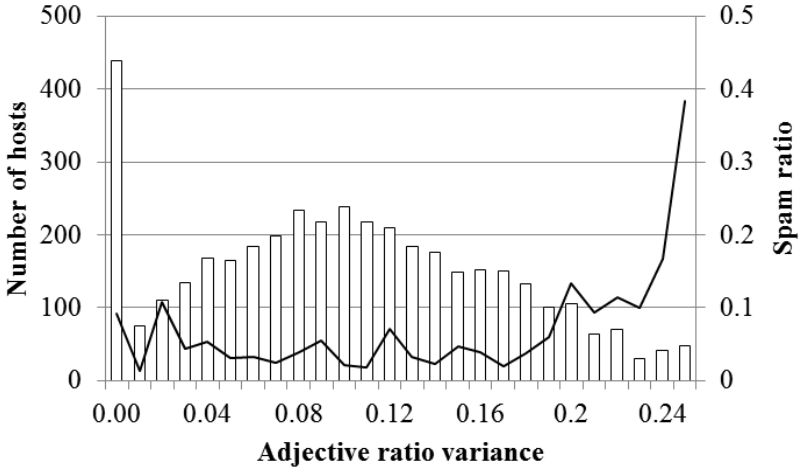


Figure 2. Prevalence of spam relative to adjective ratio variance across sentences

3.3. Topical Analysis

Web spam has a tendency to belong to several popular topics, like insurance, or pornography. We used topical features for two purposes. Firstly we used Latent Dirichlet Allocation (LDA) to measure the weights of different topics in texts and used these weights as classification features. Secondly we analyzed the frequency rank distributions for these weights in order to detect topical structure anomalies.

3.3.1. LDA

We decided to implement a set of topical classification features using Latent Dirichlet Allocation [7]. LDA is a fully generative probabilistic model for texts. LDA assumes that each document is generated by a mixture of topics. The weights of these topics can be used for topical classification. Most importantly LDA weights were used to measure topical diversity of texts. LDA-based topical diversity features are described in Section 3.3.2.

LDA has well-established parameter estimation and inference procedures, based on Monte Carlo Markov chains [18]. We used GibbsLDA++ library [19] that implements Gibbs Sampling algorithm for inference and parameter estimation. We trained LDA model on 20K random documents from WEBSPAM-UK2007 dataset, using 100 topics and $\alpha = 0.5, \beta = 0.01$ for hyper-parameters.

We could have used tf.idf for topical classification, but LDA also served as a dimensionality reduction algorithm. As a result we mapped every document in 100-dimension topic space, instead of using high-dimensional term vector space. The weights of different topics served as features in classification.

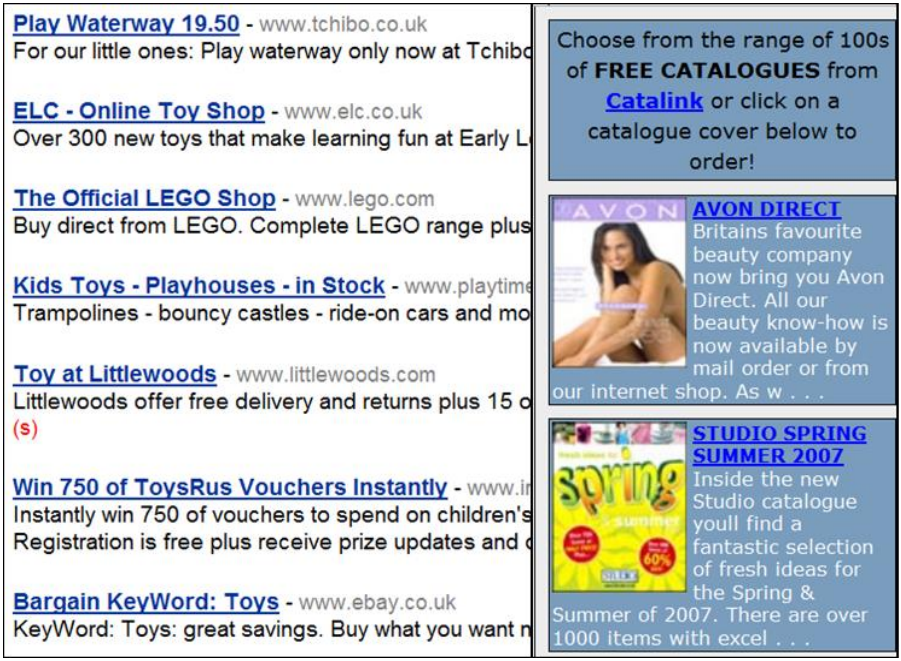


Figure 3a. Sample spam page with low topical uniformity (http://www.harrogate-toy-xmas-fair.co.uk/). The page consists of excerpts from different sources.

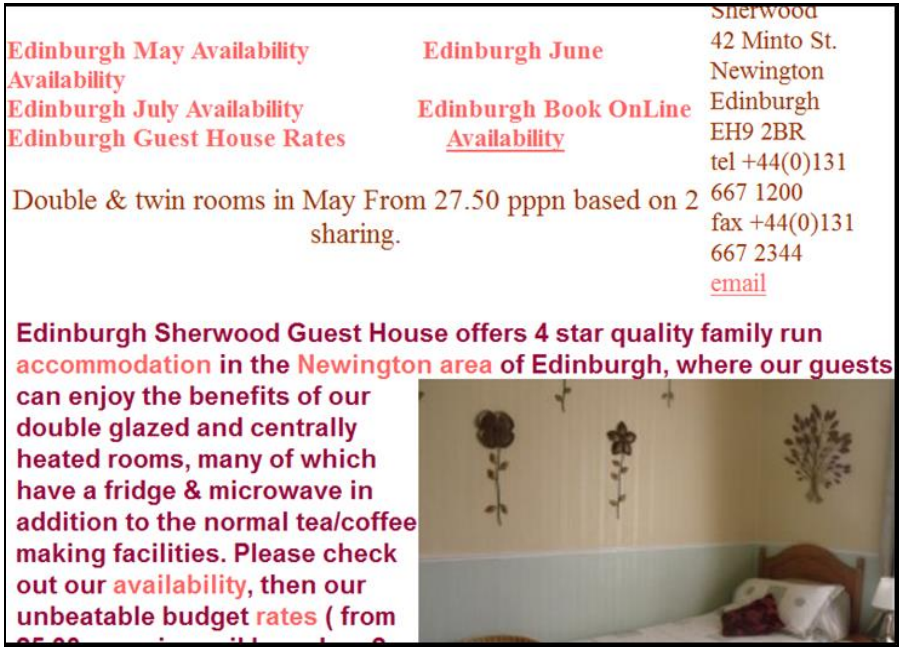


Figure 3b. Sample spam page with high topical uniformity (http://www.sherwoodguesthouseedinburgh.co.uk/). Notice keywords in the top and side of the page and highlighted keywords in text.

3.3.2. Topical Diversity

The analysis of LDA topic weights showed that these weights also have a power law distribution. Figure 5 shows the weights distribution for several samples of spam and non-spam hosts. Topical distributions are correlated with term frequency distributions, but have an advantage over them. LDA accounts for correlated terms thus a single LDA topic usually covers a whole set of terms that often co-occur. This ensures that synonyms and similar terms are counted together, and leaves spammers less chances to affect the feature.

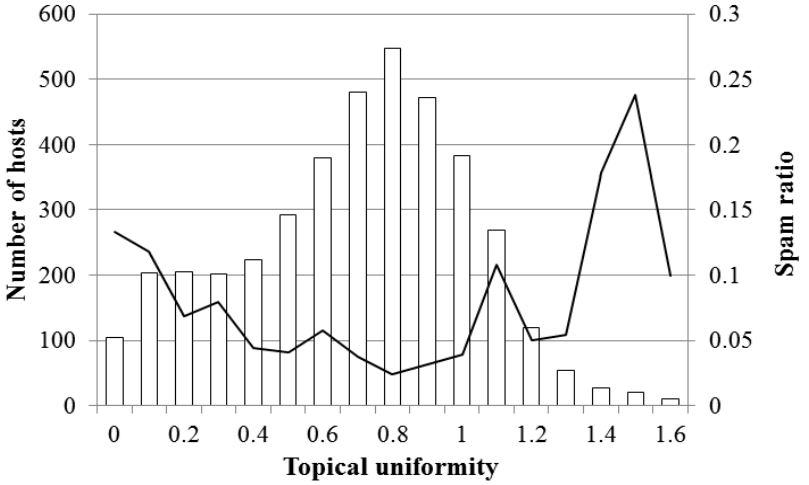


Figure 4. Prevalence of spam relative to topical uniformity

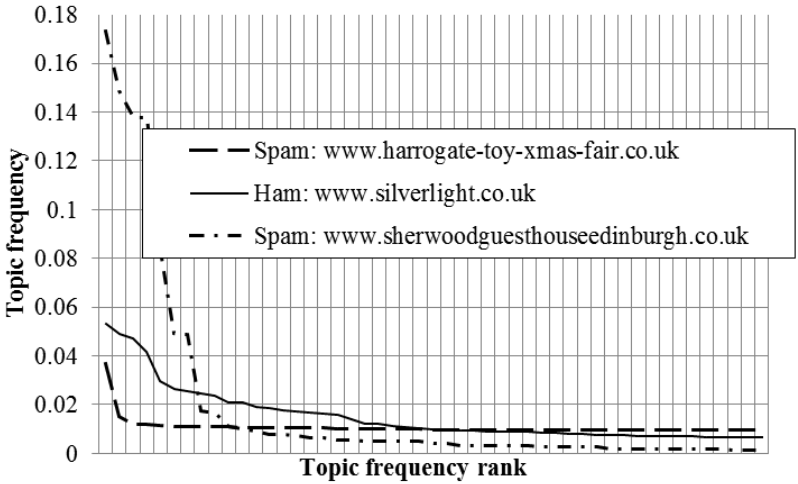


Figure 5. Topical frequency distributions for different types of spam

For each document we estimated the uniformity of frequency rank distributions of the LDA weights using the formula (*) using topic frequencies instead on word frequencies. The prevalence of spam for different levels of topical uniformity is

shown in Figure 4. The probability of spam is greater for hosts with both high and low uniformity. These two zones account for different types of content spam.

Hosts with higher uniformity usually contain texts stuffed with keywords (e.g. www.sherwoodguesthouseedinburgh.co.uk, Figure 3a). The other group of spam hosts has very low topical entropy. Texts from this group usually contain search results or sentences taken from multiple other texts (e.g. www.harrogate-toy-xmas-fair.co.uk, Figure 3b). Topical distributions for these hosts are provided in Figure 5.

We also researched an alternative approach to measuring the topical diversity. Being a probabilistic model LDA only generates the most probable topics weights distribution for the text. In order to detect spam content we calculated the probability of a document having uniform topical distribution (all topics having the same weight). Considering this as a statistical hypothesis the Pearson's chi-squared statistics can be used to check it. Let N be the number of topics, then:

$$\chi^2 = N \sum_{topic} \frac{\left(1/N - weight_{topic}\right)^2}{1/N}.$$

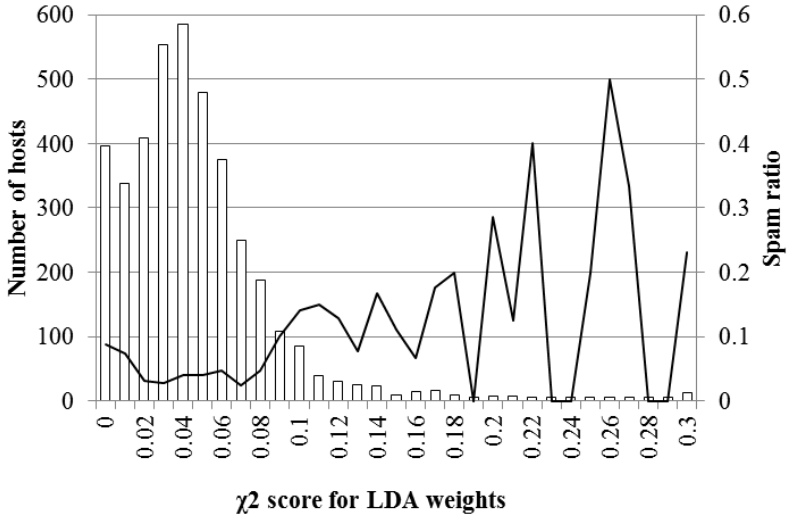


Figure 6. Prevalence of spam relative to chi-squared topical score

We used this statistics as a classification feature. The prevalence of spam depending on χ^2 score is provided in Figure 6. The higher χ^2 score means that the hosts have lower probability of having uniform topical distribution. The spam probability for

hosts with χ^2 score greater than 0.1 is substantially higher than average spam probability.

3.4. Machine Learning

Using LDA as a dimensionality reduction algorithm allowed us to use algorithms designed for dense data, without implementing complex ensembles of classifiers.

We used logistic regression with L2 regularization. We used a fixed regularization parameter value of 0.25. It generates a relatively simple linear classifier with regression coefficients which can be interpreted as contribution of features to the classification task. Some features such as topical uniformity show non-linear behavior that cannot be accounted for using a linear classification formula.

3.5. Complexity estimation

To prove that the proposed algorithm can be used in web-scale spam detection tasks we also estimated the complexity of the proposed algorithm during the classification phase. The algorithm can be loosely split in 3 parts:

- Statistical features calculation;
- Topical diversity estimation based on LDA;
- Machine learning;

The first phase includes POS tagging and compressibility analysis. We used simple POS taggers that analyze single words and do not take previous words in account. The complexity of the POS tagging process in on the order of document's length $O(|d|)$.

The first phase also includes term-level diversity calculation that implies words being sorted by their frequencies. So the complexity of the diversity calculation is on the order of $O(|d|\log(|d|))$.

The second part of the algorithm starts with LDA inference. Gibbs sampling is used for inference and complexity of each iteration is proportional to the length of the document and number of topics used [18]. Instead of running Gibbs sampling until convergence we used fixed number of iterations that suited our purposed well. So the complexity of the Gibbs sampling phase was $O(|d|)$.

The calculation of topical diversity after the topic weights were estimated depends only on number of topics and its complexity can assumed constant.

Finally in machine learning phase we used constant number of features in a linear classification formula and its complexity is also constant. In whole the complexity of the proposed classification algorithm is $O(|d|\log(|d|))$, where $|d|$ is the length of the classified document.

4. Experiments

The evaluation of the proposed framework consisted of two experiments. First we tested the ability of our approach to detect synthetic automatically generated texts. The second experiment was dedicated to measuring the benefit of the proposed features.

Finally we tested the framework in the Web Spam Challenge [4] settings.

4.1. Synthetic Text Experiment

First we tested the capability of the described features to detect automatically generated low quality texts. We created a set of synthetic texts using a Markovian text generator. The generator was trained on a collection of 20K random documents from the WEBSPAM-UK2007 dataset. Here is a sample of such synthetic text generated from this article:

> *Tf.idf and other term-weighting approaches are often used by web spammers to generate thousands of doorway pages, each optimized for a specific query, to maximize amount of text, and ratio of verbs in past tense compared to all verbs: We used POS tagger to tag every word in the dataset.*

Such texts consist of locally coherent pieces collected from other documents. We used 10K of synthetic documents and random 10K documents from the WEBSPAM-UK2007 dataset as a training set. The test set for the experiment was created in a similar fashion. We used two Markov chains of order 2 (MC2) and 3 (MC3) to measure the effects of this parameter on classification.

	Precision	Recall	F-measure
MC2, SF + LDA	96.19%	96.11%	96.15%
MC3, SF + LDA	94.08%	92.29%	93.18%
MC2, All	98.37%	97.93%	**98.14%**
MC3, All	97.72%	97.09%	**97.40%**

Table 1. Precision, Recall, and F-measure for synthetic text detection experiment

In order to measure the effect of the proposed features we made two runs of the experiment. First we used only statistical features and LDA weights as a baseline experiment (SF+LDA). During the second run we used all available features including diversity features (All).

Table 1 contains results of the experiment. High F-measure rate suggests that described features are adequate for detecting such advanced types of content spam.

Increase in Markov chain order causes the generator repeat larger pieces of original documents. This reduces detection rate, but increases the amount of non-unique content in such texts. The results also show that the proposed diversity features substantially improve the classifier. In fact they reduce the number of false positives and false negatives in half.

4.2. Feature Analysis

The purpose of the second experiment was to estimate power of each of the 334 features. The settings of this experiment were similar to the synthetic text detection experiment. We used 20K documents from WEBSPAM-UK2007 dataset as a non-spam sample and generated 20K documents using a Markov chain text generator with the chain length of 2. These sets were then split evenly in training and testing datasets.

For each feature we trained a separate classifier. Each classifier was trained on a single feature. The classification F-measure of the given classifier can be viewed as a measure of usefulness of the corresponding feature. Table 2 contains the 20 most useful features for synthetic texts classification task.

The results of the experiment show that diversity features are paramount for detecting Markov chain generated texts. The proposed topical diversity features score best on this metric, along with text compressibility. Other diversity features also can be seen among the top-20.

Feature	F-measure	Feature type
Topical uniformity	91.23%	Diversity
Gzip compression rate	89.70%	Diversity
χ^2 score for LDA weights	87.03%	Diversity
bz2 compression rate	85.04%	Diversity
Term uniformity	81.28%	Diversity
Average number of words repeated in neighbor sentences	79.60%	Diversity
Verbs in past tense ratio	74.49%	Statistical
Number of expressive punctuation marks per sentence	73.54%	Statistical
Verbs in past tense variance	73.34%	Diversity

Feature	F-measure	Feature type
Modal verbs variance	72.88%	Diversity
Fraction of sentences with several verbs	71.27%	Statistical
Personal pronouns ratio	71.13%	Statistical
Proper nouns ratio	71.06%	Statistical
Possessive endings variance	70.66%	Diversity
Words with one syllable ratio	70.63%	Statistical
Modal verbs ratio	70.59%	Statistical
Words with two syllables ratio	70.56%	Statistical
Cardinal numbers variance	70.55%	Diversity
Cardinal numbers ratio	70.06%	Statistical
Determiners ratio	69.82%	Statistical

4.3. Webspam-UK2007 Experiment

In this experiment we followed the evaluation protocol of the Web Spam Challenge [4]. Using this evaluation procedure we could compare our results with other studies. The Web Spam Challenge 2008 was held on a WEBSPAM-UK2007 dataset [15]. The training and testing labels are also defined in the dataset. The official quality measure for the challenge was the Area under ROC Curve (AUC ROC). We also calculated optimal F-measure for the classification task.

We compared against best results on this dataset. The winners of the 2008 Web Spam Challenge Geng et. al. [20] used pre-computed features and advanced bagging strategies to reach the AUC of 0.85. Biro et. al. [9] used linked LDA model to combine link and content features yielding the AUC score of 0.854. Dai et. al. [21] used temporal features and achieved classification F-measure of 0.521.

We combined the features into four groups:

- SF – statistical features (Section 3.1);
- DF – various text diversity features (Section 3.2, Section 3.3.2);
- LDA – the Latent Dirichlet Allocation topic weights (Section 3.3.1);

Features	AUC	F1
Geng et. al.	0.85	--
Biro et. al.	0.854	--
SF	0.746	0.284
DF	0.744	0.323
LDA	0.845	0.442
SF+DF	0.777	0.348
SF+LDA	0.867	0.433
DF+LDA	0.864	0.448
All (SF+DF+LDA)	**0.871**	**0.458**

SF – statistical features;

LDA – Latent Dirichlet Allocation topic weights;

DF – diversity features;

Table 3. Results for WEBSPAM-UK2007 experiment

The results of classification using various groups of features and machine learning algorithms are provided in Table 3. Using the logistic regression the best result of 0.871 AUC is achieved when combining all features. Our approach substantially improves over the nearest result of 0.854 AUC. The results show that topical classification features (LDA) are still crucial to web spam detection, but statistical features (SF) and diversity features (DF) improve the results substantially.

5. Conclusion and Future Work

The results of our research show that advanced content features are useful for content spam detection. We analyzed different aspects of natural texts and produced a set of features to cover as many aspects as possible. The resulting spam classifier performed well on both synthetic and real-life tasks.

The proposed approach is based solely on content analysis and doesn't take link data into account. Combining the proposed method with existing link-spam detection techniques is likely to improve results. Another possible extension is to use the diversity measures and rank distributions on link data to detect unnatural link structures.

Web spam is primarily an economic phenomenon and the amount of spam depends on efficiency and costs of different spam generation techniques. We hope that multiple diversity features described in this work can substantially decrease the

efficiency of automatically generated content spam. There are many properties of natural texts that are not covered by this article. We plan to continue research on various aspects of natural texts that are hard to reproduce.

References

[1] Z. Gyongyi and H. Garcia-Molina. Web Spam Taxonomy. In 1st International Workshop on Adversarial Information Retrieval on the Web, May 2005.

[2] C. Castillo, D. Donato, L. Becchetti, P. Boldi, S. Leonardi, M. Santini, S. Vigna, A reference collection for web spam, ACM SIGIR Forum, v.40 n.2, p.11-24, December 2006.

[3] M. Henzinger, R. Motwani, C. Silverstein. Challenges in Web Search Engines. SIGIR Forum 36(2), 2002.

[4] Web Spam Challenge. http://webspam.lip6.fr/wiki/pmwiki.php, 2008.

[5] A. Ntoulas , M. Najork , M. Manasse , D. Fetterly, Detecting spam web pages through content analysis, Proceedings of the 15th international conference on World Wide Web, May 23-26, 2006, Edinburgh, Scotland.

[6] J. Piskorski , M. Sydow , D. Weiss, Exploring linguistic features for web spam detection: a preliminary study, Proceedings of the 4th international workshop on Adversarial information retrieval on the web, April 22, 2008, Beijing, China.

[7] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet allocation. Journal of Machine Learning Research, 3(5):993–1022, 2003.

[8] I. Biro, J. Szabo, A. A. Benczur, Latent Dirichlet allocation in web spam filtering, Proceedings of the 4th International Workshop on Adversarial Information Retrieval on the Web, April 22, 2008, Beijing, China.

[9] I. Biro, D. Siklosi, J. Szabo, A. A. Benczur, Linked latent Dirichlet allocation in web spam filtering, Proceedings of the 5th International Workshop on Adversarial Information Retrieval on the Web, April 21-21, 2009, Madrid, Spain.

[10] Z. Gyongyi, H. Garcia-Molina and J. Pedersen. Combating Web Spam with TrustRank. In 30th International Conference on Very Large Data Bases, Aug. 2004.

[11] B. Wu, B. D. Davison. Identifying link farm spam pages. Special interest tracks and posters of the 14th international conference on World Wide Web - WWW '05. 2005.

[12] J. Abernethy, O. Chapelle, and C. Castillo. WITCH: A New Approach to Web Spam Detection. In Proceedings of the 4th International Workshop on Adversarial Information Retrieval on the Web (AIRWeb), 2008.

[13] D. Fetterly, M. Manasse, and M. Najork. Detecting phrase-level duplication on the world wide web. In Proceedings of the 28th ACM International Conference on Research and Development in Information Retrieval (SIGIR), Salvador, Brazil, 2005.

[14] H. Dang. Overview of DUC 2006. Proceedings of the Document Understanding. 2006.

[15] Yahoo! Research: "Web Spam Collections". http://barcelona.research.yahoo.net/webspam/datasets/ Crawled by the Laboratory of Web Algorithmics, University of Milan, http://law.dsi.unimi.it/. URLs retrieved May 2007.

[16] A. Bratko, G. V. Cormack, B. Filipic, T. R. Lynam, and B. Zupan. Spam filtering using statistical data compression models. Journal of Machine Learning Research, 7(Dec):2673–2698, 2006.

[17] G. Zipf, Selective Studies and the Principle of Relative Frequency in Language (Cambridge, Mass, 1932).

[18] C. Andrieu, N. de Freitas, A. Doucet, M. Jordan, An introduction to MCMC for machine learning. Machine Learning, 50: 5–43, 2003.
[19] X.-H. Phan, C.-T. Nguyen, Gibbs LDA++: A C/C++ Implementation of Latent Dirichlet Allocation (LDA) using Gibbs Sampling for Parameter Estimation and Inference. http://gibbslda.sourceforge.net/, 2008.
[20] G. Geng, X. Jin, C.-H. Wang. CASIA at Web Spam Challenge 2008 Track III. In Proceedings of the 4th International Workshop on Adversarial Information Retrieval on the Web (AIRWeb), 2008.
[21] N. Dai, B.D. Davison, X. Qi. Looking into the past to better classify web spam. Proceedings of the 5th International Workshop on Adversarial Information Retrieval on the Web - AIRWeb '09. 2009.

Обнаружение поискового спама в Вебе на основе анализа разнообразия текстов

Павлов А.С., Добров Б.В.
pavvloff@yandex.ru, dobroff@mail.cir.ru

Аннотация. Поисковый спам считается одной из основных угроз современным поисковым системам. Спамеры используют разнообразные методы порождения текстов, известные как текстовый спам, чтобы наполнить выдачу поисковых систем низкокачественными страницами. Методы борьбы с текстовым спамом должны основываться на большом количестве текстовых характеристик. В данной статье предлагается набор характеристик текстового разнообразия, основанных на ранговых распределениях для слов и тематик. Предложенные характеристики объединяются с другими факторами, в результате чего получается классификатор поискового спама, превосходящий известные аналоги.

Ключевые слова: поисковый спам; анализ признаков; тематическое разнообразие.

Extracting Objects and Their Attributes from Tables in Text Documents

Nikita Astrakhantsev
astrakhantsev@ispras.ru

Abstract. Extracting information from tables is an important and rather complex part of information retrieval.
For the task of objects extraction from HTML tables we introduce the following methods: determining table orientation, processing of aggregating objects (like Total) and scattered headers (super row labels, subheaders).

Keywords. Information extraction; information retrieval; natural language processing; table processing; table extraction; semi-structured information extraction; html; wiki markup.

1. Introduction

Significant amount of text information has relational structure, which is often represented in a table view. Since this view is designed for humans and contains many ambiguous elements, it follows that automatic processing of tables is rather complex.

For example, table 1 contains scattered header, complex hierarchical header, special objects, and other elements. They play particular roles in the table and, thus, should be treated in a special way. These elements are described in the rest of the paper.
Another non-trivial task is to determine table orientation: it can be horizontal (row wise, table 1) or vertical (column wise, table 2).

We consider tables in structured formats such as HTML and Wiki markup[1]. The main goal of our table processing is to extract objects as collections of attribute-value pairs. Thereupon we focus on determining table orientation and understanding the role of each element in the table.

[1] Wiki markup is a lightweight markup language used to write pages in wiki websites, such as Wikipedia, and is a simplified alternative/intermediate to HTML. http://en.wikipedia.org/wiki/Wiki_markup

Model	Version	Year	Power		Production
			hp	kW	
Sports cars					
SVT Lightning	9	1993	240	179	5,276
		1994			
		1995			2,280
		Total			7,556
	10	1999	360	268	4,000
		2000			4,966
		2001	380	283	6,381
		Total			28,124
		Total			35,680

Scattered header

Special objects

Complex header

Empty cell

Table 1

	CS100	CS100ER	CS300
Seat Pitch	32 in (81 cm)		
Seat Width	19 in (48 cm)		
Flight crew	2 (pilot, co-pilot)		
Length	34.9 m (115 ft)		38.0 m (124.7 ft)
Wingspan	35.1 m (115 ft)		
Wing Area (net)	112.3 m^2 (1,209 sq ft)		
Fuselage max diameter	3.7 m (12 ft)		
Cargo Volume	23.2 m^3 (820 cu ft)		30 m^3 (1,100 cu ft)
Max range	4,074 km (2,200 nmi)	5,463 km (2,950 nmi)	4,074 km (2,200 nmi)

	CS100	CS100ER	CS300
Typical cruise speed	Mach 0.78 (828 km/h, 447 kn, 514 mph)		
Landing field length	1,350 m (4,430 ft)		1,448 m (4,751 ft)

Table 2

2. Related work

Silva et al [1] distinguish five tasks of extraction information from table in their detailed survey:

1. **Location**: differentiating the table from other text elements such as body text, titles, lists, etc.
2. **Segmentation**: identifying table cells, rows, and columns and their relative positions.
3. **Functional analysis**: classifying a table area (cell, column, row) to data or attribute area.
4. **Structural analysis**: connecting each data cell to all characterizing attribute cells.
5. **Interpretation**: understanding and further using extracted information.

First of them occurs mostly in plain text and image formats. The last task depends on kind of table usage: generating ontology from tables [2], mapping extracted information to predefined scheme [3], and so on.

Our work is devoted to the three last tasks while structural analysis is a principal one.

2.1. Plain text format

Early works deal with tables in plain text, usually in ASCII format [4-7]. The main problem here is to detect table, to recognize table delimiters (spaces, tabs, special characters like hyphens, etc), and thereby to understand table structure.

Rus and Summers [4] consider a text block to be a table if it consists of columns separated by white space and if cells of such columns are lexically persistent. A similar approach is used in the work of Douglas et al. [5]; they group text blocks surrounded by white spaces and use heuristic to determine whether these blocks are parts of tables.

There are many works concerned plain text and most of them use approaches inapplicable for HTML tables. However there are some useful ideas, which were explored in these works and inspired by linguistic characteristics of table content.

For example, similarity/cohesion among different table elements (cells/lines/columns) became a common feature in future works including HTML tables oriented ones. Hurst and Douglas [6] suggest explicit string-based formulas for computing cell values cohesion.

Pinto et al. [7] present Conditional random field algorithm for classification of each row in ASCII table to one of 12 classes such as “data row”, “section header”, “super header”, etc.

2.2. Image format

The majority of these works are devoted to performing location and segmentation of a table (Tupaj et al. [8], Wang et al. [9, 10], inter alia).

At a distance, Gatterbauer et al. [11] process HTML tables treating them as images. More precisely, they distinguish two representations of web pages: DOM tree representation and visual box (topological) representation. Our work is based on the first representation, while their approach is based on the second one, and thereby it cannot be applied to our work.

2.3. HTML format

A main source of HTML tables is Web pages, but many Web-tables are created just for layout, not for representing relational information. Wang and Hu [12] call such tables *non-genuine*. They suggest machine learning classifiers (SVM and decision tree) to determine whether the table is genuine or non-genuine. Features are divided into three types: length consistency of the cell contents, type of cell content, and word group.

Chen et al. [13] apply string and content type similarity to this task. They also use cell similarity for determining table orientation.

Yoshida et al. [14] suggest Expectation Maximization algorithm for classifying each table into one of 9 predefined types. Ontological knowledge are used as a parameter for the model, e.g. "Name" and "Birthday" are more often attributes than values.

Cafarella et al. [15] process the corpus of 14 billion Web tables. They introduce a tool for synonymic attributes searching (Attribute Correlation Statistics), which can be useful in the task of attribute/value classifying. Also the statistics gathered by authors corroborates the assumption that there is a small set of schemas that most tables in the world conform to.

3. Table orientation

What is an object in a table? Often the answer is obvious. For example, in table 2 objects are two airbuses: A310-200 and A310-200F. Table 1 is a little bit vaguer. It contains SVT vehicles manufactured in different years. Sometimes tables with nondefinable objects are found, e.g. multiplication table or table 3.

NFPA 704

Fire diamond for aluminium shot

Table 3

As a rough definition, table object is a real world entity, whose attributes are set in the table. We assume that either row or column represents an object. So, table has either horizontal or vertical orientation.

We use 2 machine learning algorithms with the same features to determine table orientation: decision tree and naïve Bayes. All features have common nature: some function is computed on horizontal table as well as on vertical one and the difference between obtained values is found.

First feature is the difference in header depth. The function is very simple; it returns the depth of table header hierarchy. E.g. the header depth for horizontal orientation of table 1 is 2, due to *Power*, *hp*, and *kW* cells; the header depth for the vertical table is 1. The header depths for both orientations of table 2 are 1. The motivation is following: orientation with deeper header is more likely to be correct.

Second feature is difference in maximum cell cohesion. Cell cohesion is an average string similarity of all cells in a row or in a column. Average string similarity is computed by summation of all pairwise string similarities/metrics of cells and normalization (dividing by square of total count of cells). Maximum cell cohesion for the horizontal table is simply a maximum of cell cohesion in all rows; the same stands for the vertical one with replacement of rows by columns.

Third feature is the difference in average cell cohesion. The only distinction from the previous feature is that all cell cohesions are summed and then divided by total count of rows or columns.

After experiments with a small set of Wikipedia tables (about 70) we found that the most valuable feature is the difference in average cell cohesion. All other features aren't presented in decision tree without overlearning (see figure 1).

Also we experimented with introducing the third type of tables, which contain no object, but effectiveness decreased dramatically. This can be easy explained because even human can't label classify some table into the third type with confidence. So we don't take into account tables without objects.

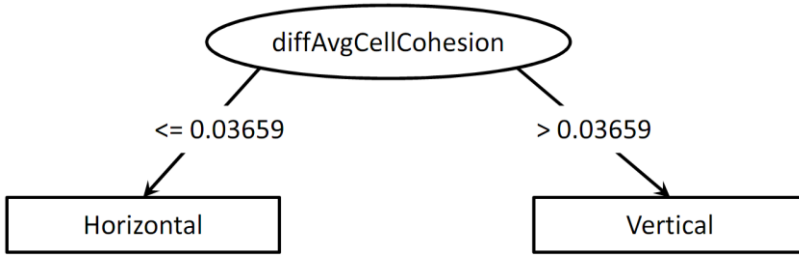


Figure 1

4. Aggregating objects processing

Some tables contain *aggregating objects*, which actually store information about other objects from the same table. For example, in table 4 the last row isn't an ordinary entity and should be processed in special way.

System	Affiliates	%
FONASA	12,248,257	72.69
ISAPRE	2,780,396	16.50
Total pop.	**16,849,081**	**100.00**

Table 4

Recognizing such objects by only presence of a keyword (e.g. *Total*) isn't efficient because of cells like "Total depravity, with prevenient grace, does not preclude free will". So we collected statistics on thousands of Wikipedia tables and developed the heuristic for determining aggregating objects on basis of the next features:

- Type of keyword in the cell content.
- Number of words in the cell.
- *Position* of the cell (sequence number of its column).
- *Decoration* (bold, uppercase, <th>-tag).

All keywords are split into 2 types: *the strong type* (**Total, Totals, Tot., Subtotal**) and *the weak type* (**All**, **Sum**).

The cells containing words from the weak type must satisfy next conditions:

- exactly 1 word

- decorated
- position is not greater than 2

The cells containing words from the strong type must satisfy next conditions:

- if there is 1 word - no condition
- if there are 2 words - the cell must be decorated
- if there are from 3 to 5 words - the cell must be decorated and its position must be not greater than 2

The heuristic was inspired by the following considerations. The aggregating cells store information about special objects, so they should be noticeable by human readers. If the cell string is long or if it contains common word like *all*, then it must have some decorations in order to attract attention.

In future we plan to develop machine learning approach with these heuristics as features.

5. Scattered header processing

Some tables contain special rows, or *scattered headers*, which add structural information and are not table objects. Example is the row *Sports car* from table 1. We called the object extracted from such scattered header row during the initial processing *the scattered header object*.

We extract scattered header objects only from the object set, which corresponds to the horizontal orientation of the table. Vertical objects aren't processed, because width of any table is limited and scattered headers are useless in vertical tables.

5.1. Scattered header recognizing

Deciding whether each row is a scattered header (SH) is based on the assumptions that only one cell in the row is nonempty and the row must stand out against other table. In addition, we don't consider empty rows and last rows.

We divide all SHs into three subclasses: *single-cell SH, middle SH, first-cell SH*.

1) **Single-cell SH** is a row with just one cell in a table where other rows have more than 1 cell (in HTML terms, a row with colspan greater than 1).

The example is given in table 5 (rows *Monohydric alcohols* and *Polyhydric alcohols*).

Chemical Formula	**IUPAC Name**	**Common Name**
Monohydric alcohols		
CH_3OH	Methanol	Wood alcohol
C_2H_5OH	Ethanol	Grain alcohol

$C_5H_{11}OH$	Pentanol	Amyl alcohol
Polyhydric alcohols		
$C_2H_4(OH)_2$	Ethane-1 ,2-diol	Ethylene glycol
$C_3H_5(OH)_3$	Propane-1 ,2,3-triol	Glycerin
$C_4H_6(OH)_4$	Butane-1 ,2,3,4-tetraol	Erythritol

Table 5

We consider such row to be an SH without other criteria.

2) **Middle SH** is a row with just one nonempty cell, located in the middle of the table.

The 7th row *Bonus track* of table 6 is a middle SH.

#	**Title**	**Songwriters**	**Length**
1.	"Mixing pot" ("Tacho")	Hermeto Pascoal	9:18
2.	"Slaves mass" ("Missa dos escravos")	Hermeto Pascoal	4:19
3.	"Little cry for him" ("Chorinho para ele")	Hermeto Pascoal	2:11
4.	"Cannon (Dedicated to Cannonball Adderley)"	Hermeto Pascoal	5:20
5.	"Just listen" ("Escuta meu piano")	Hermeto Pascoal	7:08
6.	"That waltz" ("Aquela valsa")	Hermeto Pascoal	2:46
	Bonus tracks		
7.	"Open field" ("Campo aberto")	Hermeto Pascoal	4:25
8.	"Pica pau (Take 1)"	Hermeto Pascoal	14:20

Table 6

We require the nonempty cell of such row to be decorated.

In addition, we check the table to be non-sparse. Sparse table is a table with many empty cells, e.g. table 7. It is evident that the heuristic for determining middle SH doesn't work correctly with sparse tables.

Km	Exit	Junctions	To	Remarks
		SMT Bandar Penawar	Sekolah Menengah Teknik Bandar	
		BANDAR PENAWAR	**SOUTH** Bandar Penawar	T-junctions
		SMS Kota Tinggi	Sekolah Menengah Sains Kota Tinggi	
		SMKA Bandar	Sekolah Menengah Kebangsaan Penawar	
0		SMK Bandar Penawar	Sekolah Menengah Kebangsaan	
		Desaru guardpost		
		DESARU	Desaru Impian Desaru Golf & Country Resort	

Table 7

So we check the rows near the concerned middle SH row (3 rows above and 3 below). If they have empty cells, then the row under review isn't a middle SH. Of course, the row with empty cells can be another middle SH; to address this issue we don't take such rows into account while checking their cells. But previous and next rows must differ from the concerned one, because scattered headers never have the same structure and content with another SH.

Thereby table 7 contains no SH.

3) **First-cell SH** is a row whose only nonempty cell is first.

For example, the third row *Cities (10 Largest)* of table 8 belongs to this type.

Census Metropolitan Areas:	**2006**	**2001**	**1996**
Calgary CMA	1,079,310	951,395	821,628
Edmonton CMA	1,034,945	937,845	862,597

Cities (10 Largest):			
Calgary	988,193	878,866	768,082
Edmonton	730,372	666,104	616,306
Red Deer	82,772	67,707	60,075
Strathcona County	82,511	71,986	64,176

Table 8

The criteria for this type are the same as for the previous type.

5.2. Scattered header processing

Scattered header object are removed from the original set before the aggregating objects processing. Therefore created aggregating objects have no information about scattered header objects.

We update attributes of all objects by adding the new field; currently its name is *Type*. Every table object located below the current scattered header and above the next scattered header (if it exists) is updated by adding a new value, which is the content of the corresponding scattered header.

6. Conclusions and future work

In this paper we concerned on the task of objects extraction from HTML tables, gave a short survey of occurring problems, and introduced methods (mostly heuristics) for their solving. Of course, there are many cases uncovered by this paper, e.g. accurate detection of table header, processing of non-aggregating special objects and so on. It's the scope of future research.

The most common usage of extracted objects is to map them to predefined relational scheme. Embley et al. [3] worked towards this task, but we believe that fully automatic processing will be ineffective. Gatterbauer et al. [11] make a similar note: “domain-independent table interpretation cannot result in unambiguously structured information because of existing inherent domain-specific ambiguities that can sometimes not even be resolved by humans”. Therefore, we consider the computer-aided way to be more promising for the task of objects mapping and, in general, for table interpretation.

References

[1] A.C. Silva, A.M. Jorge, L. Torg. Design of an end-to-end method to extract information from tables // International Journal of Document Analysis and Recognition. 2006. 8. N 2–3. P. 144–171.

[2] Y. A. Tijerino, D. W. Embley, D. W. Lonsdale,. Y. Ding, and G. Nagy. Towards ontology generation from tables. World Wide Web. 2005. 8. N 3. 261–285.
[3] D.W. Embley, C. Tao, S.W. Liddle. Automating the Extraction of Data from HTML Tables with Unknown Structure // Data & Knowledge Engineering. 2003. N 54. P. 3–28.
[4] D. Rus, K. Summers. Using white space for automated document structuring. Workshop on the Principles of Document Processing, 1994.
[5] S. Douglas, M. Hurst, D. Quinn. Using Natural Language Processing for Identifying and Interpreting tables in Plain Text. In: Fourth Symposium on Document Analysis and Information Retrieval, pp. 535–545, 1995.
[6] M. Hurst, S. Douglas. Layout and Language: Preliminary investigations in recognizing the structure of tables // Proceedings of International Conference on Document Analysis and Recognition. Washington, DC, USA: IEEE Computer Society, 1997. P. 1043–1047.
[7] D. Pinto, A. McCallum, X. Wei, W.B. Croft. Table Extraction Using Conditional Random Fields // Proceedings of the ACM SIGIR N 26. New York, USA: ACM New York, 2003. P. 235–242.
[8] S. Tupaj, Z. Shi, C.H. Chang, A. Hassan. Extracting tabular information from text files, EECS Department. Tufts University, 1996.
[9] Y. Wang, T.P. Ihsin, H. Robert. Improvements of zone content classification by using background analysis // Document Analysis Systems. 2000. N 4. P. 10–13.
[10] Y. Wang, T.P. Ihsin, H. Robert. Automatic ground truth generation and a background-analysis-based table structure extraction method // Proceedings of the Sixth International Conference on Document Analysis and Recognition. Washington, DC, USA: IEEE Computer Society, 2001. P. 528–532.
[11] W. Gatterbauer, P. Bohunsky, M. Herzog, B. Krüpl, B. Pollak. Towards domain-independent information extraction from web tables // Proceedings of the 16th WWW. New York, USA: ACM New York, 2007. P. 71–80.
[12] Y. Wang, J. Hu. A machine learning based approach for table detection on the web // Proceedings of the 11th WWW. New York, USA: ACM New York, 2002. P. 242–250.
[13] H.-H. Chen, S.-C. Tsai, S.-C., J.-H. Tsai. Mining tables from large scale HTML texts // 18th International Conference on Computational Linguistics. Saarbrücken, Germany: Morgan Kaufmann, 2000. P. 166–172.
[14] M. Yoshida, K. Torisawa, J. Tsujii. A method to integrate tables of the WorldWideWeb // Proceedings of the First International Workshop on Web Document Analysis. Seattle, USA: PRImA Press, 2001. P. 31–34.
[15] M.J. Cafarella, A. Halevy, Y. Zhang, D.Z. Wang, E. Wu. WebTables: Exploring the Power of Tables on the Web // ACM SIGMOD Record. 2008. N 37. P. 55–61.

Извлечение объектов и их атрибутов из таблиц текстовых документов

Никита Астраханцев
astrakhantsev@ispras.ru

Аннотация. Извлечение информации из таблиц является важной и достаточно сложной частью информационного поиска. В рамках задачи извлечения объектов из HTML-таблиц предлагаются методы, решающие следующие проблемы: определение ориентации таблицы, обработка агрегирующих объектов (таких как *Total*) и разрозненных заголовков (подзаголовков, перерезов).

Ключевые слова. Извлечение информации; информационный поиск; обработка естественного языка; обработка таблиц; извлечение таблиц; html; wiki markup.

WikifyMe: Creating Testbed for Wikifiers

Sergey Bartunov, Alexander Boldakov, Denis Turdakov
sbartunov@gmail.com, boldakov@gmail.com, turdakov@ispras.ru

Abstract. Finding relationships between words in text and articles from Wikipedia is an extremely popular task known as wikification. However there is still no gold standard corpus for wikifiers comparison. We present WikifyMe, the online tool for collaborative work on universal test collection which allows users to easily prepare tests for two most difficult problems in wikification: word-sense disambiguation and keyphrase extraction.

Keywords: "Wikipedia", natural languages processing, word sense disambiguation, keywords extraction, testing, collective work, crowdsourcing, Web tools, text preparatiion.

1. Introduction

Enrichment of text documents with links to Wikipedia's pages has become an extremely popular task. This task is called *wikification*. Wikification is necessary for intelligent systems that use knowledge extracted from Wikipedia for different purposes [5 ,8]. Showing wikified documents to reader of blogs or news feed is common as well [10 ,4].

Enrichment text with links to Wikipedia usually consists of two steps: extraction of key terms from a document and associating these terms with Wikipedia pages.

Lexical ambiguity of language presents a main difficulty for automatic wikification. Therefore, word sense disambiguation (WSD) is a necessary step for the automatic wikifiers.

Another challenge for the automatic wikification is choosing terms that should be associated with Wikipedia articles. Marking every term described in Wikipedia with links makes the document hard to read. Therefore, only most relevant terms should be presented as links for a particular document. Such terms are usually called *key terms*.

There are many approaches to automatic wikification. Most successful wikifiers use supervised learning algorithms for word sense disambiguation and key terms extraction. For such algorithms, Wikipedia serves as a training corpus. However, the lack of testing corpora based on real data makes it extremely hard to compare differrent wikifiers and choose the best one.

In order to estimate the quality of automatic wikifier on real data, part of this data should be wikified manually by human expert. Difficulty of manual wikification depends on the number of key terms that should be linked to Wikipedia. In general

case, all terms in text should be associated with Wikipedia articles and some of them should be marked as key terms. This is required for separated testing of WSD and key term extraction algorithms.

This paper introduces WikifyMe[1], a Web-based system that aims at creating large wikified corpora with the aid of Web users. This system has a user-friendly interface that makes manual wikification much easier. We expect that this system will yield good corpora for comparing different wikifiers at a relatively lower cost.

The rest of the paper is organized as follows. Related work is described in the next section. Sect. 3 gives overview of the WikifyMe and provides intuition for decisions we made during development of the system. In Sect. 4, a description of a current dataset is presented. Conclusion and future work are discussed in Sect. 5.

2. Related Work

Wikipedia is an evident corpus for wikifiers evaluation. Each regular Wikipedia's page describes one unambiguous concept and has links to other pages of Wikipedia. In general case, each link consists of two parts: destination page and caption shown to readers. Therefore, the link could be interpreted as the annotation of the text in caption with meaning described by destination page. Another assumption concerning internal links is that users of Wikipedia make links only for key terms. Based on these ideas, researches extract random samples of Wikipedia's regular pages and use them as testing corpora.

Main drawback of this approach is a bias of testing results for algorithms that use Wikipedia's links for training. In addition, behaviour of key terms extractors trained with the aid of Wikipedia's internal links on real data is not well studied. Therefore, researchers make their own corpora based on different data sources.

Mihalcea [9] manually mapped some Wikipedia terms to WordNet terms in order to carry out experiments on commonly accepted standard tests of the SenseEval corpus. However, there is no one-for-one mapping between Wikipedia and Wordnet, therefore this approach is not commonly used.

Cucerzan created his own corpus for evaluation of the system described in the paper [3]. A set of 100 news stories on a diverse range of topics was marked with named entities, which were also associated with articles of Wikipedia. This corpus is publicly available, but annotations in there are sparse and limited to a few entity types.

Milne and Witten [10] used Mechanical Turk [1] service to annotate subset of 50 documents from the AQUAINT text corpus: a collection of newswire stories from the Xinhua News Service, the New York Times, and the Associated Press. However they only ask to annotate key terms. Therefore their corpus cannot be used for WSD evaluation with high recall.

---

1 http://wikifyme.ispras.ru

Kulkarni et. al. [7] developed browser based annotation tool for creating test corpus. They collected about 19,000 annotations by six volunteers. Documents for manual annotation were collected from the links within homepages of popular sites belonging to a handful domains including sports, entertainment, science, technology, and health. The number of distinct Wikipedia entities that were linked to was about 3,800. About 40% of the spots was labeled n/a, highlighting the importance of backoffs. This corpus is good for testing WSD algorithms, but it doesn't contain any information about keywords.

Similar corpus was created for evaluation of the algorithms described in paper [11]. Like previous one, this corpus has tags for all possible segments, even though there is no correct mark for them (these segments are marked as n/a). This corpus didn't provide any information about keywords as well. We added this corpus to our system, then revised marks and included information about keywords.

The idea of involving Web users into creation of training and testing corpora was described and implemented in OMWE project [2]. The aim of this project was creation of a large corpus for WSD task with the aid of Web users. Result of this project was a corpus for WSD tracks on the Senseval 3 conference. However, this corpus is based on WordNet senses. Therefore, it could not be directly used for wikifiers evaluation.

3. Description of the System

3.1. Terminology

To create a new test, the user have to upload and mark up a text file (we call such file "*a document*"). Document consists of plain text and metadata that represents *terms*, *concepts* and *keyphrases*. *Term* models a continous part of text which have significant semantic value and thus some *meaning*. *Meanings* are represented by *concepts*, that is, articles in Wikipedia. We defined the special "*not-in-wikipedia*" *concept* for cases when the term have valuable sense, but there is no right *concept* to reflect the sense.

The union of all *term meanings* forms the set of document's *concepts*. Some *concepts* may be thought as *key concepts*, which reflect main topic(s) of the document. So we think of *keyphrases* as the *terms* (that is pieces of text) whose *meanings* are *key concepts*.

3.2. Process of the Wikification

User selects by mouse some part of the text to mark up a *term* there. It's very important to accurately select the term boundaries, so we had implemented several techniques that help users to do that.

The first feature is selection expansion to the boundaries of selected words. For example, selecton *"Scala is a great p***[rogramming langu]***age"* would be expanded to *"Scala is a great* **[programming language]***".*

The second technique allows to remove unnecessary spaces from the selection. *"Evaluation of* **[delimited continuations]***is supported"* becomes *"Evaluation of* **[delimited continuations]** *is supported"*. Both techniques can be enabled or disabled at any moment.

After the *term* has been created the user is offered to select a *meaning* for the *term* (see Figure 1). The *meaning* can be represented by any article in Wikipedia, hovewer for each term we provide a list of recommended *concepts*. These *concepts* were obtained from wiki-links appeared in Wikipedia articles that contain the term text. The *concepts* are ranked according to how often links to them anchored the *term* text. If certain *concept* was used once as a *meaning* for the *term* in the document, then the system put it in the top of list

Ошибка! Объект не может быть создан из кодов полей редактирования.

Figure 1: *List of recommended meanings for the "AT&T" term*

List of document *concepts* are shown on the right panel (fig. 2). User may click on any *concept* and mark it a *key concept*. This will mark all *term* representation of the *concept* as *keyphrases*.

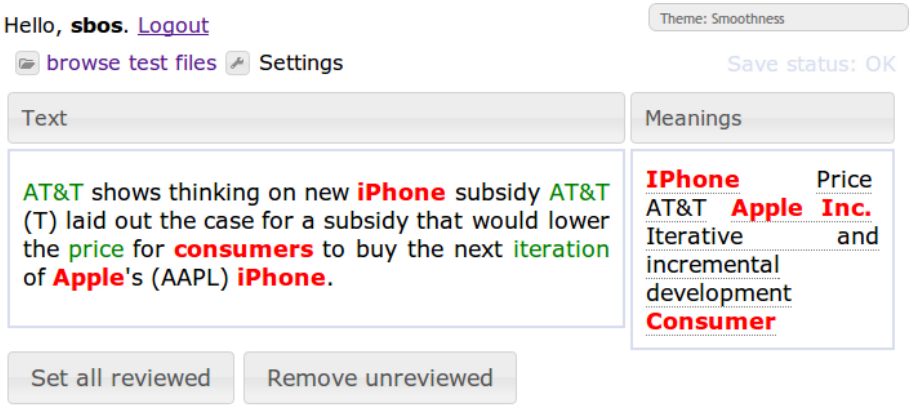


Figure 2: *Preparing the test. Green terms are reviewed, red ones are unreviewed.*

Bold concepts on the left side are marked as key concepts.We have restricted the *term* markup by only one term on a single part of text. That means no two different *terms* could be intersected by each other. We have found such restriction is a reasonable simplification, which lighten the user interface and facilitate user's interaction with the system. Also, our experience in the creation of WSD tests shows that single user has no need in making one piece of text a part of several

terms and this limitation is very common. However, if several users select overlapping parts of text as a terms in their versions of the same document, then this will be represented in resulting test as we describe in 3.5.

3.3. Preprocessing

To make the test creating process more easy we provide automatic preprocessing feature which uses wikifier described in [6] to automatically detect terms in documents, assign them right meanings and select key concepts. Meanings assigned in such way are marked as *non-reviewed*. This feature significantly improves the speed and usability of test creation process because users should just review these *meanings* as well as "key" status of document *concepts*.

3.4. Documents and Folders

Documents in WikifyMe are organized in folders. Each folder has a name and optionally a description. Users are able to create new folders, so the user who creates a new folder is treated as this folder owner. Each folder is accessible to all users. However only folder owner can delete it or upload new documents into it. To allow other users upload new documents to the folder, it has to be marked as "public" by its owner.

Whenever user opens a document uploaded by another user, the new version of the document is being created. This version doesn't contain any information from the original document except the plain text, so users have to work on the same documents independently. This is good because each user is not affected by possible mistakes of others. Users can delete their versions of documents, but original documents can be deleted only by owners of containing folders.

3.5. Getting the Tests

Everyone can get the whole test collection by click on the "Merge and download" button. WikifyMe will merge all versions of all files and provide the results in a singe archive.

The process of merging is quite simple: to merge a set of documents WikifyMe builds a resulting document which consists of terms, meanings and key concepts from all these documents. Then the system counts an agree level (we call it a *confidence*) for each term, meaning and key concept (a *keyphraseness*) selection.

The *meaning confidence* for each *term* is counted by formula:

$$\text{confidence} = \frac{|\,\text{this meaning selections}\,|}{|\,\text{this term selections}\,|} \quad (1)$$

The *keyphraseness* of *key concepts* is counted as:

$$\text{keyphraseness} = \frac{|\text{versions where the concept is key}|}{|\text{versions where the concept appear}|} \quad (2)$$

WikifyMe also count the *confidence* of *term* selection:

$$\text{confidence} = \frac{|\text{this term selections}|}{|\text{other terms overlapped by this term}|} \quad (3)$$

We treat two *terms* the same if their boundaries are matching exactly. So the *confidence* of two *terms* which *meanings* just overlap does not decrease, but the *confidence* of *term* selection does.

3.6. Output format

The XML as a widespread format for annotated text files has been chosen for the output format of merged documents. The example of the document is shown in Figure 3.

The **concept** tag define the *concept* in the document with **name** and **id** attributes that refer to Wikipedia article's name and ID obtained from Wikipedia dump. **concept** tag also contain the **representation** tags, each of them define the *term* associated with containing **concept** as their *meaning*. **span** attribute have a "*start..end*" and indicate the position of *term* in the text.

term tag also defines a *term* and completely duplicates an information from certain combination of **concept** and **representation**. This redundancy is due to different data structures are more suitable for different tasks. Thus, usage of **term** tags is convenient for word-sense disambiguation while **concept** tags are suitable for semantic analysis of the document.

Sense of **confidence** and **keyphraseness** attributes have been described above.

Ошибка! Объект не может быть создан из кодов полей редактирования.

Figure 3: *Example of downloaded XML test file.*

4. Data

Currently, WikifyMe contains 8 folders with 132 documents from very different sources - from scientific papers and blog posts to summaries from Google News.

Such variety is quite helpful for testing on different kind of texts and we except the document collection to be broaden by users.

Folder	# of terms	avg. doc length
Greg-January-2008	661	336.7
Monah-DBMS2-May-2008	686	242.7
news_google_com_26_may_2008	844	386.3
radar_oreilly_jan_2007	482	803.6
scientific_papers	858	1761
sqlsummit-June2008	419	89.5
UPI_Entertainment_17_22_may_2008	1898	162.6
UPI_Health_01_06_june_2008	1297	201.2

Table 1. *Statistics for base corpora*

Greg-January-2008, *Monah-DBMS2-May-2008*, *radar_oreilly_jan_2007* refer to blog posts collection from Greg Linden, DBMS2 and Tim O'Reilly blogs respectively. *news_google_com_26_may_2008* folder contains news articles by 26th May of 2008 from Google News, *UPI_Entertainment_17_22_may_2008* and *UPI_Health_01_06_june_2008* - from Health and Entertainment sections of "United Press International". *scientific_papers* as the name suggests consists of scientfic papers directly converted from PDF to plain text and *sqlsummit-June2008* contains short news summaries from "SQL Summit" blog. Summary for the corpora is presented in the Table 1.

Initially the base corpora has been marked up by one person in average, thus the *confedence* and *keyphraseness* metrics are about 1.0 and are not representative at the current stage.

Corpus	Number of terms
WikifyMe	7145
Milne et. al. tests	314
Kulkarni et. al. (IITB)	17200

Table 2*. Comparison of corpora.*

Table 2 contains the comparison by number of terms between Kulkarni et. al. [7] manually collected "ground truth" corpus named IITB, Millne et. al. [10] test corpus, which was automatically wikified by their tool and manually verified then, and WikifyMe manually collected corpus. As we can see, at the moment WikifyMe's corpus is comparable to IITB and outperforms Millne et. al. corpus by number of tagged terms, so it's suitable enough for WSD benchmarking tasks.

5. Conclusion

Despite WikifyMe is a ready-to-work system already there are still lot of possibilites to make it better and at first we plan to add the existing test corpora such as Kulkarni et. al. [7] and Milne et. al. [10] used in their researches.

As a key of the whole project success is the active contribution of users we will add several features to the web tool to stimulate the user activity. For example, public statistics for amount of work made by each user (maybe included in the archive with tests). We believe that it will make a sense because it's important for a user to feel that he or she is a part of the project and the value of self contribution made is visible to everyone.

We hope that WikifyMe will gather the active user community and help to create a large and high-quality test collection useful for researchers in wikification.

References

[1] **Элементы оглавления не найдены.**

WikifyMe: создание модели сравнения для викификаторов

Бартунов С.О., Болдаков А.А., Турдаков Д.Ю.
sbartunov@gmail.com, boldakov@gmail.com, turdakov@ispras.ru

Аннотация. Поиск взаимосвязей между словами в тестке и статьями "Википедии"-чрезвычайно популярная задача, известная как викификация. Не смотря на её популярность, до сих пор не существует общепризнанного тестового корпуса для сравнения викификаторов. В данной статье представлен онлайн-инструмент для совместной работы над универсальной коллекцией тестов для двух наиболее сложных задач в викификации – разрешения лексической многозначности и выделения ключевых слов.

Ключевые слова: "Википедия", обработка естесственных языков, разрешение лексической многозначности, выделение ключевых слов, тестирование, совместная работа, краудсорсинг, веб-инструменты, подготовка тестов.

A category-driven approach to deriving domain specific subsets of Wikipedia

Anton V. Korshunov,
Denis Yu. Turdakov
{korshunov, turdakov}@ispras.ru.

Jinguk Jeong, Minho Lee, Changsung Moon
Convergence Solution Team, DMC R&D Center,
Samsung Electronics Co., Ltd.
{jinguk.jeong, minho03.lee, albert.moon}@samsung.com

Abstract. While many researchers attempt to build up different kinds of ontologies by means of Wikipedia, the possibility of deriving high-quality domain specific subset of Wikipedia using its own category structure still remains undervalued. We prove the necessity of such processing in this paper and also propose an appropriate technique. As a result, the size of knowledge base for our text processing framework has been reduced by more than order, while the precision of disambiguating musical metadata (ID3 tags) has decreased from 98% to 64%.

Keywords. Wikipedia; ontology; automated ontology building; category; taxonomy; semantic relatedness; natural language processing; Texterra.

1. Introduction

There's no need to introduce Wikipedia as a world's largest and most rapidly expanding source of information on many domains in almost every language. At the time of drafting this paper, there are 279 Wikipedias in different languages, with more than 17,870,000 articles, 1,920,000 uploaded images, and 27,540,000 registered users [1]. 39 Wikipedias contain more than 100,000 articles each. The English edition remains the largest Wikipedia, over three times as large as the second largest edition, the German Wikipedia. That's why it is reasonable to start investigating new possibilities with English Wikipedia as a most comprehensive source.

By now, Wikipedia has already got a lot of colorful detailed descriptions. Conventional features of Wikipedia are well-known and discussed widely in [2, 25]. They include concept identification by ID or URL, multiple and dense link structure, and category system which is edited and maintained by Wikipedia users as well as articles.

Researching community has proved more than once that structure and content of Wikipedia are very peculiar and valuable domains to study. One of the most promising directions is *automated ontology building* which may be accomplished by extracting well-defined concepts and relations among them from Wikipedia. The latest efforts in this field are DBpedia [3], the work of Ponzetto et al. [4], YAGO [5], etc. All these approaches mainly exploit the data derived from infoboxes and category structure. Due to continuous enriching these techniques with better algorithms and data sources, the quality of resulting ontologies becomes remarkable.

Despite these successes in automated upper-level ontology building, most of domain specific ontologies (for instance, UMLS [6]) are still assembled manually. Needless to say, they are often costly to build and to keep them up to date.

In this paper, we consider the possibility of loosely supervised extraction of domain specific ontologies from upper-level ones. This is becoming feasible in recent years as high-quality upper-level ontologies grow more versatile and involve the knowledge of many new domains. Moreover, many field experts from all around the world often tend to pay their efforts to expand existing widespread ontologies, rather than to create their own or refine domain specific ones.

Our interest in this research is caused by necessity of reducing the knowledge base for our text processing framework Texterra[1]. This base comprises a number of textual indices produced by our Wikipedia parser. These indices are loaded into RAM during the initialization of Texterra server and take roughly 4.5 Gb of disk space and 2 Gb of RAM. Such a consumption is acceptable for workstations, but not for mobile devices with restricted amount of memory. Thus, if one would attempt to build a standalone mobile application intended for text processing, then its data structures simply would not fit in the RAM. Obviously, such applications - if developed - would be highly demanded to date.

The disambiguation[2] of musical metadata (i.e., ID3 tags of MP3 files) is an important part of Texterra functionality. These algorithms are well-tuned by the moment and show the precision of 98% on our test set. But it's obvious that only a narrow range of Wikipedia dictionary is utilized during processing ID3 tags of MP3 files. There are mostly named entities, such as musical compositions, song writers, singers, etc. Generally speaking, the amount of knowledge required to perform such a disambiguation is likely much less than that available from the entire knowledge base.

1 http://modis.ispras.ru/texterra

2 **Word sense disambiguation** is an open problem of natural language processing, which governs the process of identifying which sense of a word (i.e. meaning) is used in a sentence, when the word has multiple meanings (polysemy).

Making the knowledge base more specialized by removing concepts which are unimportant for this task is what comes to mind first in this case. So, we've implemented a system intended to produce Wikipedia subsets covering the knowledge of the given domain. We decided not to narrow the scope of our research to Wikipedia derivates (such as YAGO). But, although the algorithms have been evaluated with Wikipedia only, they are applicable to any ontology with well-defined polyhierarchical taxonomy.

Of many interesting aspects of Wikipedia, here we take into account only categorization and linkage within its content. The category system in Wikipedia plays the role of a taxonomy and provides the function to search articles by narrowing down categories. Given this, the task of deriving domain specific concepts from Wikipedia dictionary seems to be solvable by mining the category system for the list of concepts tightly connected with the given *base categories*. However, it is impossible to simply determine all concepts belonging to a particular category. The reason is that the category system of Wikipedia is organized into network structure, not a perfect tree structure. Furthermore, the lists of parent categories for many articles are redundant and contradictory. Thus, they don't allow detecting the most relevant categories for a given page without analyzing the neighbour pages.

Therefore, we utilized *concept vectorization method* specialized for the category system in Wikipedia, with several additional expansion methods, as proposed in [2]. The authors express affiliation relations among concepts as *category-based concept vectors*. Each element (dimension) of a concept vector represents not only binary affiliation information (whether the concept belongs to a certain category or not), but also the degree of affiliation (called *belonging degree* in the rest of this paper). After applying different cutting techniques to the obtained concept vectors, the list of domain specific concept IDs is outputted. This list is further applied to Texterra knowledge base resulting in the reduced domain specific version of the latter. The explanation of our approach is given in Section 4.

Section 2 contains the review of related work. The key features of Wikipedia category structure are discussed in Section 3.

Having a fast access to any point of Wikipedia category structure was one of the crucial tasks for our research. After a number of unsuccessful tries to reuse the existing graph libraries, we've coded our own implementation, named **WikiGraph**. See Section 5 for details.

Given a set of specialized versions of the knowledge base, we conducted a series of experiments in order to learn how the reducing of Texterra knowledge base affects the accuracy of results. Refer to Section 6 for evaluation methodology. Section 7 contains experimental results and discussion.

We conclude in Section 8 with our research findings and discussion about possible improvements.

2. Related work

2.1. Automated ontology building

The field of automated ontology learning usually acts by taking textual input and transforming it into a taxonomy or a proper ontology. The texts are usually obtained from printed sources (books, magazines, newspapers) and Internet (online media, blogposts, results of querying search engines). However, the learned ontologies are small and hard-to-update; in addition, evaluations have revealed a rather poor performance [7].

As mentioned before, the most accurate and, thus, valuable non-human assembled ontologies are now built by automatically deriving explicit facts from Wikipedia. One of the early attempts was the work of Gregorowicz and Kramer [8]. They focused on deriving a term-concept map which consists of terms, concepts, and relationships between them. Only articles, redirects, and disambiguation pages are considered. Ponzetto and Strube [4] were deriving a taxonomy from the entire Wikipedia category structure. Despite their successes, the resulted taxonomy is rather simple (supports only *is-a* and *not-is-a* relations) and domain independent. The work described in [9] enhances this taxonomy with *instance* and *class* information for each node. Cui et al. [10] introduce even more sophisticated approach to building the ontology of concepts by making use of *infobox* structures, definition sentences, and category labels. Finally, YAGO2 [11] is probably the most complete and accurate semantic knowledge base derived automatically from Wikipedia and other sources. The information extraction technique for YAGO2 assumes varied utilization of infoboxes, category structure, redirects, and other data within Wikipedia. Furthermore, the quality check is performed to find possible mistakes. As a result, the quality of extracted ontology is sufficient for the majority of IR- and NLP-related tasks.

Notwithstanding the foregoing, newly emerging research fields often require well-structured and comprehensive *domain specific* ontologies. Researchers don't need a huge knowledge base, but an extensible corpus of specific concepts with good coverage of domain knowledge. If such ontologies were built in a completely automated way, then this would avoid the necessity of assembling them from scratch manually. We believe that Wikipedia contains enough information for this task to be completed. Below is the description of approaches we've applied to narrowing Wikipedia dictionary to domain specific subset.

2.2. Computing semantic relatedness

We consider the key problem of our study as the computation of *semantic relatedness* between base top-level categories and underlying articles. According to Budanitsky and Hirst [12], semantic relatedness is defined to cover any kind of lexical or functional association that may exist between two words. This definition suits us more than *semantic similarity*, which is typically defined via the lexical relations of synonymy and hypernymy.

There were many approaches proposed for estimation of semantic relatedness between concepts in Wordnet (Rada et al., Leacock and Chodorow, Wu and Palmer, Resnik, Jiang and Conrath, Lin) and between Wikipedia articles [13] (Dice, Jaccard, SimRank). Among them are purely graph-based measures and those involving information content. In this paper, we consider only graph-based approaches.

Since a significant part of Wikipedia knowledge is encoded in its graph-like link structure, it seems reasonable to apply existing graph-based methods or introduce new ones. Zesch and Gurevych [14] proved that many Wordnet-based semantic similarity measures are applicable to Wikipedia with minor changes.

Obviously, it's also attractive to estimate the strength of ties between different levels of taxonomy (between base categories and its articles, in our case). In this context, the links among articles appear not so important. Therefore, new category-based semantic relatedness measures are emerging.

Chernov et al. [15] measured semantic relatedness *between categories*, not between concepts and categories, as it's required for our task. Nonetheless, they proposed several very useful and applicable techniques.

Strube and Ponzetto [16] employed Wikipedia, Wordnet, and Google for computing semantic relatedness *between concepts*. For two Wikipedia articles being compared, they extracted two categories lists. Given the category lists, for each category pair they performed a depth-limited search of maximum depth of 4 for a least common ancestor. As they noticed, limiting the search improves the results. But this is obviously inappropriate for computing relatedness between top-level category and articles from all its subcategories.

2.3. Identifying domain specific concepts

Syed et al. [17] tried to predict the topic of textual documents by matching them against Wikipedia articles based on cosine similarity[3]. Then, they extracted categories of found articles and scored them based on different scoring schemes with or without spreading activation[4]. The proposed approach implies scoring the links with many categories for each given Wikipedia article using bottom-up

3 **Cosine similarity** is a measure of similarity between two vectors by measuring the cosine of the angle between them. Calculating the cosine of the angle between two vectors thus determines whether two vectors are pointing in roughly the same direction. This is often used to compare documents in text mining.

4 **Spreading activation** is a method for searching associative networks, neural networks, or semantic networks. The search process is initiated by labelling a set of source nodes with weights or "activation" and then iteratively propagating or "spreading" that activation out to other nodes linked to the source nodes.

traversing of category structure. This is acceptable for a small set of articles, but not for our task.

To the best of our knowledge, Cui et al. [18] were the first who introduced an approach to deriving domain specific corpus from Wikipedia. The main idea is to generate a domain hierarchy from the hyperlinked pages of Wikipedia. Then, only articles strongly linked to this hierarchy are selected. They build a so-called *Classification Tree* by traversing down the directed graph of Wikipedia category structure starting from the *root node*. This tree includes both categories and articles and in fact is merely a connected branch of Wikipedia classification graph with a specified root node. Then, the Classification Tree is traversed with a simple adaption of breadth-first search algorithm. During the traversal, each node is given a score on the relevance to the specific domain. Once the traversal is completed, the terminal nodes (article pages) are ranked according to the domain relevance scores. Pages over a certain threshold are considered domain relevant. The node score can consider either ingoing or outgoing edges. Despite the proposed technique is quite simple, the results are remarkable.

A more sophisticated algorithm has been proposed later by Shirakawa et al. [2]. The *concept vectorization method* is introduced for finding concepts which are highly correlated with the base category (refer to Section 1 for brief explanation). The main assumption there is that the relatedness between categories gets lower as the number of traversed pages (i.e., hopcount) increases. In addition to the number of links for each node, they also take into account the *number of paths* between the concept and the base category, as well as *hopcounts* of these paths. As it seems to us, this understanding reflects the nature of Wikipedia classification approach much more precisely than ever before. Several heuristics are suggested for estimation of semantic relatedness by counting paths properties in the subgraph of desired base category. However, authors didn't compute these scores for thousands of articles with hundreds of paths for each of them at a time, as it's required in this research. Moreover, the efficiency of both approaches described in [2] and [18] has not been evaluated in real tasks. In both cases, the evaluation was performed by comparing the results of algorithms with answers of experts knowledgeable in certain fields. This looks persuasively when proving the theoretical applicability, but is not enough for unconditional embedding into the real system.

In this work, we mainly exploited the ideas formulated in [2] and [18]. Our main goal was to estimate the scalability and practical applicability of these approaches for real tasks which imply processing of large amount of data.

3. Features of Wikipedia category structure

The advantages of Wikipedia category structure were studied by authors of [14] and many others. Here we summarize only those features needed for better understanding of our approach.

Categories of Wikipedia can be organized in a graph, where the nodes are categories and the edges are hyperlinks. In this work we also add articles to this graph. However, we still name it the *Wikipedia category graph* (*WCG* in the rest of the paper).

The links expressing which concept belongs to what categories are called *category links*. We call them *belonging links* or *belonged links* according to their direction. In this paper, we only consider the belonging links, i.e. links from articles or subcategories to upper-level categories. The English version of Wikipedia, as of September 2010, contains ~13 million category links.

```
<page>
  <title>My Heart Will Go On</title>
  <id>923235</id>
  <revision>
    <id>390604275</id>
    <timestamp>2010-10-14T00:14:19Z</timestamp>
    ...
    <text xml:space="preserve">
    {{Infobox single
      | Name           = My Heart Will Go On
      ...
    }}

    "'''My Heart Will Go On'''" is the
    [[theme song]] of the 1997 blockbuster
    [[film]] ''[[Titanic (1997 film)|Titanic]]''.

    ...

    [[Category:Celine Dion songs]]
    [[Category:1997 singles]]
    [[Category:1998 singles]]
    [[Category:1990s ballads]]
    [[Category:Love themes]]
    [[Category:Pop ballads]]
    ...
    </text>
  </revision>
</page>
```

Fig. 1. *Sample XML structure of categorized Wikipedia article page*

The typical code of categorized article page is shown at Fig. 1. It combines XML and Wiki markup. The list of belonging categories is situated at the bottom. Categories have their own pages similar to articles. Category links at these pages also express which category belongs to what categories.

Categorization is a useful tool to group articles for ease of navigation, and correlating similar information. However, not every verifiable fact (or the intersection of two or more such facts) in an article requires an associated category. For lengthy articles, this could potentially result in hundreds of categories, most of which aren't particularly relevant. This may also make it more difficult to find any particular category for a specific article. Such *overcategorization* is also known as "*category clutter*" [19].

For these reasons, the WCG has an extremely complex nature. It is directed and has not a strong hierarchical structure as some may expect. Any category may branch into subcategories, and it is possible for a category to be a subcategory of more than one parent [20]. Upon closer inspection, the WCG is rather a *polyhierarchy*, or even a *net* (Fig. 2).

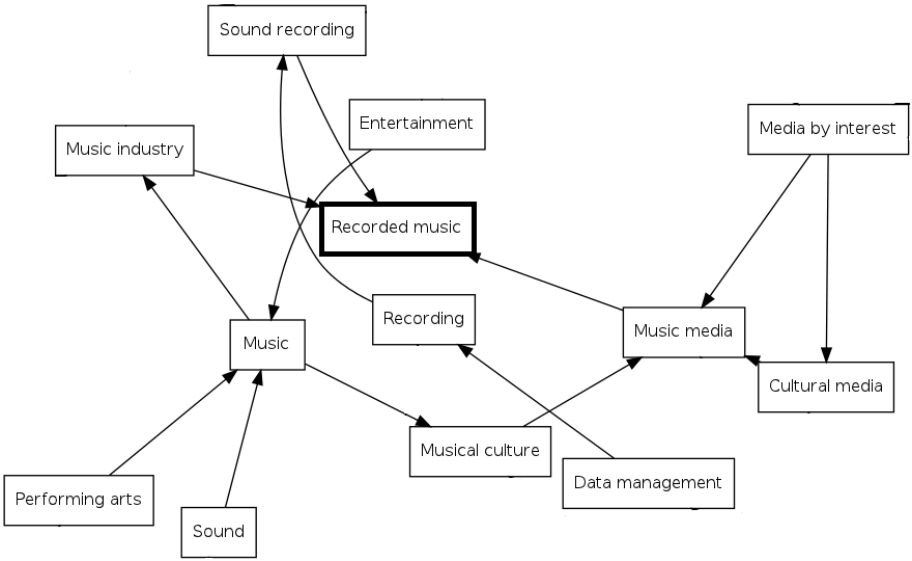


Fig. 2. *A fragment of Wikipedia category structure*

The figure has been produced by *CatGraph* [21]. This tool draws a cloud of links for the desired category. Each rectangle represents a category. Each arrow connecting two rectangles denotes a "*belongs-to*" relation, that is, the destination category is a subcategory of the initial one (an example of *belonging link*). The cloud shown in the figure is for "Recorded music" category (bolded).

It's worth noting here that not every Wikipedia page is categorized. According to statistics [22, 23], there are thousands of uncategorized articles and categories. Moreover, certain categories are assigned incorrectly [24]. We suggest considering these facts as a possible drawbacks for any category-based algorithm. In addition,

automated categorizing (i.e., determining a topic of an uncategorized page) seems to be a challenging task. This can be done with certain accuracy by processing page title and text with specific NLP techniques and finding appropriate categories in WCG.

4. Deriving a domain specific subsets

The developed system consists of three main parts:

1. **Link Filter** produces a ready-for-load textual representation of WCG;
2. **Topic Deriver** performs the main processing;
3. **Reducer** produces a domain specific version of the Texterra knowledge base[5].

All algorithms evaluated in this paper were implemented in the Java programming language.

4.1. Link Filter

The input for **Link Filter** is *Wikipedia links file* containing information about all links between Wikipedia pages, along with their type and direction. The result is *category links file* that contains only links forming the WCG. Every line of this file denotes the affiliation of *belonging* between two pages and sets the *type of the belonging page*. For example,

12 780754 0

means that page with ID 12 ("Anarchism") is *belonging* to the category with ID 780754 ("Category:Anarchism"). Moreover, *the belonging page* ("Anarchism") is *an article* because the last field is "0" ("1" would mean that the belonging page is *a category*).

Thus, category links file is a complete textual representation of the WCG and contains no unwanted data such as page titles and link types. Furthermore, unlike the authors of [18], we've also removed pages of certain types: lists, classifications, portals, redirects, disambiguation pages, and user pages. This helped us to make the WCG more lightweight without loss of any meaningful concepts. As a result, category links file contains 13,001,687 links between 593,796 categories and 3,156,822 articles.

4.2. Topic Deriver

Topic Deriver loads category links file on start and fills in the internal structures of WikiGraph (refer to Section 5 for details). The workflow for this stage is shown at the Fig. 3. Here we touch on only the main steps.

5 The Texterra knowledge base for this research has been obtained by parsing the dump of English Wikipedia, as of September 2010.

For our analysis, we denote **W** as a set of concepts, **V** as a set of categories, and **E** as a set of belonging links. Then, the category system in Wikipedia is expressed as a directed graph **G={W, V,E}**. A path is a sequence of edges that connects one node with another. The path length (hopcount) is the number of edges along that path.

Musical compositions
Recorded music
Music-related lists

Texterra knowledge base

ID_1
940477
ID_2
2722900
ID_3
1520543

Wikipedia category graph

"Musical compositions"
SUBGRAPH
"Recorded music"
SUBGRAPH
"Music-related lists"
SUBGRAPH
"Musical compositions"
VECTOR
"Recorded music"
VECTOR
"Music-related lists"
VECTOR

16256
723678
90381
2763203
392
84678
280482
1.56
1.12
0.92
0.56
0.35
0.24
0.03
1234

491358
76036
386421
6032
16937
1.25
1.042
0.89
0.33
0.19
0.01
67891
2297721
6788201
404313
51578
30557
18358
1.02
0.90
0.78
0.44
0.36
0.20
0.09

LIST OF EXTRACTED CONCEPTS

Knowledge base (complete)

Knowledge base

(domain specific)

Step 1. Get the **list of categories** to be processed.
Step 2. Get connected to Texterra server and obtain an **ID for each category**.
Step 3. Explore the pre-loaded Wikipedia category graph and build a **separate subgraph for each category**.
Step 5. Apply **cutting technique** to each vector. Collect the remaining IDs into common **list of extracted concepts**.
Step 4. Traverse the subgraphs and build **concept vectors**. Each vector consists of "**concept ID → belonging degree**" pairs.
Step 6. Process Texterra knowledge base in a way to save **only records associated with obtained list**.

Fig. 3. *Overall architecture of Topic Deriver*

The key task of Topic Deriver is to obtain the list of concepts connected semantically with certain domain. Herewith, this connection should be the tightest

one, that is, these concepts should be more relevant to the desired domain than to others. As this task is computationally complex and, thus, supposed to be run rarely, it's allowably to choose the base categories manually for experiments. We've selected 3 base categories that likely cover the majority of concepts required for disambiguating musical metadata:

- *Category:Musical compositions*
- *Category:Recorded music*
- *Category:Music-related lists*

For each of selected base categories, a separate *subgraph* is built. This subgraph is almost the same as *Classification Tree* in [18]. The base category serves as a root node, and a tree-like structure of underlying pages is obtained from WCG. The only difference from approach proposed in [18] is that we use depth-first search (DFS), not breadth-first search (BFS). The reason for this is that the resulting subgraphs are often large enough, thus, it's inappropriate to waste the memory for storing the FIFO queue required for BFS traversal [26]. Moreover, as depth-first tree is expected to contain back edges and cross edges, the list of visited nodes has been added to avoid repetitive visiting and loops.

A *concept vector* in our research specifies the degree of affiliation between *the base category* and *each of articles* reachable by traversing down the subgraph of the base category starting from its root node. As mentioned, the heuristics for building concept vectors have been borrowed from [2] with some modifications. We describe them briefly below, for more detailed information refer to the source paper.

BVG (Basic Vector Generation method) generates concept vectors by tracking back parent categories in the category system and calculating the belonging degree to each concept.

The belonging degree **I(wi,vj)** from concept **wi** to category **vj** is defined by the following equation:

$$Iwi,vj = p \in Pij 1d(tl) \qquad (1)$$

Here, **Pij** denotes a set of paths from **wi** to **vj**, tl denotes the hopcount of path **pl**, **d** denotes a monotonically increasing function on the hopcount of path pl (given as 2**tl**).

It's noteworthy that in the original method [2] paths with a hopcount of more than 4 were ignored. We asked the authors for the reasons of this. The response was "*Because long paths scarcely affect values in concept vectors in most cases. Of course, sometimes long paths affect the values*". We've decided to remove this constraint in our experiments, that is, we consider *all paths* between two nodes.

As a result, processing time may become too large for base categories from high levels of the WCG hierarchy. The reason for such behaviour is an exclusive computational complexity of finding all paths between two arbitrary nodes in the graph. It's well-known that this task is NP-complete in general case.

Since WCG contains millions of edges, the maximal path length may reach hundreds of edges, leading to impetuous increase of processing time when trying to process top-level categories. This is exactly why we've picked up a "safe" set of categories, which are processed relatively fast and cover the knowledge of field we've chosen for experiments.

Notwithstanding, we believe that taking all existing paths between two nodes into account allows to estimate the belonging degree more precisely. However, we didn't confirm this assumption experimentally.

To reduce the complexity of finding all paths between two arbitrary nodes, we tried to re-use one of existing techniques [27-29]. Finally, the APAC algorithm [29] has been chosen. This algorithm does not need to keep track of all visited vertices and only stores the feasible paths.

For domain specific areas where categories are excessively segmentalized, the BVG method cannot extract accurately concept vectors due to the increase in hopcount. To solve this problem, the **Single Parent Integration (SPI) method** is proposed. The authors confirmed from their experiences that a part in the category system which corresponds to (excessively segmentalized) categories for a domain specific area forms almost a tree structure. Based on this fact, when a concept or a category has only one (onehop/multihop) belonging link, the SPI method shortens the belonging link. This is based on the idea that the characteristic is not dispersed even when parent categories are tracked back if the concept or category has only one (onehop or multihop) belonging link.

In the **SPI** method, if there is only one belonging link ek from node **vi** (or **wi**) to **v**∈**V**, the path length of **ek** is accounted as **0**, which results in reformation of **E** to **E**', and then the **BVG** method is applied to **G'={W, V,E'}**.

VVG (Variance-based Vector Generation method) considers the weight of each category link. This method is based on the idea that the belonging degree from a certain category (concept) to parent categories depends on the number of parent categories, thus the weight of each category link is inversely proportional to the number of parent categories.

Thus, the weight of a category link becomes 1 if the category has only one parent category. That's why the authors argue that the VVG method contains the same feature as the SPI method. Therefore, they didn't combined VVG with SPI. We, on the other hand, tried both BVG + SPI and VVG + SPI combinations and confirmed that VVG + SPI performs slightly better than VVG itself (see Section 7 for details).

In the **VVG** method, weights are set to all belonging links, and the belonging degree from concept **wi** to category **vj** is calculated according to the weights. When the number of belonging links from node **vi** (or **wi**) to category **v**∈**V** is n, weight **bek** of each of the belonging links ek is defined as follows:

$$bek=1n \qquad (2)$$

Then, given all paths **P={p1,p2,…,pn}** from **wi** to **vj**, belonging degree **I(wi,vj)** from concept **wi** to category **vj** is defined as follows:

$$Iwi,vj=p\in Pc(pl) \quad (3)$$

c(pl) is the weight of path pl, calculated by the following equation:

$$cpl=eh\in Elbeh \quad (4)$$

Here, El={e1,e2,…,em} denotes a set of all belonging links forming path pl and eh denotes a belonging link.

After the vector is built and sorted in descending order of belonging degree, it's time to apply cutting technique to it and get the list of IDs *most relevant* to the base category. We've tried out two approaches:

1. *belonging degree threshold* – concepts with belonging degrees less than the mean value of belonging degree for each vector are filtered;
2. *percent threshold* - 25% of concepts with the lowest belonging degrees are filtered.

Finally, Topic Deriver produces *domain concepts file* that contains IDs of derived concepts.

4.3. Reducer

Reducer is the final part of the system. It takes domain concepts file as input, applies the concepts' list to complete Texterra knowledge base, and produces the reduced domain specific version of the latter. It contains not only concept IDs, but also full information about each of them, including the part of category structure that covers selected concepts. Therefore, the domain specific version is consistent and ready for loading into Texterra.

5. An approach to storing Wikipedia category graph

As showed above, fast access to any point of Wikipedia category structure is necessary for efficiency of all described computations. In particular, VVG method requires both entire WCG and subgraph of current base category to be available simultaneously.

Chernov et al. [15] studied semantic relationships between Wikipedia categories. They exported the dataset of about 670 thousands pages into a MySQL database. The data size was ~1.2 Gb. But, like many other researchers, they picked just a small sample of pages for processing (few thousands). For such small-scale approaches, even a usual on-disk relational DB is fast enough.

But our goal was to create a technique for fast iterative traversing through even a top-level categories with millions pages. Thus, we resorted to in-memory storage of WCG.

5.1. Evaluation results for known graph libraries

We've tried out two third-party libraries for storing the WCG in the JVM's memory. First of them, **JUNG** [30], showed satisfying performance results on small-scale subgraphs. But the entire WCG was impetuously expanding while loading and didn't fit in the RAM of the test machine (8 Gb). Second one, **JGraphT** [31], demonstrated almost the same behaviour: the WCG consumed a bit less amount of memory, but still too much. These observations hinder to utilize these libraries as a solution for WCG storing.

But there are a number of other libraries for graph storage which provide handy interface to stored data. We've found **neo4j** [32] and **WebGraph** [33] libraries. They may appear useful during the further research.

5.2. WikiGraph

The common shortcoming of all Java graph implementations we've tested seemed to be the redundancy of data stored in the RAM. Thus, the right way is to change the data storage manner. We put this into practice in **WikiGraph**.

All the prominent features of WikiGraph are due to the fact that it's *intended* to store WCG:

1. It is *directed* (as category links have a direction);
2. It introduces the notions of *category* and *article* and provides a powerful tooling to store and maintain the data on affiliations between them;
3. Only *IDs and types of pages* are stored. Each vertex is presented as a map consisting of [*ID, isCategory*] entries. This allows to store page type as a Boolean variable (*TRUE* is for category, *FALSE* is for article). All page data are saved as primitive variables, not an objects;
4. *Incidence list* has been chosen as a main data structure (along with vertices and edges lists). This is particularly important as the WCG is quite *dense*: number of edgesnumber of vertices≈41. The incidence list is organized into a set of [*vertex,* [*list of incident edges*]] entries. Each list is sorted in ascending order of edges IDs just after the loading. This avoids the need to look over the entire incidence list to get all the edges incident to an arbitrary vertex. Moreover, due to sorting of the lists, it's allowed to interrupt the search over them after the edge with greatest expected ID is found;
5. All *kinks* (self-to-self links) are removed;

6. *Initial capacity* of the incidence list is beforehand set to approximate amount of vertices in WCG (3,500,000 for this case). This saves some memory allocation costs while loading;

7. A set of *helper methods* is developed also (for instance, a method for deriving a subgraph of a given base category). This set provides usable and fast interface to the WCG data.

After the described features were implemented, they allowed us to fit WCG entirely in the RAM (~4 Gb needed) and lead to significant speed-up of loading and processing.

6. Evaluation methodology

Obviously, a domain specific subset of Wikipedia should have a good coverage of domain knowledge. But there is no easy direct way to evaluate quality of such a subset. The reason is that we must evaluate the completeness of knowledge available from Wikipedia's articles in resulting subset compared to that of specific domain. It's clear that this is rather difficult. In addition, the quality of link structure in the resulting subset should be also evaluated.

Therefore, we applied so called *in vivo* approach for evaluation. To estimate the quality of proposed methods, we studied how applying of the extracted subsets affects the performance of Texterra as a whole.

As mentioned before, one of Texterra parts is the system that enriches ID3 tags for musical recordings with links to corresponding articles of Wikipedia. This system utilizes graph structure to compute semantic relatedness between Wikipedia pages [13]. Then, semantic relatedness is exploited by word sense disambiguation algorithm. The latter is intended to choose the most relevant Wikipedia page from several homonymic variants.

We assume that each page of Wikipedia describes one possible meaning. WSD algorithm selects the most consistent combination of meanings that correspond to input ID3 tags. For sequence of input tags, it computes similarity between all pairs of meanings. The weight of a sequence is a sum of weights of all its pairs. Then, the algorithm detects a sequence with greatest weight.

To show good results, the derived subset should include as much as possible Wikipedia articles associated with a specific domain. In additional, link structure should be good enough for relatedness computation. Therefore, this approach allows evaluating both the quality of dictionary content and the quality of link structure.

For testing purpose, we derived several music-related subsets of Wikipedia by running different combinations of heuristics and used these subsets for described system. Then, we consequentially loaded these domain specific versions of knowledge base into Texterra and ran the tests. We used a small corpus of 20 random musical compositions and 49 different tags. Then, we estimated the

precision of automated disambiguation by comparing the results of algorithm with manually disambiguated tags.

7. Experimental results

The configuration of test workstation was as follows: Intel Core 2 Duo CPU (3.16 GHz), 8 Gb RAM, Windows 7 Enterprise 64 bit, Java SE 6 Development Kit 1.6.0.20.

Sample vectors for different combinations of heuristics are provided in Tables 1-4. Each sample vector comprises three concepts with highest belonging degree and three concepts with lowest values. The base category is *Category:Musical compositions*. It's noteworthy that BVG vector differs significantly from VVG one. Furthermore, enabling SPI affects both vectors.

ID	Title	Belonging degree
1784928	Candle in the Wind 1997	0.54
1523941	Axel F	0.50
1728643	Jeremy (song)	0.49
3720518	Ludwig Streicher	$2.44 * 10^{-4}$
2175948	Ian Bousfield	$2.44 * 10^{-4}$
875344	Willi Boskovsky	$2.44 * 10^{-4}$

***Table 1**. Sample vector produced by the BVG method*

ID	Title	Belonging degree
27684606	Niagara Falls Suite	0.50
20053503	Kumikyoku Nico Nico Douga	0.50
2501716	Megamix	0.50
14054430	Oh, by the Way	$6.02 * 10^{-7}$
454136	A Collection of Great Dance Songs	$6.02 * 10^{-7}$
361654	Echoes: The Best of Pink Floyd	$6.02 * 10^{-7}$

***Table 2**. Sample vector produced by the VVG method*

ID	Title	Belonging degree
1523941	Axel F	5.53
1815726	I Will Always Love You	3.88
923235	My Heart Will Go On	3.50
9010	Dance Dance Revolution	0.00
4527	Béla Bartók	0.00
1370	Ambrose	0.00

***Table 3**. Sample vector produced by the BVG method with SPI enabled*

	SPI enabled	Threshold	Number of IDs	Size, Mb	Precision, %
Basic Vector Generation	yes	belonging degree	334,575	112,3	45,10
		percent	574,810	251,2	**62,75**
	no	belonging degree	434,229	159,2	45,10
		percent	574,940	251,8	**64,71**
Variance-based Vector Generation	yes	belonging degree	420,302	154,8	49,02
		percent	549,791	244,5	60,78
	no	belonging degree	418,998	150,1	41,18
		percent	549,639	245,5	56,86
No threshold	—	—	675,228	311,5	72,55
Ground truth	—	—	8,476,942	4528,3	98,04

***Table 4**. Sample vector produced by the VVG method with SPI enabled*

ID	Title	Belonging degree
27684606	Niagara Falls Suite	0.50
20053503	Kumikyoku Nico Nico Douga	0.50
2501716	Megamix	0.50
9010	Dance Dance Revolution	0.00
4527	Béla Bartók	0.00
1370	Ambrose	0.00

***Table 5**. Experimental results*

The results of the experiments with different combinations of vector generation methods and cutting techniques are presented in Table 5. Contents of all 3 base categories listed in Section 4.2 are included.

Ground truth row corresponds to original Texterra knowledge base. As can be seen, it is huge, but ensures the best accuracy of disambiguation.

No threshold is for case when no cutting technique is applied to the concept vectors. In other words, this set of IDs exactly matches the set of all articles from subgraphs of all base categories. This version is much smaller, but the precision gets lower also. This precision drop (when no threshold is applied yet) is only due to imperfect choice of base categories. They merely don't cover all concepts required for precise disambiguation. It's also obvious that all comparisons of heuristics results should be done with no threshold results, not with ground truth.

As one can see, BVG performs a bit better than VVG. The most accurate combinations of heuristics are *BVG + percent threshold* and *BVG + SPI + percent*

threshold. Enabling SPI for VVG slightly increases the precision of disambiguation. Percent threshold is definitely better than belonging degree threshold.

What's important here is that the size of Texterra knowledge base (both on disk and in RAM) depends linearly on the number of concepts. Thus, the challenge is to find a compromise between the precision of disambiguation and the size (and contents) of the knowledge base.

The conducted experiment was just our first effort of this kind. Implemented algorithms allowed us to reduce the size of Texterra knowledge base by more than order, while the precision of disambiguating musical metadata has decreased from 98% to 64%. We believe that these results prove the applicability of proposed approach for deriving domain specific subset of Wikipedia. Certainly, there're still many things to improve.

8. Conclusion and future work

According to the results of this study, we outline the following:

- Wikipedia categories network may be utilized for domain specific subset of Wikipedia;
- Using concept vectors seems to be appropriate way to represent the affiliations of belonging between Wikipedia pages;
- BVG performs a bit better than VVG;
- SPI often improves the results;
- Percent threshold showed the best results as a cutting technique.

Possible directions of future work include:

1. As noted by the authors of [18], the selection of the root node is vital to the quality of the domain specific corpus. Thus, it's reasonable to introduce some heuristics for automated identifying of the most appropriate base category given just a set of specific keywords. Moreover, there can be several base categories with either manually or automatically set *relevance levels*. For example, to perform the search for "*Musicians of World War II*" a user should provide 2 base categories as an input: *Category:Musicians* with relevance level of 0.9 and *Category:World_War_II* with relevance level of 0.5;
2. Try other cutting techniques for concept vectors (i.e., attempt to detect the *distribution* of belonging degrees and utilize it);
3. Detect and remove meaningless pages from the WCG (i.e., pages from administrative section of Wikipedia [4]);
4. Distinguishing between classes and instances among categories [9] may help to prune and/or reorganize the WCG;

5. Add a facility for storing the results of semantic relatedness computation to boost the further processing;

6. Develop the approximation algorithm for finding all paths between two arbitrary nodes in the WCG.

In this work, we've demonstrated the possible benefits of automated building of the domain specific ontologies. Also, we've tested different heuristics while implementing the system for such processing. An original approach to storing WCG in the RAM has been proposed, along with specific evaluation methodology.

The described approach can be applied to any ontology with well-defined polyhierarchical taxonomy (for instance, YAGO2). As it seems to us, weighting the existing semantic connections is always a challenging task while building any more or less large ontology. This may be helpful for any domain dependent Wikipedia-related research [34, 35].

9. Acknowledgement

This research was collaborated with and supported by the Samsung Electronics Co., Ltd. DMC R&D Center Convergence Solution Team.

References

[1] *List of Wikipedias - Meta*. http://meta.wikimedia.org/wiki/List_of_Wikipedias
[2] M. Shirakawa, K. Nakayama, T. Hara, S. Nishio. *Concept Vector Extraction from Wikipedia Category Network*. In Proceedings of 3rd International Conference on Ubiquitous Information Management and Communication (ICUIMC 2009), pp. 71-79, 2009.
[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, Z. G. Ives. *Dbpedia: A nucleus for a web of open data*. In ISWC, volume 4825 of LNCS, pages 722–735. Springer, 2007.
[4] Simone P. Ponzetto, Michael Strube. *Deriving a large scale taxonomy from Wikipedia*. In AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence, pp. 1440-1445, 2007.
[5] Fabian M. Suchanek, Gjergji Kasneci, Gerhard Weikum. *YAGO: A Large Ontology from Wikipedia and WordNet*. In Elsevier Journal of Web Semantics, Vol. 6, No. 3, pp. 203-217, 2008.
[6] *Unified Medical Language System (UMLS) - Home*. http://www.nlm.nih.gov/research/umls/
[7] P. Buitelaar, P. Cimiano, B. Magnini (Eds.). *Ontology Learning from Text: Methods, Evaluation and Applications*. In Frontiers in Artificial Intelligence and Applications Series, Vol. 123, IOS Press, July 2005.
[8] A. Gregorowicz, M. A. Kramer. *Mining a Large-Scale Term-Concept Network from Wikipedia*. Technical Report #06-1028, The MITRE Corp., Oct. 2006.
[9] Cäcilia Zirn, Vivi Nastase, Michael Strube. *Distinguishing between instances and classes in the Wikipedia taxonomy*. In Proc. of ESWC-08, pages 376-387, 2008.

[10] Gaoying Cui, Qin Lu, Wenjie Li, Yi-Rong Chen. *Mining Concepts from Wikipedia for Ontology Construction*. In Proceedings of Web Intelligence/IAT Workshops, pp.287-290, 2009.
[11] J. Hoffart, F. Suchanek, K. Berberich, G. Weikum. *YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia*. Research Report MPI-I-2010-5-007, Max-Planck-Institut für Informatik, November 2010.
[12] A. Budanitsky, G. Hirst. *Evaluating WordNet-based measures of semantic distance*. In Computational Linguistics, 32(1), pp. 13-47, March 2006.
[13] D. Turdakov, P. Velikhov. *Semantic Relatedness Metric for Wikipedia Concepts Based on Link Analysis and its Application to Word Sense Disambiguation*. In Proc. of SYRCoDIS, 2008.
[14] T. Zesch, I. Gurevych. *Analysis of the Wikipedia Category Graph for NLP Applications*. In Proceedings of the TextGraphs-2 Workshop (NAACL-HLT), 2007.
[15] S. Chernov, T. Iofciu, W. Nejdl, X. Zhou. *Extracting Semantic Relationships between Wikipedia Categories*. In Proceedings of the First International Workshop on Semantic Wikis - From Wiki To Semantics, June 2006.
[16] M. Strube, S. P. Ponzetto. *WikiRelate! Computing semantic relatedness using Wikipedia*. In Proceedings of the 21st national conference on Artificial intelligence (AAAI'06), pp. 1419-1424, 2006.
[17] Z. Syed, T. Finin, and A. Joshi. *Wikipedia as an Ontology for Describing Documents*. In Proceedings of the Second International Conference on Weblogs and Social Media, 2008.
[18] G. Y. Cui, Q. Lu, W. J. Li, Y. R. Chen. *Corpus Exploitation from Wikipedia for Ontology Construction*. In LREC 2008, Marrakech, pp. 2125-2132, 2008.
[19] *Wikipedia:Overcategorization - Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Wikipedia:Overcategorization
[20] *Wikipedia:Categorization - Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Wikipedia:Categorization
[21] *Catgraph.* http://toolserver.org/~dapete/catgraph/
[22] *Wikipedia:WikiProject Categories/uncategorized - Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Wikipedia:WikiProject_Categories/uncategorized
[23] *Wikipedia:Database reports/Uncategorized categories - Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Wikipedia:Database_reports/Uncategorized_categories
[24] *Category:Better category needed - Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Category:Better_category_needed
[25] J. Soto. *Wikipedia: A Quantitative Analysis*. PhD thesis, 2009.
[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001. Section 22.3: Depth-first search, pp. 540–549.
[27] L.-E. *Thorelli. An algorithm for computing all paths in a graph*. In BIT 6, 347—349, 1966.
[28] M. Migliore , V. Martorana , F. Sciortino. *An algorithm to find all paths between two nodes in a graph*. In Journal of Computational Physics, v.87 n.1, pp.231-236, March 1990.
[29] R. Simoes. *APAC: An exact algorithm for retrieving cycles and paths in all kinds of graphs*. In Tékhne, no.12, p.39-55, 2009.
[30] *JUNG - Java Universal Network/Graph Framework*. http://jung.sourceforge.net/
[31] *JGraphT - a free Java graph library*. http://www.jgrapht.org/

[32] *neo4j open source nosql graph database*. http://neo4j.org/
[33] *WebGraph*. http://webgraph.dsi.unimi.it/
[34] *Wikipedia:Academic studies of Wikipedia - Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Wikipedia:Academic_studies_of_Wikipedia
[35] *Academic studies about Wikipedia - Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Academic_studies_about_Wikipedia#Natural_language_processing

Извлечение предметно-ориентированных подмножеств словаря Википедии с использованием структуры категорий

А.В. Коршунов, Д.Ю. Турдаков
{korshunov, turdakov}@ispras.ru

Чингук Чонг, Минхо Ли, Чансунг Мун
Группа Разработки Решений для Обеспечения Совместимости, Центр НИОКР по Цифровым СМИ и Коммуникациям,
Самсунг Электроникс Ко., Лтд.
{jinguk.jeong, minho03.lee, albert.moon}@samsung.com

Аннотация. В то время как многие исследователи пытаются построить различные онтологии с помощью Википедии, возможность получения качественных предметно-ориентированных подмножеств словаря Википедии остаётся недооценённой. Мы демонстрируем необходимость подобной процедуры и предлагаем соответствующую методику. В результате размер базы знаний нашего фреймворка для обработки текстов уменьшился более чем на порядок, а точность дизамбигуации метаданных музыкальных файлов (ID3-тегов) уменьшилась с 98% до 64%.

Ключевые слова. Википедия; онтология; автоматизированное построение онтологий; категория; таксономия; семантическая связанность; обработка естественного языка; Текстерра.

Heuristic data modeling in information systems

Martin Davtyan
martin.davtyan@gmail.com

Abstract. The paper describes research-in-progress work on information systems that store semi-structured data and use heuristics to make hypotheses about possible data structure. Such systems are intended to be more user-friendly as data formalization is performed by answering simple questions.

Keywords: data modeling, information systems, semi-structured data.

1. Introduction

As the information technology spreads over the globe, more and more people become involved in tasks of storing and exchanging information. Most of these people have no skills or time for programming or installing database software, neither these people can do data formalization needed. Blogging systems made it convenient for every person to share textual information or multimedia content. In some cases, like notes or shopping lists, the textual data representation is enough. But in other cases, people also need to structure their information somehow, to make it convenient to understand or share, for example, when storing small shop catalogues or club participant lists. There are also search benefits to that point, as having some data structure allows us to make search queries more meaningful than the full-text search queries. There are no tools allowing a non-professional user to set up a system to store and exchange semi-structured data. The requirements for such a system would be:

- Users should be able to store information and share it with other users.
- It should have a user-friendly interface for data structuring.
- It should also have a convenient search tool making use of the structure created.

In this article we propose an approach to constructing such a system.

Such a system would allow users with no preliminary knowledge of data structures or logic take full advantage of structured data, such as convenient navigation and powerful search. It would also free users from data formalization attempts when there is not enough data to formalize, but it still needs to be published.

The main challenge here is setting up the data structure. Making up a 'schema' implies proper formalization of the data, which is not always possible due to inability of a user. Formalization is also difficult if it is impossible to predict, what kind of properties or variables are about to be stored, before we get enough data to analyze.

This paper proposes a new approach for building user-friendly information system, letting a user store data in terms of objects of no predefined structure, and to simplify formalization of the stored data. Making propositions about a possible structure of data and asking user simple questions about it, we can use provided answers to construct a semantic-aware search in addition to a full-text search. Our hypothesis about 'schema' that fits the stored data can be also used for effective visual representation of this data on the Web, allowing more convenient way to share it. That way we can make use of data structure without forcing user to explicitly set it up for us.

2. Existing systems

There are two ways to build an information system required that people use now. The first way is to store information in tables. There is a variety of convenient spreadsheet software products providing intuitive way of data formalization, and there is no problem with adding new columns if we encounter new fields. The problems here are:

- The formalism user provides when setting up the column names is not used for searching, the search is still full-text. If there are ways to use complex queries in particular spreadsheet software system (i.e. queries that use the 'schema' provided by user), they are way too complicated for an average computer user to understand.
- Sharing is complicated, as users often end up emailing spreadsheet files to each other, copying rows from one table to another.

The second way of solving the problem is to invite programmers and data modelers to build a website. The user explains how exactly the data he intends to store looks like, and data modelers make up a database, in most cases a relational one, that is capable of display that data. Programmers then add a web application that allows user to create, edit and share that data on the Web. The problems of this approach are:

- Programmers and data modelers' work requires money and time.
- Any change in a data scheme will require calling programmers and data modelers again.

3. System overview and motivating example

In this section we propose a model of an information system that meets the requirements stated in Introduction.

The simplest and fastest way to share information with someone is to use the Internet, so we propose a system to have a website interface. User should be able to create objects – named key-value mappings of no predefined structure (as mentioned above) – and store them in a personal set. To illustrate the related workflow, let us consider an example.

Suppose the user intends to store car rental data and share it with current and prospective clients. There is only one basic action a user can perform when starting to work with the system – create an object. The user describes a car in terms of properties: {'transmission' : 'manual', 'mileage' : '30000km', 'horsepower' : '101'}. He also gives the object a name – 'Skoda Octavia I Tour' – and saves it. After saving the object, he is offered to create another one with the same property names, so he now only needs to fill in property values. As the quantity of user's objects increase, user gets exposed to the first question. The system outputs a number of objects (most likely with the same property set) and asks a user if it is possible to give all these objects a common name. The user's answer, if positive (e.g. 'Vehicle'), is used as a name of a new class. This is how classes are created.

Classes emerge in the system as the user inputs information. Suppose we have two classes registered – 'Vehicle' and 'Truck', only differentiating in a 'cargo weight' property the 'Truck' class has. The system outputs a number of objects of both classes and asks whether all 'Truck's are 'Vehicle's. If user confirms, the corresponding subclass relationship is registered.

By the time the user has published enough information, he is also being asked, whether the 'horsepower' property is always a number. If true, this property is registered to be of a certain data type.

All the data structure information the system gets from user's answers is used for user interface optimization. Being given an object classification, the navigation through user's objects is arranged accordingly. User also gets organized shortcuts for creating objects of registered classes. Data type information allows validation of newly created objects if they belong to a registered class.

The example shows how the system takes advantage of formal data structure without forcing user to do the formalization. Asking simple questions and providing related object examples to assist answering, data structure is exposed to the extent that is usable for crucial interface and search improvements.

4. System architecture and workflow

4.1. Input data and storage

As questions of building user interface and developing web applications are not in the scope of this paper, we shall make assumptions about what data we get from user. We assume that user provides objects, i.e. lists of key-value pairs, where key values are called property names, and values are called property values. We can state the data we store is semi-structured [1].

An intuitive solution for storing objects discussed above is a document-oriented database, such as CouchDB or MongoDB, as the structure of 'object' described above strictly corresponds to the structure of 'document'. In addition, using document-oriented database will make it easy to store data in a cloud storage. Another solution could be an XML database.

4.2. Workflow

We have no data model predefined when user starts to push data into the system, and, accordingly, our database has no schema. But as the user inputs some data, we make hypotheses about its structure, which we then check by asking the user simple yes/no questions.

One can think of information about data structure we get from user as a schema [2]. As schema in relational database systems is defined before storing actual data, and should be supported by DBMS, we propose a term 'heuristic schema' for data model information that is built according to data. Heuristic schema can emerge only after the data is stored, and it is only supported on application level.

4.3. Making hypotheses

Suppose we have a number of objects in our database, and consider different hypotheses can be made when analyzing their structure. The names examples of such hypotheses are given below:

- Heuristic class. If a number of objects have exactly the same property names, they are probably in the same class. The problem of finding classes is discussed below.
- Heuristic subclass. If heuristic class A has property names P_1, P_2, P_3 … P_n, and class B has property names P_1, P_2, P_3, … P_n, … P_m, heuristic class B is a subclass of heuristic class A.
- Heuristic data type. If property values stored under property name N are always numbers, property N is probably numerical. The same idea can be used with any popular data types, as strings or URLs. In general case one can think of any constraint that applies to values of N property as a data type [3].

Each of these hypotheses triggers asking user a corresponding question, as his answer causes refutation or confirmation. If a hypothesis is confirmed, user is also

asked to give a new heuristic data structure a name, to make it usable in further data input or search queries.

4.4. Using string distance for class hypotheses

Making heuristic class hypotheses only in case of exact match of all the object property names is useless in case of user's misspellings. This problem can be solved using string distances for property names.

As two objects belong to the same class only having the equal number of properties and property values have no impact on making class hypothesis, let us consider the sets of N property name strings. Dealing with misspelling problem, we use Damerau–Levenshtein distance for measuring difference between the strings. Damerau–Levenshtein distance is a string metric given by counting minimal amount of insertions, deletions or transpositions to transform one string to another. These operations correspond to more than 80 percent of human misspellings [4].

With distance between individual property name strings defined, we now need to define a similarity measure between string sets, which represent objects. Let us consider two sets A = {a_1, a_2, …, a_N} and B = {b_1, b_2, …, b_N} and Damerau–Levenshtein distance function d(a_i, b_j). Given a particular mapping F: A → B, similarity between two sets can be calculated as follows:

$$S = \sum_{i=1}^{N} d(a_i, F(a_i)) \quad (1)$$

As objects having the same property names belong to the same class regardless of the order in which properties are listed, we need to find the least possible distance between two sets. The problem of finding a mapping function F so that S is minimized is known as an assignment problem [5], which can be effectively solved by Munkres algorithm [6]. After using the algorithm to determine F, we use the corresponding S as a similarity measure. Setting up a threshold value for this measure will determine whether the class hypothesis should be made.

4.5. Using heuristic schema

These are the most important benefits of having schema in semi-structured data storage:

- Better user interface. After making user passively define a class and give it a name, we are able to create a template for this class. User now just has to choose a class for an object, instead of manually inputting property names.
- Classes can be also used for the Web representation of the stored data, with class hierarchy acting like a catalogue tree for navigation through site.
- Heuristic classes and data types can be used for data search using semantics, not only full-text search. For example, search query can set constraints on a certain heuristic field (e.g. number field), or certain class.

5. Related work

Ideas discussed in this paper are connected with Semantic Web technologies in a way they also intend to add some additional structure to the weakly structured data sets [7]. Semantic Web has a great stack of technologies for storing and exchanging semantic data, like OWL, RDF and others, that can be used as a formal representation of heuristic schema.

6. Future work

We are currently working on a prototype of the described system. It is planned to:

- Use semantic web standards for schema representation.
- Use current research results in automated ontology construction for heuristics.
- However, the main direction of our research is associated with cluster analysis. It is planned to define similarity on the whole object set, regardless of property amount. This will enable using clustering algorithms, which are useful for hypothesis construction:
- Hierarchical clustering results in a dendrogram, which can be used to build a taxonomy hypothesis.
- Validation within unclassified object set – error hypothesis proposed for objects not belonging to any existing cluster.

7. Conclusion

The described system acts like a compromise between easy and convenient text storage and powerful and searchable fixed schema databases. It allows user to publish data as soon as possible, and it also has all benefits of a structured data storage.

References

[1] Peter Buneman. Semi-structured data. Tutorial, PODS '97 http://www.cis.upenn.edu/~db/abstracts/semistructured.html

[2] http://en.wikipedia.org/wiki/Database_schema Database schema - Wikipedia, the free encyclopedia, February 2011

[3] Luca Cardelli, Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism, 1985

[4] Fred J. Damerau, A technique for computer detection and correction of spelling errors, Communications of the ACM, 1964

[5] http://en.wikipedia.org/wiki/Assignment_problem Assignment problem - Wikipedia, the free encyclopedia, April 2011

[6] Geoffrey C. Fox, Roy D. Williams, Guiseppe C. Messina, Parallel Computing Works, 1994

[7] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph. Foundations of Semantic Web Technologies, 2009

Эвристическое моделирование данных в информационных системах

Мартин Давтян
martin.davtyan@gmail.com
МГТУ им. Н. Э. Баумана

Аннотация. В статье описывается разработка информационных систем, которые хранят слабоструктурированные данные и используют эвристические методы для формирования гипотез относительно возможной структуры хранимых данных. Предполагается, что такие системы будут удобны в использовании, так как формализация данных будет производится путем ответов на простые вопросы.

Ключевые слова: моделирование данных, информационные системы, слабоструктурированные данные.

Development of a Parallel DBMS on the Basis of PostgreSQL*

C. S. Pan
kvapen@gmail.com

Abstract. The paper describes the architecture and the design of PargreSQL parallel database management system (DBMS) for distributed memory multiprocessors. PargreSQL is based upon PostgreSQL open-source DBMS and exploits partitioned parallelism.

Keywords: partitioned parallelism; postgresql; parallel dbms.

1. Introduction

Currently open-source PostgreSQL DBMS [1] is a reliable alternative for commercial DBMSes. There are many both practical database applications based upon PostgreSQL and research projects devoted to extension and improvement of PostgreSQL.

One of the directions mentioned above is to adapt PostgreSQL for parallel query processing. In this paper we describe the architecture and design of PargreSQL parallel DBMS for analytical data processing on distributed multiprocessors. PargreSQL represents PostgreSQL with embedded partitioned parallelism.

The paper is organized as follows. Section 2 briefly discusses related work. Section 3 gives a description of the PostgreSQL DBMS architecture. Section 4 introduces design principles and architecture of PargreSQL DBMS. The results of experiments on the current partial implementation are shown in section 5. Section 6 contains concluding remarks and directions for future work.

2. Related Work

The research on extension and improvement of PostgreSQL DBMS includes the following.

In [2] native XML type support in PostgreSQL is discussed. Adding data types to provide support of HL7 medical information exchange standard in PostgreSQL is

* The paper is supported by the Ministry of Education and Science of the Russian Federation (state contract No. 07.514.11.4036

described in [3]. The authors of [4] propose an image-handling extension to PostgreSQL. In [5] an approach to integration of PostgreSQL with the Semantic Web is presented.

There are papers investigating adoption of PostgreSQL for parallel query processing as well. In [6] the authors introduce their work on extending PostgreSQL to support distributed query processing. Several limitations in PostgreSQL's query engine and corresponding query execution techniques to improve performance of distributed query processing are presented. ParGRES [7] is an open-source database cluster middleware for high performance OLAP query processing. ParGRES exploits intra-query parallelism on PC clusters and uses adaptive virtual partitioning of the database. GParGRES [8] exploits database replication and inter- and intra-query parallelism to efficiently support OLAP queries in a grid. The approach has two levels of query splitting: grid-level splitting, implemented by GParGRES, and node-level splitting, implemented by ParGRES.

In [9] building a hybrid between MapReduce and parallel database is explored. The authors have created a prototype named HadoopDB on the basis of Hadoop and PostgreSQL, that is as efficient as a parallel DBMS, but as scalable, fault tolerant and flexible as MapReduce systems. PostgreSQL is used as the database layer and Hadoop as the communication layer.

Out contribution is embedding partitioned parallelism [10] into PostgreSQL. We use methods for parallel query processing, proposed in [11] and [12].

3. PostgreSQL Architecture

PostgreSQL is based on the client-server model. A session involves three processes into interaction: a frontend, a backend and a daemon (see fig. 1).

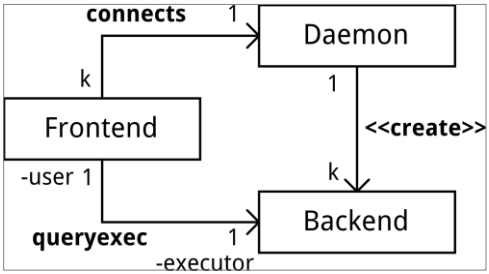


Fig. 1. PostgreSQL processes

The daemon handles incoming connections from frontends and creates a backend for each one. Each backend executes queries received from the related frontend. The activity diagram of a PostgreSQL session is shown in fig. 2.

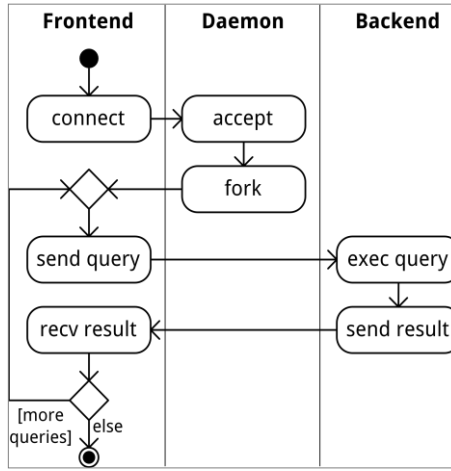


Fig. 2. A PostgreSQL session

There are following steps of query processing in PostgreSQL: *parse*, *rewrite*, *plan/optimize*, and *execute*.

Respective PostgreSQL subsystems are depicted in fig. 3. *Parser* checks the syntax of the query string and builds a parse tree. *Rewriter* processes the tree according to the rules specified by the user (e.g. view definitions). *Planner* creates an optimal execution plan for this query tree. *Executor* takes the execution plan and processes it recursively from the root. *Storage* provides functions to store and retrieve tuples and metadata.

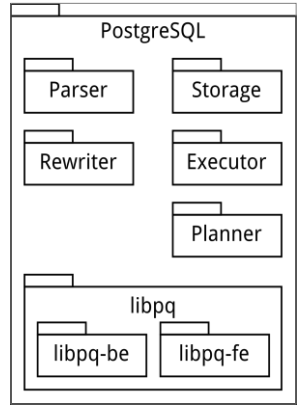


Fig. 3. PostgreSQL subsystems

libpq implements frontend-backend interaction protocol and consists of two parts: the frontend (*libpq-fe*) and the backend (*libpq-be*). The former is deployed on the client side and serves as an API for the end-user application. The latter is deployed on the server side and serves as an API for libpq-fe, as shown in fig. 4.

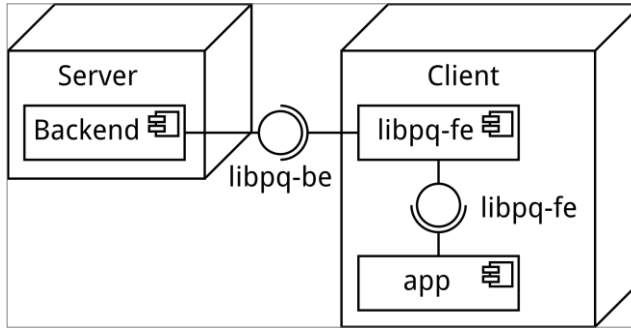


Fig 4. PostgreSQL deployment

4. PargreSQL Architecture

PargreSQL utilizes the idea of partitioned parallelism [12] as shown in fig. 5. This form of parallelism supposes partitioning relations among the disks of the multiprocessor system.

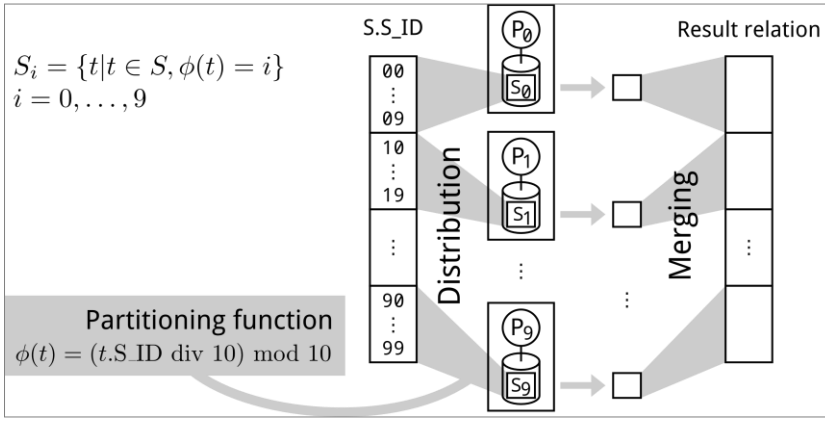


Fig. 5. Parallel query processing

The way the partitioning is done is defined by a *fragmentation function*, which for each tuple of the relation calculates the number of the processor node which this tuple should be placed at. A query is executed in parallel on all processor nodes as a

set of parallel *agents*. Each agent processes its own fragment and generates a partial query result. The partial results are merged into the resulting relation.

The architecture of PargreSQL, in contrast with PostgreSQL, assumes that a client connects to two or more servers (see fig. 6).

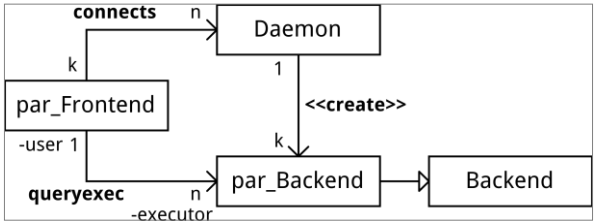


Fig. 6. PargreSQL processes

The interaction sequence is shown in fig. 7. As opposed to PostgreSQL there are many daemons running in PargreSQL. A frontend connects to each of them, sends the same query to many backends, and receives the result relation.

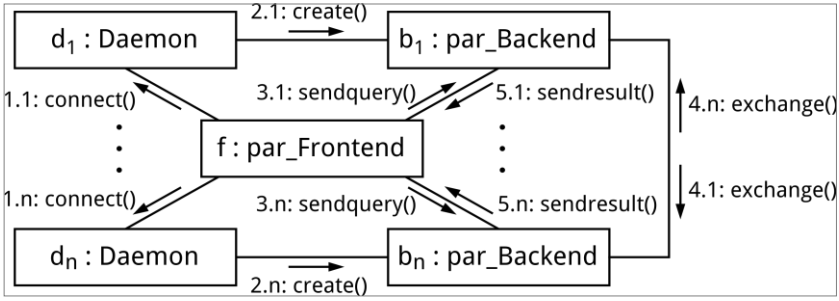


Fig. 7. Interaction of PargreSQL clients and servers

Parallel query processing in PargreSQL is done in more steps: *parse*, *rewrite*, *plan/optimize*, ***parallelize***, *execute*, and ***balance***. During the query execution each agent processes its own part of the relation independently so, to obtain the correct result, transfers of tuples are required. *Parallelization* stages creation of a parallel plan by inserting special exchange operators into the corresponding places of the plan. *Balance* provides load-balancing of the server nodes.

PargreSQL subsystems are depicted in fig. 8. PostgreSQL is one of them. PargreSQL development involves changes in Storage, Executor, and Planner subsystems of PostgreSQL.

The changes in the old code are needed in order to integrate it with the new subsystems. *par_Storage* is responsible for storing partitioning metadata of the relations. *par_Exchange* encapsulates the *exchange* operator implementation. *Exchange* operator is meant to compute the exchange function ψ for each tuple of

the relation, send “alien” tuples to the other nodes, and receive “own” tuples in response.

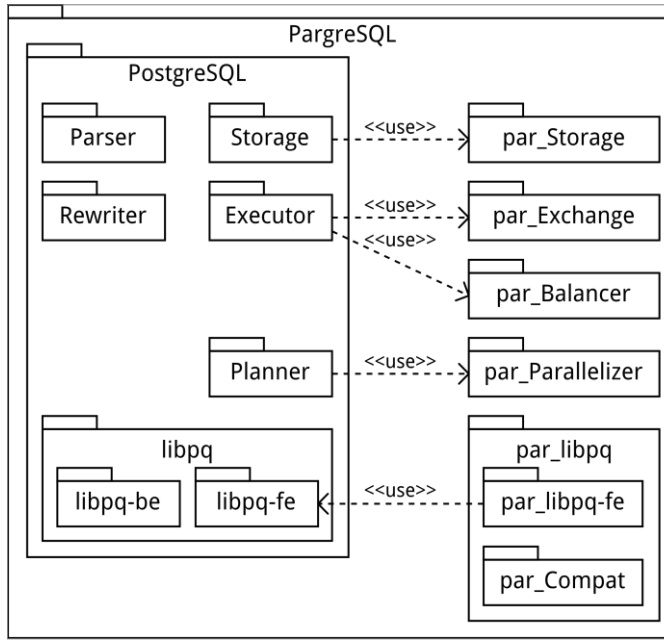


Fig. 8. PargreSQL subsystems

There are however some new subsystems which do not require any changes in the old code: *par_libpq-fe* and *par_Compat*. *par_libpq-fe* is a wrapper around *libpq-fe*, it is needed in order to propagate queries from an application to many servers. *par_Compat* makes this propagation transparent to the application.

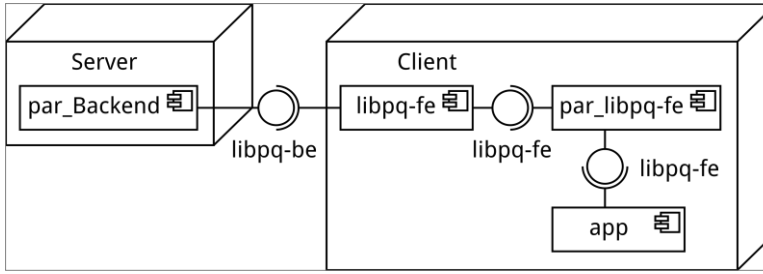


Fig. 9. PargreSQL deployment

The only difference of the deployment schemes (see fig. 9) is that there is one more component on the client side – the *libpq-fe* wrapper.

4.1. par_libpq Design

par_libpq subsystem consists of *par_lib-fe* library and a set of macros (*par_Compat*).

par_libpq-fe is a library that is linked into frontend applications instead of the original PostgreSQL *libpq-fe*, arouch which it is a wrapper. Its design is illustrated with a class diagram in fig. 10.

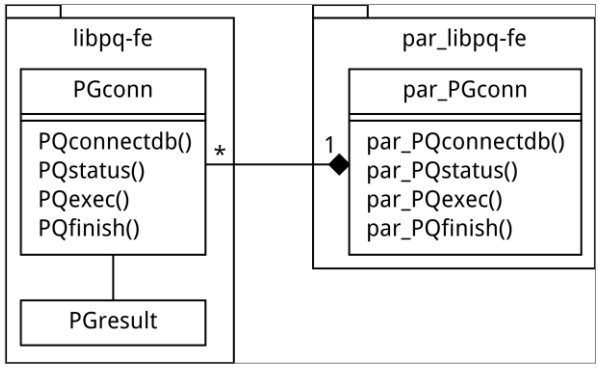


Fig. 10. PargreSQL libpq-fe wrapper

The idea is to use the original library for connecting to many servers simultaneously.

par_Compat is a set of C preprocessor definitions for transparent usage of *par_libpq-fe.* An example of what these macros are is given in fig. 11.

```
#define PGconn par_PGconn
#define PQconnectdb(X) par_PQconnectdb()
#define PQfinish(X) par_PQfinish(X)
#define PQstatus(X) par_PQstatus(X)
#define PQexec(X,Y) par_PQexec(X,Y)
```

Fig. 11. PargreSQL compatibility macros

Using these macros an application programmer can switch from PostgreSQL to PargreSQL without global changes in the application code.

4.2. Exchange Operator Design

Exchange operator [11, 12] serves to exchange tuples between the parallel agents. It is inserted into execution plans by Parallelizer subsystem. The operator's architecture is presented in fig. 12.

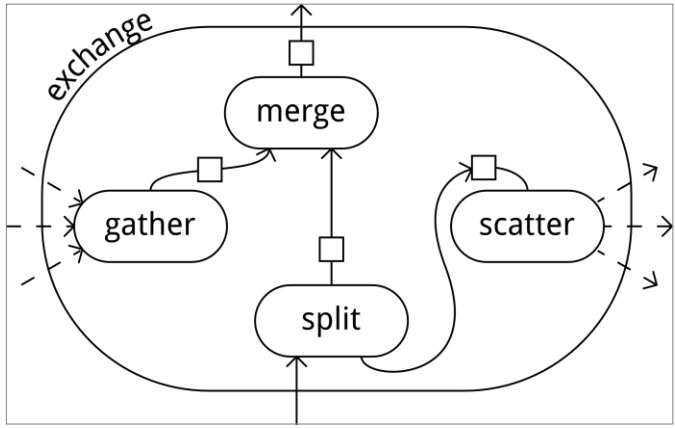


Fig. 12. Exchange operator architecture

Fig. 13 shows new classes (grouped in par_Exchange package) that implement *exchange* operator.

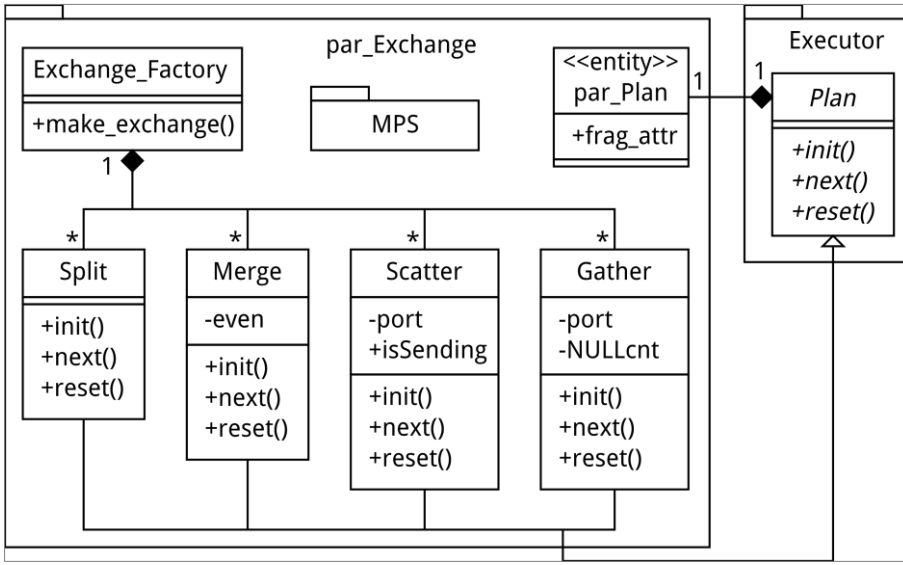


Fig. 13. Exchange operator classes

MPS subsystem (Message Passing System) is used by Scatter and Gather to transmit tuples. Its interface is like MPI reduced to three methods: ISend, IRecv, and Test. They are actually implemented on top of MPI.

Figs. 14, 15, 16, and 17 show algorithms for **next()** method of four exchange subnodes.

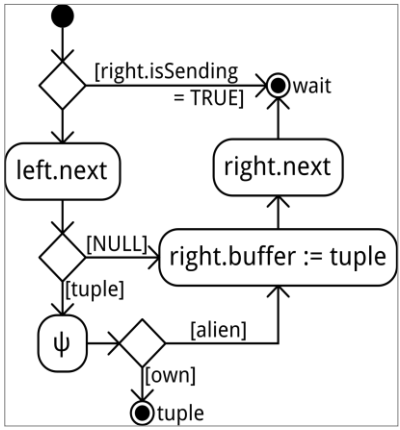


Fig. 14. ***Split.next()*** *method*

Split is meant to calculate the exchange function for each tuple and to choose whether to keep the tuple on the processor node or send it to other processor node.

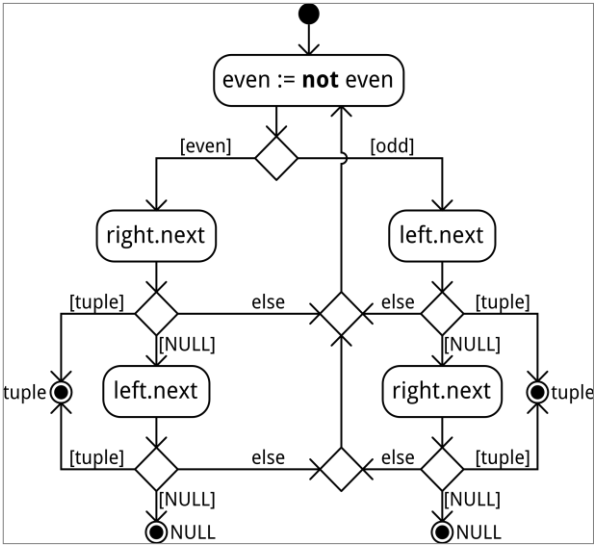


Fig. 15. ***Merge.next()*** *method*

Merge merges tuples from *Gather* and *Split*.

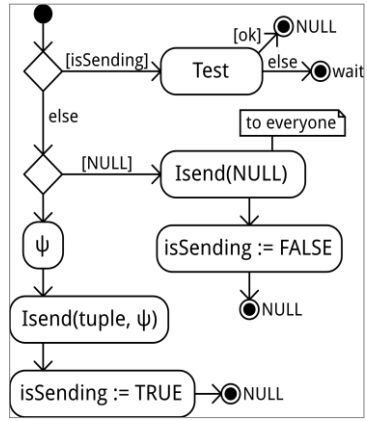


Fig. 16. ***Scatter.next()*** *method*

Scatter sends tuples coming from *Split* to other processor nodes.

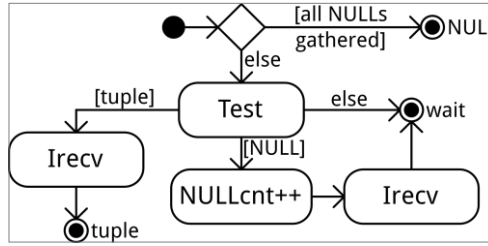


Fig. 17. ***Gather.next()*** *method*

Gather does the opposite, receiving tuples from other processor nodes.

5. Experimental Evaluation

At the moment we have implemented *par_libpq* and *par_Exchange* subsystems of PargreSQL. The implementation has been tested on the following query:

```
select * from tab where tab.col % 10000 = 0
```

The query has been run against table *tab* consisting of 10^8 tuples. The speedup over PostgreSQL is shown in fig. 18.

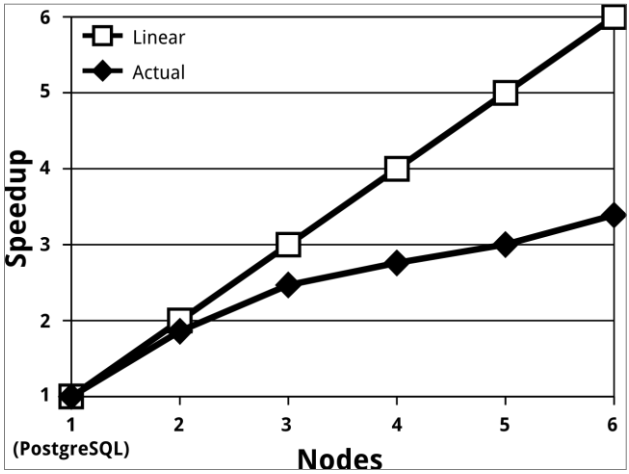


Fig. 18. PargreSQL speedup

6. Conclusion

In this paper we have described the architecture and the design of PargreSQL parallel DBNS for distributed memory multiprocessors. PargreSQL is based upon PostgreSQL open-source DBMS and exploits partitioned parallelism.

There are following issues in out future research. We plan to complete the implementation and to investigate its speedup and scalability. The future research is also going to be concentrated on implementing data updates, transactions and fault tolerance.

References

[1] M. Stonebraker, G. Kemnitz. The POSTGRES next generation database management system. Commun ACM, 34:78—92. October 1991.
[2] N. Samokhvalov. XML Support in PostgreSQL. In Sergei D. Kuznetsov, Andrey Fomichev, Boris Novikov, and Dmitry Shaporenkov, editors, SYRCoDIS, volume 256 of CEUR Workshop Proceedings. CEUR-WS.org, 2007.
[3] Y. Havinga, W. Dijkstra, A. de Keijzer. Adding HL7 version 3 data types to PostgreSQL. CoRR, abs/1003.3370, 2010.
[4] D. Guliato, E. V. de Melo, R M. Rangayyan, R. C. Soares. POSTGRESQL-IE: An Image-handling Extension for PostgreSQL. J. Digital Imaging, 22(2):149—165, 2009.
[5] D. V. Levshin, A. S. Markov. Algorith,s for integrating PostgreSQL with the semantic web. Programming and Computer Software, 35(3):136—144, 2009.
[6] R. Lee, M. Zhou. Extending PostgreSQL to Support Distributed/Heterogeneous Query Processing. In Kotagiri Ramamohanarao, P. Radha Krishna, Mukesh K. Mohania, and Ekawit Nantajeewarawat, editors, DASFAA, volume 4443 of Lecture Notes in Computer Science, pages 1086—1097. Springer, 2007.

[7] M. Paes, A. A. B. Lima, P. Valduriez, M. Mattoso. High-Performance Query Processing of a Real-World OLAP Database woth ParGRES. In Jose M. Laginha M. Pal ma, Patrick Amestoy, Michel K. Dayde, Marta Mattoso, and Joao Correia Lopes, editors, VECPAR, volume 5336 of Lecture Notes in Computer Science, pages 188—200. Springer, 2008.
[8] N. Kotowski, A. A. B. Lima, E. Pacitti, P. Valduriez, M. Mattoso. Parallel query processing for OLAP in grids. Concurrency and Computation: Practice and Experience, 20(17):2039—2048, 2008.
[9] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. PVLDB, 2(1):922—933, 2009.
[10] D. J. DeWitt, J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. Commun. ACM, 35(6):85—98, 1992.
[11] L. B. Sokolinsky. Organization of Parallel Query Processing in Multiprocessor Database Machines with Hierarchical Architecture. Programming and Computer Software, 27(6):297—308, 2001.
[12] A. V. Lepikhov, L. B. Sokolinsky. Query Processing in a DBMS for cluster systems. Programming and Computer Software, 36(4):205—215, 2010.

Разработка параллельной СУБД на основе PostgreSQL*

К. С. Пан
kvapen@gmail.com
Южно-Уральский государственный университет

Аннотация. Данная работа посвящена архитектуре и проектированию параллельной системы управления базами данных PargreSQL для многопроцессорных вычислительных систем с распределенной памятью. PargreSQL основана на СУБД с открытым исходным кодом PostgreSQL и использует фрагментный параллелизм.

Ключевые слова: фрагментный параллелизм; postgresql; параллельная субд.

* Работа выполнена при финансовой поддержке Минобрнауки РФ (государственный контракт № 07.514.11.4036)