ИСП
Российская Академия наук
Институт Системного Программирования

ISSN 2079-8156 (Print)
ISSN 2220-6426 (Online)

Труды Института Системного Программирования

Том 24

Москва 2013

ИСП

Российская Академия наук
Институт Системного Программирования

Труды Института Системного Программирования

Том 24

Под редакцией
академика РАН В.П. Иванникова

Москва 2013

УДК004.45

Труды Института системного программирования: Том 24.
/Под ред. Академика РАН В.П. Иванникова/ – М.: ИСП РАН, 2013.

В двадцать четвертом томе Трудов Института системного программирования публикуются статьи исследователей, аспирантов и студентов из ИСП РАН, Московского государственного университета и нескольких других организаций. В статьях описываются результаты исследований, выполненных в конце 2012 г. и первой половине 2013 г.

ISSN 2079-8156 (Print)

© Институт Системного Программирования РАН, 2013

Содержание

Предисловие.. 5

Разработка и реализация облачной системы для решения высокопроизводительных задач

А.О. Кудрявцев, В.К. Кошелев, А.О. Избышев, И.А. Дудина, Ш.Ф. Курмангалеев, А.И. Аветисян, В.П. Иванников, В.Е. Велихов, Е.А. Рябинкин *13*

Разработка и реализация облачного планировщика, учитывающего топологию коммуникационной среды при высокопроизводительных вычислениях

И.А. Дудина, А.О. Кудрявцев, С.С. Гайсарян... *35*

Исследование и разработка шаблонов неэффективного поведения в параллельных MPI, UPC приложениях

О. Д. Борисенко, Д. Ю. Турдаков, С. Д. Кузнецов *49*

Анализ эффективности итерационных методов решения систем линейных алгебраических уравнений, реализованных в пакете OpenFOAM

И.К. Марчевский, В.В. Пузикова.. 71

Расчет течений непрерывно стратифицированной жидкости с использованием открытых вычислительных пакетов на базе технологической платформы UniHUB.

Я.В. Загуменный, Ю.Д. Чашечкин 87

Прямая передача данных между ПЛИС Virtex-7 по шине PCI Express.

Ю.А. Румянцев 107

Методы оптимизации Си/Си++ - приложений распространяемых в биткоде LLVM с учетом специфики оборудования

Ш.Ф. Курмангалеев. 127

О методах деобфускации программ

Ш.Ф. Курмангалеев, К.Ю. Долгорукова, В.В. Савченко, А.Р. Нурмухаметов, Р.А. Матевосян, В.П. Корчагин 145

Вывод типов для языка Python

И.Е. Бронштейн 161

Автоматический поиск ошибок синхронизации в приложениях на платформе Android

В.П. Иванников, С.П. Вартанов, М.К. Ермаков 191

Опыт использования UniTESK как зеркало развития технологий тестирования на основе моделей

В.П. Иванников, А.К. Петренко, В.В. Кулямин, А.В. Максимов 207

Введение в метод CEGAR — уточнение абстракции по контрпримерам

М.У. Мандрыкин, В.С. Мутилин, А.В. Хорошилов 219

Построение спецификаций программных интерфейсов в открытой системе покомпонентной верификации ядра Linux

Е.М. Новиков 293

Автоматизация регрессионного тестирования при помощи анализа трасс событий

Владимир Федотов 317

Распределенные горизонтально масштабируемые решения для управления данными

С.Д. Кузнецов, А.В. Посконин 327

Инструментальные средства оценки качества научно-технических документов

С.В. Герасимов, Р.В. Курынин, И.В. Машечкин, М.И. Петровский, Д.В. Царёв,

А.А. Шестимеров 359

Современные методы поиска и индексации многомерных данных в приложениях моделирования больших динамических сцен

В.А. Золотов, В.А. Семенов 381

Особенности табличных выражений SQL и их соответствие с концепциями реляционной модели данных

И.В. Блудов 417

Роль предыстории при оценке сложного объекта в управлении по прецедентам

Л.Е. Карпов, В.Н. Юдин 437

Гибридный подход к построению систем поддержки решений

В.Н. Юдин, Л.Е. Карпов 447

Вероятностный анализ нового алгоритма упаковки прямоугольников в полосу

М.А. Трушников 457

П р е д и с л о в и е

В 24-м томе Трудов Института системного программирования РАН представлены статьи исследователей, аспирантов и студентов из ИСП РАН, Московского государственного университета и нескольких других организаций. В статьях описываются результаты исследований, выполненных в конце 2012 г. и первой половине 2013 г. Статьи данного тома сгруппированы в пять тематических блоков. Первый блок посвящен различным аспектам современных высокопроизводительных вычислений.

В статье А.О. Кудрявцева и др. «Разработка и реализация облачной системы для решения высокопроизводительных задач» описаны основные проблемы, возникающие при переносе высокопроизводительных вычислений в облако. Рассматривается подход к организации высокопроизводительного облачного сервиса с использованием виртуализации. Описана архитектура разработанной системы «виртуальный суперкомпьютер», основанной на облачной платформе OpenStack и системе виртуализации KVM/QEMU.

И.А. Дудина, А.О. Кудрявцев и С.С. Гайсарян представили статью «Разработка и реализация облачного планировщика, учитывающего топологию коммуникационной среды при высокопроизводительных вычислениях», в которой отмечается, что для эффективного решения высокопроизводительных задач в облаке необходимо планирование виртуальных машин на серверах, учитывающее особенности высокопроизводительных вычислений. Рассматривается подход к планированию, основанный на оценке размещения с помощью Hop-Byte метрики.

В статье М.С. Акопяна и Н.Е. Андреева «Исследование и разработка шаблонов неэффективного поведения в параллельных MPI, UPC приложениях» обсуждаются шаблоны в параллельных программах, приводящие к потере производительности. Рассматриваются шаблоны как в параллельных MPI приложениях для вычислительных систем с распределенной памятью, так и в параллельных UPC программах для систем с разделенным глобальным адресным пространством (PGAS). Предложен метод автоматизированного обнаружения шаблонов неэффективного поведения.

В статье «Анализ эффективности итерационных методов решения систем линейных алгебраических уравнений, реализованных в пакете OpenFOAM» предложен способ определения коэффициентов усиления гармоник, основанный на использовании дискретного преобразования Фурье. В качестве примера приведён анализ эффективности метода BiCGStab с ILU и многосеточным предобуславливанием из пакета OpenFOAM при решении разностных аналогов уравнений Гельмгольца и Пуассона, возникающих при моделировании течения вязкой несжимаемой жидкости в квадратной каверне.

В статье Я.В. Загуменного и Ю.Д. Чашечкина «Расчет течений непрерывно стратифицированной жидкости с использованием открытых вычислительных пакетов на базе технологической платформы UniHUB» описывается авторский опыт использования технологической платформы UniHUB при численном моделировании и проведении расчетов течений непрерывно стратифицированной жидкости с применением свободных прикладных вычислительных пакетов OpenFOAM, Salome и ParaView. Внимание уделяется вопросам построения высокоразрешающих расчетных сеток, постановки сложных граничных с помощью встроенных и расширенных утилит пакета OpenFOAM, разработки собственных решателей, обработки и визуализации расчетных данных, а также проведения расчетов в параллельном режиме на вычислительном кластере МСЦ РАН.

Завершает первый блок статья Ю.А. Румянцева «Прямая передача данных между ПЛИС Virtex-7 по шине PCI Express», в которой рассматривается передача данных по шине PCI Express с одновременным участием нескольких ПЛИС. В статье, в частности, показано, что при одновременных передачах через общий канал скорость отдельных передач не уменьшается до тех пор, пока суммарная скорость передачи не превышает пропускную способность общего канала; затем канал используется на 100%, а его пропускная способность делится поровну между устройствами.

Второй тематический блок содержит статьи, в которых обсуждаются методы компиляции и анализа программ, а также их приложений.

Статья Ш.Ф. Курмангалеева «Методы оптимизации Си/Си++ - приложений распространяемых в биткоде LLVM с учетом специфики оборудования» посвящена методам оптимизации Си/Си++ приложений, которые применяются в системе двухэтапной компиляции, позволяющей распространять такие приложения в промежуточном представлении LLVM. Описывается метод статического инструментирования с неполным покрытием всех дуг потока управления программы и последующей его коррекцией для сбора профиля, позволяющий получать профиль, сравнимый по качеству с классическим подходом, но при этом обеспечивающий существенное снижение накладных расходов на сбор профиля. Предлагается подход к построению специализированного облачного хранилища приложений, позволяющий решать вопросы оптимизации и защиты программ, а также обеспечивающий снижение накладных расходов на компиляцию и оптимизацию приложений в облачной инфраструктуре.

В статье Ш.Ф. Курмангалеева и др. «О методах деобфускации программ» отмечается, что в связи со схожестью проблем, стоящих перед оптимизирующим компилятором и деобфускатором, представляется разумным использование компиляторной инфраструктуры LLVM в качестве ядра деобфускатора. Хотя из-за высокой зашумленности выходного кода авторам пришлось разработать свою инфраструктуру, которая позволяет добиться более чистого выходного кода, применение LLVM или аналогичной

разработки остается одним из перспективных направлений при разработке деобфусцирующего программного обеспечения.

И.Е. Бронштейн в статье «Вывод типов для языка Python» приводит обзор описанных в научной литературе алгоритмов вывода типов для языков с параметрическим полиморфизмом. Затем описывается новый алгоритм, являющийся модификацией алгоритма декартова произведения. Демонстрируется, как модуль вывода типов, использующий новый алгоритм, анализирует различные конструкции языка Python.

Наконец, в статье С.П. Вартанова и М.К. Ермакова «Автоматический поиск ошибок синхронизации в приложениях на платформе Android рассматривается задача автоматического поиска ошибок синхронизации при проведении динамического анализа приложений в рамках платформы Android. Приводится обзор существующих инструментов для анализа компонентов Android-приложений. Описывается механизм обнаружения ошибок синхронизации, используемый инструментом динамического анализа байт-кода Coffee Machine, и его основные аспекты: инструментирование, генерация трассы отношений предшествования и использование инструмента ThreadSanitizer Offline для обнаружения состояния гонок по сгенерированной трассе для платформы Android.

Третий блок статей посвящен использованию формальных методов для тестирования программного обеспечения.

В.П. Иванников, А.К. Петренко и др. в статье «Опыт использования UniTESK как зеркало развития технологий тестирования на основе моделей» описывают наиболее значимые применения технологии UniTESK в промышленных проектах, суммируют их опыт и оценивают перспективные направления развития компонентных технологий тестирования на основе моделей.

М.У. Мандрыкин, В.С. Мутилин и Хорошилов А. В. в статье «Введение в метод CEGAR – уточнение абстракции по контрпримерам» указывают, что успешный автоматизированный анализ программных систем среднего размера с использованием проверки моделей, получаемых при помощи предикатной абстракции, стал возможен благодаря развитию метода уточнения абстракции по контрпримерам – CEGAR. Этот метод так или иначе используется в таких инструментах, как SLAM, BLAST, SATABS и CPAchecker. В статье рассматривается метод CEGAR в том виде, как он реализован в инструментах статической верификации BLAST и CPAchecker.

Статья Е.М. Новикова «Построение спецификаций программных интерфейсов в открытой системе покомпонентной верификации ядра Linux» содержит описание нового подхода к построению спецификаций программных интерфейсов, который позволяет достаточно эффективно применять инструменты статической верификации для проверки выполнения правил использования программных интерфейсов в условиях неполноты модели окружения.

В статье В.Н. Федотова «Автоматизация регрессионного тестирования при помощи анализа трасс событий» представлен алгоритм process mining, предназначеный для применения в инструментированной распределенной системе. Система инструментируется таким образом, что взаимодействия между модулями системы проходят через единую шину, связывающую все компоненты системы. Записанные при обработке на шине события анализируются представленным в работе алгоритмом, результатом работы которого является модель взаимодействий между модулями системы. Полученная модель используется для разработки покрывающего регрессионного тестового набора.

Четвертый тематический блок содержит ряд достаточно разнородных статей, так или иначе связанных с управлением данными и их анализом.

В статье С.Д. Кузнецова и А.В. Посконина «Распределенные горизонтально масштабируемые решения для управления данными» отмечается, что новые потребности современных приложений привели к появлению большого числа специализированных распределённых систем, способных удовлетворительно справляться с возникающими задачами. Предлагается обзор некоторых современных решений, обеспечивающих масштабируемость при работе с большими объемами данных под высокими нагрузками.

С.В. Герасимов и др. в статье «Инструментальные средства оценки качества научно-технических документов» предлагают комбинированный подход к оценке качества научно-технических документов, учитывающий различные категории автоматически рассчитываемых характеристик качества документов – как существующие библиометрические и наукометрические характеристики, так и новые типы характеристик, основанные на семантическом анализе текстов научно-технических документов, применении эвристических правил и методов оценки наличия прямых текстовых заимствований.

Статья В.А. Золотова и В.А. Семенова «Современные методы поиска и индексации многомерных данных в приложениях моделирования больших динамических сцен» посвящена современным методам поиска и индексации многомерных данных применительно к задачам моделирования больших динамических сцен. Для решения этих задач требуется применение специальных схем индексации многомерных данных. В статье обсуждаются два фундаментальных подхода к организации подобных схем, а также проводится их сравнительный анализ в контексте комплексных требований, предъявляемых к приложениям обсуждаемого класса.

В статье И.В. Блудова «Особенности табличных выражений SQL и их соответствие с концепциями реляционной модели данных» анализируется соответствие операторов SQL реляционной модели данных. Рассмотрены особенности табличных выражений в стандарте SQL и показаны случаи, в которых эти выражения противоречат реляционной модели. Приведены

варианты использования таких табличных выражений в SQL, которые наиболее точно отражают концепции реляционной модели.

Статья Л.Е. Карпова и В.Н. Юдина «Роль предыстории при оценке сложного объекта в управлении по прецедентам» посвящена принципам построения системы управления для нестационарных объектов – развитию подхода к управлению объектами по прецедентам, разработанного в рамках более ранних работ авторов, применительно к нестационарным объектам с неполным описанием. Упор сделан на разработку методики, позволяющей снизить неоднозначность оценки объекта до и после воздействия. Источником дополнительной информации может служить предыдущее поведение объекта.

В статье тех же авторов «Гибридный подход к построению систем поддержки решений» отмечается, что в существующих на данный момент гибридных системах поддержки принятия решений основным инструментом вывода являются порождающие правила, а прецеденты используются лишь для обработки исключений. В описываемом подходе рассуждения по правилам и прецедентам дополняют друг друга в условиях неоднозначной оценки неполностью описанного случая. Разработана методика двухэтапной оценки случая, где сначала используется прецедентный подход с целью получить представление о возможной принадлежности случая тому или иному классу. На основе этой информации, на втором этапе, уже по правилам вывода проводится обратный логический вывод от возможного заключения как гипотезы – к фактам, подтверждающим эту гипотезу.

Последний блок тома посвящен математическим проблемам компьютерной науки и содержит одну статью: М.А. Трушникова «Вероятностный анализ нового алгоритма упаковки прямоугольников в полосу». В статье говорится, что в 1993 году Коффман и Шор предложили онлайновый алгоритм упаковки прямоугольников в полосу с оценкой $O\left(N^{2/3}\right)$ для математического ожидания незаполненной площади полосы в стандартной вероятностной модели. С тех пор вопрос о возможности улучшения этой оценки в классе онлайновых алгоритмов оставался открытым. В данной работе проанализирован принципиально новый онлайновый алгоритм упаковки, предложенный ранее автором. Для него доказана оценка $O\left(N^{1/2}\left(\log N\right)^{3/2}\right)$ для математического ожидания незаполненной площади полосы.

Академик РАН В.П. Иванников

Разработка и реализация облачной системы для решения высокопроизводительных задач[1]

А.О. Кудрявцев, В.К. Кошелев, А.О. Избышев, И.А. Дудина,
Ш.Ф. Курмангалеев, А.И. Аветисян, В.П. Иванников,
В.Е. Велихов, Е.А. Рябинкин

Аннотация. В данной работе описаны основные проблемы, возникающие при переносе высокопроизводительных вычислений в облако. Рассматривается подход к организации высокопроизводительного облачного сервиса с использованием виртуализации. Описана архитектура разработанной системы «виртуальный суперкомпьютер», основанной на облачной платформе OpenStack и системе виртуализации KVM/QEMU. Компоненты системы ВСК доработаны для учета специфики высокопроизводительных вычислений, в частности, выполнена доводка и настройка системы виртуализации, что позволило достичь уровня накладных расходов не более 10% при использовании по крайней мере 1024 процессорных ядер.

Ключевые слова: облачные вычисления; виртуализация; мониторы виртуальных машин; высокопроизводительные вычисления; параллельные вычисления

1 Введение

При решении различных вычислительных задач, как правило, возникают колебания в объеме задействованных ресурсов, которые связаны с множеством факторов, начиная от характера решаемых задач и заканчивая временем года. В результате возникают ситуации, когда вычислительные ресурсы простаивают, либо ощущается нехватка ресурсов для решения задач пользователей, и, как следствие, возникают убытки от простоя оборудования или срыва сроков.

---

[1] Работа выполнена при финансовой поддержке Минобрнауки России по государственному контракту от 15.06.2012 г. № 07.524.11.4018 в рамках ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы»

С развитием сети Интернет появилась возможность удаленной работы с вычислительными ресурсами. В результате развития этой идеи была предложена концепция облачных вычислений, подразумевающая удаленную работу с ресурсом, причем задействованный объем ресурса может варьироваться в зависимости от потребностей. Такой ресурс предоставляется облачными провайдерами в многопользовательском режиме, оплата происходит по факту использованного объема ресурсов.

В последние годы популярна концепция облачного сервиса уровня инфраструктуры [1], позволяющая пользователю получать доступ к серверам, сетям передачи данных, хранилищам. Такой сервис позволяет организациям полностью отказаться от использования собственных вычислительных ресурсов. Экономия достигается за счет отсутствия необходимости покупки, установки, обслуживания собственного оборудования.

Ключевой особенностью облачных систем уровня инфраструктуры является использование виртуализации для реализации облачных серверов. Виртуализация предоставляет изолированное окружение, или «виртуальную машину» (ВМ), в которой возможно выполнение всего стека ПО сервера, включая операционную систему. Виртуальные машины могут быть консолидированы – запущены на одном сервере, что позволяет максимально задействовать ресурсы конкретных серверов. В связи с этим, такие облачные системы позволяют достигать существенной экономии средств за счет обеспечения высокой загрузки серверов. Виртуализация на платформе x86 изначально была реализована программным путем, в настоящее время используются аппаратные расширения процессора, позволяющие существенно повысить производительность виртуальных машин.

Облачные системы всегда были интересны научному сообществу как источник «бесконечных» ресурсов. Важным преимуществом переноса научных задач в облако является переносимость всего стека ПО между используемыми вычислительными ресурсами, а также между исследователями. Однако существующие ограничения производительности облачных систем долгое время не позволяли эффективно решать высокопроизводительные задачи.

Кроме того, существующие облачные платформы требуют доработки для обеспечения удобства пользователей – должна быть разработана экосистема, позволяющая ученому решать свои задачи без необходимости большого числа подготовительных шагов. Например, при необходимости запуска задачи на MPI-кластере, пользователь не должен самостоятельно настраивать данный кластер с использованием выделенных в облаке ресурсов.

Отдельной проблемой является обеспечение эффективного хранилища для данных (общей файловой системы). Такое хранилище должно обеспечивать высокую производительность в облаке, в том числе поддерживать параллельный ввод-вывод. Кроме того, должны

поддерживаться удобные средства вывода данных из системы. Проблема заключается в том, что при росте объема данных (и переходе к концепции BigData) становится все сложнее удовлетворять данным требованиям. При работе с BigData требуется также поддержка специализированных моделей обработки таких данных (в частности, MapReduce).

При переносе в облако также возникают проблемы, связанные с безопасностью, поскольку возможности пользователя по конфигурированию всего стека ПО существенно возрастают по сравнению с традиционными вычислительными кластерами. Неопытный пользователь может настроить облачный сервер таким образом, что он станет уязвимым для вредоносного ПО.

В настоящее время, в связи с развитием технологий виртуализации и повышением производительности облачных ресурсов, возникают новые возможности применения концепции облачных вычислений. Целью данной работы является разработка облачной системы, предназначенной для организации высокопроизводительных вычислений. В работе приводится обзор архитектуры разработанной и реализованной облачной системы «виртуальный суперкомпьютер» (далее – ВСК), позволяющей эффективно (с минимальными накладными расходами) решать высокопроизводительные задачи. Основным вкладом данной работы является:

1) Описание основных проблем, возникающих при переносе высокопроизводительных вычислений в облако, и методов их решения. В том числе, выявление основных источников накладных расходов при виртуализации и разработка методов их снижения.
2) Разработка и реализация облачной системы «виртуальный суперкомпьютер», предназначенной для решения высокопроизводительных задач в облаке. В системе ВСК реализован сервис уровня инфраструктуры, на базе которого реализован сервис по предоставлению доступа к MPI-кластеру, а также сервис OpenFOAM, демонстрирующий возможность работы с пакетом OpenFOAM как с облачным приложением. В системе основной упор делается на обеспечение производительности виртуальной среды на уровне не менее 90% от производительности реального кластера.
3) Описание перспективных подходов к организации научных и промышленных вычислений в облачной среде, перспективных разработок в данной области в мире.

В разделе 2 приводятся обзор литературы. Особенности архитектуры разработанной системы ВСК рассматриваются в разделе 3.

2 Обзор литературы

Концепция облачных вычислений подразумевает удаленный доступ к вычислительным ресурсам с возможностью получения ровно того объема ресурсов, который требуется. Согласно определению Национального института стандартов и технологий США (NIST) [2], облачные вычисления – это модель организации удаленного доступа по запросу к разделяемому набору конфигурируемых вычислительных ресурсов (например, сетей, серверов, хранилищ, приложений и сервисов), которые могут быть быстро выделены и освобождены с минимальными расходами на управление или взаимодействие с провайдером услуг.

Согласно NIST, выделяют несколько уровней сервиса, предоставляемого облачной системой. Сервис самого «низкого» уровня – предоставление инфраструктуры (IaaS, infrastructure as a service), позволяет получать доступ к вычислительным серверам, сетям передачи данных, блочным хранилищам. Как правило, сервис такого уровня реализуется с использованием технологий виртуализации, однако возможны и решения на базе аппаратного управления вычислительными ресурсами [3, 4] (которые также можно считать подходом наподобие виртуализации). Наиболее известными облачными платформами уровня инфраструктуры с открытым исходным кодом являются системы Eucalyptus [5], Nimbus [6], OpenNebula [7], CloudStack [8], OpenStack [9]. Среди систем с закрытым исходным кодом известны Amazon EC2 [10], Windows Azure [11], Google Compute Engine [12] и другие.

Сервис более высокого уровня – предоставление доступа к платформе (PaaS, platform as a service) и приложению (SaaS, software as a service). Сервис уровня платформы подразумевает доступ к вычислительной платформе, например, к окружению для разработки и выполнения программ, к СУБД, веб-серверу. Сервис уровня приложений позволяет работать с конкретным приложением удаленно. Зачастую сервисы более высокого уровня реализуются на основе сервиса IaaS, предоставляющего удобные средства для управления набором необходимых вычислительных ресурсов.

С точки зрения высокопроизводительных задач, можно выделить некоторые наиболее актуальные требования к облачной системе для эффективного решения таких задач:

1) Обеспечение высокой производительности вычислительных ресурсов. Данная задача должна быть решена на уровне сервиса инфраструктуры. В рамках данной работы рассматриваются методы эффективного выполнения параллельных программ в виртуальных машинах.
2) Предоставление доступа к коммуникационной среде, достаточной по своим характеристикам для решения конкретной задачи. Данная задача также относится к сервису

инфраструктуры. В рамках системы ВСК предоставляется доступ к сети Infiniband.

3) Организация удобной для пользователя экосистемы, позволяющей решать требуемые задачи без необходимости выполнения лишних шагов по конфигурированию и настройке, которые могут быть автоматизированы. Данное требование, по сути, определяет специализированный сервис уровня платформы. В данной работе приводится пример организации такого сервиса по доступу к приложению OpenFOAM.
4) Ключевым компонентом облачной экосистемы для решения научных задач также является высокопроизводительное хранилище. Такое хранилище должно быть доступно посредством различных интерфейсов, по крайней мере, оно должно быть доступно в облачной среде (в виде общей ФС) и должна быть возможность удобного подключения хранилища к конкретному облачному сервису. Также необходимо предусмотреть возможность работы с хранилищем на ПК пользователя. В настоящее время подобной системы организации облачного хранилища не существует.

В следующем подразделе приводится классификация HPC-приложений, позволяющая выделить классы приложений, которые могут быть эффективно перенесены в облако.

2.1 *Классификация HPC приложений и переносимость в облако*

Можно выделить несколько классов HPC-приложений в соответствии с их требованиями к вычислительной среде. В работе [13] существующие HPC-приложения классифицируются по характеру используемых ресурсов: сильносвязанные вычисления большого масштаба, вычисления среднего масштаба и высокомощные вычисления (high throughput computing). Для каждого из классов можно выделить основные плюсы от переноса в облако.

Сильносвязанные широкомасштабные вычисления подразумевают запуск MPI-задач на суперкомпьютерах с использованием большого числа процессоров (порядка тысяч и более ядер). Такие задачи требуют значительного времени для завершения и, как правило, запускаются посредством системы очередей задач. Суперкомпьютерные центры предоставляют для таких задач архивное хранилище и параллельные ФС. Такие задачи, как правило, показывают существенное снижение производительности в облаке.

Сильносвязанные вычисления среднего масштаба подразумевают запуск на ресурсах объемом 10-1000 вычислительных ядер. Для запуска таких задач чаще используются небольшие кластеры, обслуживаемые самими

научными группами. Данный класс задач хорошо подходит для переноса в облако.

Высокомощными называют задачи, которые подразумевают выполнение асинхронных независимых вычислений. Такие задачи часто связаны с пред- и пост-обработкой данных, и во многих случаях производятся на ПК исследователя. Однако в последние годы объем научных данных неуклонно растет (в частности, происходит переход к концепции BigData), и стадии пред- и пост-обработки также выполняются на высокопроизводительных вычислителях. Высокомощные задачи очень хорошо подходят для переноса в облако, однако требуют большой пропускной способности и значительного объема хранилища.

При разработке системы ВСК основной упор делался на приложения, относящиеся к классу сильносвязанных вычислений среднего масштаба, поскольку именно такие приложения требуют достаточно высокой производительности системы виртуализации и при этом могут эффективно выполняться в виртуальной среде. Высокомощные задачи решаются и в существующих облачных системах.

В следующем подразделе приводится обзор перспективных подходов к переносу HPC в облако, разрабатываемых в мире.

2.2 *Перспективные подходы к переносу HPC в облако*

В настоящее время исследователи рассматривают несколько основных задач, связанных с переносом HPC в облако.

Как правило, исследование облачных систем требует значительного объема вычислительных ресурсов. Исследовательскими организациями развернуты тестовые платформы, позволяющие оценить существующие и перспективные технологии. В США разрабатывается проект FutureGrid [14], который представляет собой программно-аппаратную платформу, объединяющую несколько удаленных вычислительных кластеров и предназначенную для проведения экспериментов в области облачных и грид вычислений. Общий объем задействованных ресурсов составляет около 5600 вычислительных ядер. Платформа поддерживает запуск как в виртуализированной среде, так и на реальном оборудовании. Особый упор делается на возможность сравнить различные технологии с точки зрения как производительности, так и удобства использования, а также на воспроизводимость экспериментов. В качестве облачных платформ доступны системы Nimbus и Eucalyptus.

Особенностью платформы FutureGrid является динамическое выделение ресурсов в зависимости от текущих запросов пользователей. В частности, возможно динамическое переконфигурирование серверов для решения HPC-задач в кластере либо для предоставления облачных ресурсов (виртуальных машин), причем поддерживается создание различных специфичных окружений для решения задач в конкретной области. Для переконфигурирования используются системы управления

аппаратными ресурсами xCAT [3] и Moab [4]. Пользователям платформы доступны команды для создания конкретных окружений, например, Hadoop-кластера. Фактически, пользователю предоставляется облачный сервис уровня платформы, реализованный с использованием виртуализации на уровне оборудования.

Особое внимание в проекте FutureGrid уделяется воспроизводимости экспериментов. Каждый эксперимент определяется как совокупность образа и последовательности определенных шагов. Образ в системе может быть использован как для запуска в облаке, так и для запуска на реальном сервере. Образ специально подготавливается для интеграции в систему FutureGrid. В качестве средства конфигурирования ПО образа применяется система BCFG2 [15], клиент которой интегрируется в каждый образ. Это позволяет пользователю при генерации образа указать требуемый набор ПО.

Другим известным проектом по анализу применимости облачных систем для решения HPC-задач стал проект Magellan [13]. В рамках данного проекта, выполненного в США по заказу Министерства энергетики (Department of Energy), проведено исследование облачных систем Eucalyptus, OpenStack, а также открытой платформы Hadoop. В качестве тестового стенда для оценки облачных технологий использовались HPC ресурсы общим объемом порядка 10000 ядер.

В настоящее время также рассматриваются вопросы использования существующих облачных систем уровня инфраструктуры для расширения вычислительного поля задачи по запросу. В данном случае речь идет о специализированных сервисах уровня платформы, позволяющих использовать преимущества облачных вычислений для повышения качества сервиса.

В работе [16] авторы описывают облачную систему, позволяющую объединять вычислительные ресурсы уровня инфраструктуры в кластеры различного масштаба, при этом поддерживается гибкость (эластичность) системы, то есть возможность динамически изменять объем задействованных в кластере серверов (ВМ). Система основана на облачном планировщике EPU (Elastic Processing Unit) [17] и для масштабирования производительности может использовать ресурсы проекта FutureGrid, а также сервиса Amazon EC2.

Для осуществления обратной связи выполнена интеграция с планировщиком задач Torque [18]. В планировщик встроен сенсор, позволяющий отслеживать текущие запросы ресурсов приложениями (в частности, отслеживается размер очереди задач). Для обработки данных сенсора реализован механизм принятия решений, использующий заданные администратором политики. По результатам работы механизма возможен запуск или остановка ВМ с использованием удаленных облачных ресурсов.

Для ускорения базовой конфигурации ВМ авторы используют фреймворк Chef [19], позволяющий автоматизировать действия по

системному администрированию ВМ, вследствие чего возможно использование практически любого базового образа ВМ на основе Linux, доступного в конкретном облаке.

Отдельным компонентом системы является брокер контекста, позволяющий интегрировать вновь запущенные ВМ в вычислительную среду, а также отсоединить удаляемые ВМ. Брокер, в частности, обеспечивает обмен IP-адресами и SSH-ключами в кластере. Брокер при этом выполняется непрерывно, что позволяет динамически изменять размер вычислительного кластера.

В целом, в качестве особенностей системы можно выделить независимость от конкретного поставщика облачного сервиса, а также независимость от конкретного параллельного приложения.

В работе [20] этой же группы исследователей рассматривается проблема ненадежности удаленных облачных ресурсов и каналов доступа к ним. Авторы описывают архитектуру сервиса, ключевой особенностью которого является поддержка высокой доступности (high availability) ресурсов и самого сервиса. Сервис использует облачные ресурсы различных провайдеров, что позволяет снизить риски при сбое одного из провайдеров. При этом концепция сервиса подразумевает, что выполняемая задача является слабосвязанной и возобновляемой при сбое конкретной ВМ. Таким образом, накладываются существенные ограничения на модель программирования, подразумевается наличие упорядоченных и структурированных подзадач, которые могут независимо выполняться в ВМ.

Отдельным направлением исследований является адаптация существующих приложений к облачной среде либо разработка специализированных моделей программирования для облачных вычислений.

В работе [21] авторы рассматривают особенности переноса научного приложения в облачную систему, построенную на основе ресурсов проектов FutureGrid, Magellan, а также облака Amazon EC2. Рассматриваются приложения, относящиеся к классу слабосвязанных параллельных приложений. Такие приложения могут быть сравнительно эффективно выполнены в облачной среде за счет невысоких требований к производительности и пропускной способности вычислительной сети.

Авторы рассматривают научную задачу в концепции процесса (workflow), описывающего последовательности вызываемых программ и их входных и выходных данных в виде направленного графа. Для выполнения процесса необходимо выделить ресурсы, отобразить процесс (вызываемые программы) на выделенные ресурсы, выполнить задачу, и освободить ресурсы.

Отображение процесса на вычислительные ресурсы может выполняться с использованием различных подходов. Авторы применяют для этого фреймворк Pegasus [22]. Фреймворк позволяет преобразовать абстрактное описание процесса в набор описаний задач, пригодный для

выполнения. Далее задача решается с использованием нескольких независимых облачных систем. Такая концепция также известна под названием «Sky computing» [23].

С нашей точки зрения, в настоящее время наиболее актуален перенос HPC-приложений в облако без необходимости модификации самого приложения. В частности, объем существующих MPI-приложений настолько велик, что зачастую не представляется возможным каким-либо образом переписать приложение в новой модели программирования. При этом сервис уровня инфраструктуры является плохо применимым для научных вычислений, так как требует от пользователя умений в области системного администрирования ОС и кластера в целом. Таким образом, целесообразно развивать сервисы более высокого уровня, которые бы позволяли пользователю быстро получать доступ к необходимой вычислительной среде, будь то MPI кластер или конкретный прикладной пакет. В следующем разделе описывается архитектура системы ВСК, в которой решаются некоторые из проблем, возникающих при переносе HPC в облако.

3 Особенности архитектуры облачной системы ВСК

Система ВСК состоит из нескольких относительно независимых компонентов. Облачная платформа разработана на основе системы OpenStack.

Основными компонентами системы ВСК являются (см. рис. 1):

1) Система виртуализации. В рамках системы ВСК применяется полная виртуализация платформы с использованием аппаратных возможностей процессора. Более подробное описание системы виртуализации приведено в подразделе 3.1.

2) Система управления вычислительными ресурсами (контроллер облака) – OpenStack Compute (Nova). Отвечает за предоставление виртуальных машин посредством управления монитором ВМ, выделение и освобождение различных ресурсов (виртуальных сетей, хранилищ), реализацию API (включая Amazon EC2 API и собственный интерфейс OpenStack API). Более подробное описание контроллера облака приведено в подразделе 3.2.

3) Хранилище образов ВМ OpenStack Image Service (Glance). Предназначено для организации хранения данных образов ВМ в различных форматах и поддержания каталога информации о доступных образах ВМ. Данные могут храниться с использованием различных систем хранения.

4) Хранилище данных. Предоставляет объектное и блочное хранилище. Функциональность аналогична сервисам Amazon S3 (Simple Storage Service) и EBS (Elastic Block Storage).

   Объектное хранилище позволяет пользователю хранить данные произвольной природы с использованием удаленного веб-сервиса. В хранилище обеспечивается автоматическая репликация данных. Объектное хранилище может быть использовано для хранения данных образов.

   Блочное хранилище (Cinder) позволяет выделать блочные устройства, которые являются устойчивым хранилищем информации и могут быть подключены к ВМ. В то же время, основные диски ВМ не являются устойчивыми, после удаления ВМ основной диск также удаляется. Блочное устройство может быть одновременно подключено не более чем к одной ВМ.

   Как объектное, так и блочное хранилища реализованы с использованием распределенной ФС CEPH [24]. CEPH предоставляет ряд интерфейсов для доступа к данным, среди которых S3-совместимое API, API уровня блочного хранилища, библиотека для программного доступа. Таким образом, CEPH позволяет реализовать все необходимые сервисы хранения данных на основе единого распределенного хранилища, поддерживающего репликацию.

5) Система авторизации и аутентификации OpenStack Identity (Keystone). Система используется остальными компонентами для авторизации и аутентификации различных агентов (пользователей, сервисов).

6) Пользовательский веб-интерфейс, основанный на системе OpenStack Dashboard (Horizon). Интерфейс переработан с учетом специфики решаемой задачи, выполнен перевод на русский язык.

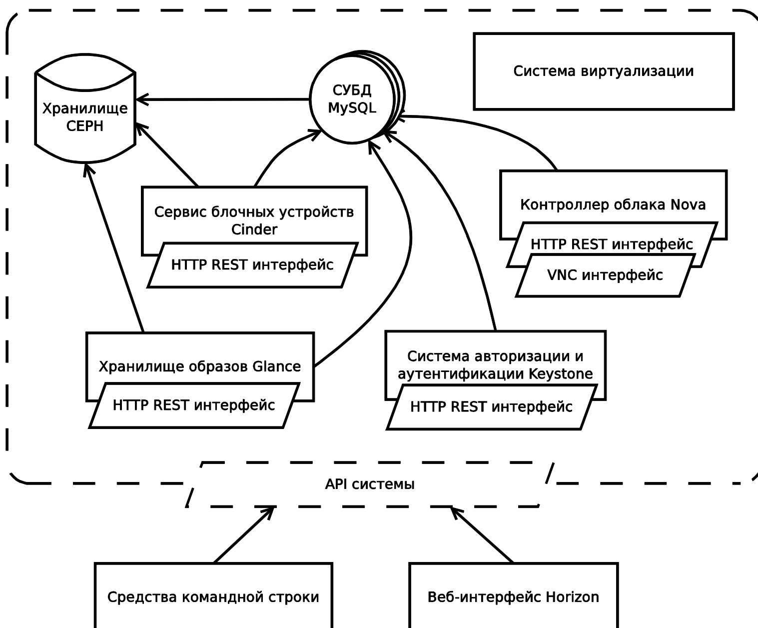


Рис. 1. Компоненты системы ВСК.

Компоненты Nova, Keystone, Glance, Cinder используют для работы БД SQL. В системе ВСК используется СУБД MySQL. Каждый из перечисленных компонентов работает с БД независимо. При этом для хранения БД также используется реплицируемое хранилище CEPH, что повышает надежность БД.

Компоненты Nova, Keystone, Glance, Cinder реализуют HTTP REST интерфейсы, с использованием которых осуществляется взаимодействие между ними. Данные интерфейсы также используются системами взаимодействия с пользователем (веб-интерфейс Horizon, средства командной строки).

Недостатком системы ВСК является отсутствие интегрированной в облако концепции высокопроизводительной общей ФС. Как известно, существующие облачные системы не имеют средств для предоставления и управления такими ФС. Более того, на основе выделяемых пользователю ресурсов практически невозможно организовать общую ФС достаточной производительности. В рамках системы ВСК данная проблема обходится путем организации независимой от системы общей ФС, используемой специализированными образами ВМ. С точки зрения концепции облачных вычислений, такое решение не является удобным, масштабируемым, безопасным, поскольку в рамках облачной системы все используемые ресурсы должны находиться под управлением самой системы. Таким образом, в настоящее время существует потребность в

специализированных сервисах, позволяющих предоставлять ВМ доступ к высокопроизводительным ФС в рамках облачной платформы.

3.1 *Система виртуализации*

Наиболее распространенные системы виртуализации в целом обладают одинаковыми возможностями. Такие системы используют доступные на сервере технологии аппаратной виртуализации. Среди систем с открытым исходным кодом, наиболее популярны KVM (Kernel-based Virtual Machine) [25] и гипервизоры, основанные на Xen [26]. В данной работе используется гипервизор KVM, который, на наш взгляд, более перспективен и удобен в использовании.

Гипервизор KVM реализует виртуальный процессор и систему памяти, остальные компоненты виртуальной машины реализованы эмулятором QEMU [27]. QEMU управляет ресурсами и предоставляет виртуальные устройства. С помощью QEMU можно запускать ВМ с использованием технологий аппаратной виртуализации (для этого используется KVM) либо бинарной трансляции (генератор кода встроен в QEMU). Для взаимодействия с гипервизором используется библиотека libvirt.

3.1.1 Обеспечение производительности ВМ

Основные полученные результаты в области производительности системы виртуализации на платформе x86 описаны в работах [28-31].

Можно выделить следующие этапы и методы улучшения производительности, разработанные в ходе выполнения проекта:

1) Выделение всех доступных ресурсов ВМ. В частности, выделение всех процессорных ядер, большей части памяти ВМ.
2) Обеспечение корректного отображения ресурсов сервера в ВМ. Необходимо правильно отобразить топологию системы (SMP или NUMA) в ВМ для того, чтобы гостевая ОС могла оптимально управлять процессами и памятью.
3) Настройка основной ОС, гостевой ОС для улучшения производительности приложений. Для выявления накладных расходов на виртуализацию необходимо добиться эффективной работы конкретных приложений.
4) Оптимизация производительности ввода-вывода. Требуется передать коммуникационное устройство в ВМ, что ведет к дополнительным накладным расходам.

Для тестирования применялись пакеты NAS Parallel Benchmarks [32], HPC Challenge [33], SPEC MPI2007 [34]. В целом, достигнут заданный уровень накладных расходов – не более 10% (на большинстве тестов в пределах 6%) при использовании до 1024 процессорных ядер. При этом в ходе работ были выявлены характеристики приложения, влияющие на производительность в ВМ. Среди таких характеристик:

1) Характер коммуникаций. Определяется частотой, объемом, типом коммуникаций. При виртуализации накладные расходы при «мелкозернистых» коммуникациях существенно повышаются. Синтетические тесты, содержащие частые вызовы групповых операций MPI, могут приводить к накладным расходам от десятков до сотен процентов, в зависимости от объема посылок и частоты коммуникаций. Это связано с возрастающим влиянием шума основной ОС на производительность. Проблема влияния шума ОС на производительность параллельного приложения рассматривается в статье [35].
2) Особенности работы с вводом-выводом данных. В рамках данной работы исследовались приложения, не предъявляющие высоких требований к вводу-выводу данных, в том числе к общей ФС. В качестве общей ФС использовалась система NFS, предоставляемая отдельным сервером. Данная ФС подключалась к вычислительным серверам посредством сети Ethernet, в случае виртуальных машин ряд компонентов такой сети (сетевой адаптер ВМ, сетевой мост) реализованы программным путем. Это приводит к дополнительным накладным расходам, которые, однако, не оказали существенного влияния на исследуемые приложения. При рассмотрении приложений, осуществляющих большие объемы взаимодействия с хранилищем, в том числе параллельный ввод-вывод, необходимо использовать файловую систему, реализованную на основе сети высокой пропускной способности (например, Fibre Channel, Infiniband). В таком случае могут возникнуть проблемы, связанные с эффективным предоставлением коммуникационного адаптера виртуальной машине.

3.2 *Контроллер облака*

Система Nova построена в концепции «неразделяемых ресурсов» (shared-nothing architecture) с использованием модели передачи сообщений. За счет этого, все основные компоненты системы Nova могут выполняться на отдельных серверах. Взаимодействие происходит с использованием удаленного вызова процедур (RPC, remote procedure call), реализованного на базе протокола AMQP [36]. Данные системы хранятся в БД SQL, используемой всеми компонентами Nova. БД обновляется при выполнении каждой операции.

Основными компонентами (сервисами в смысле режима выполнения) контроллера облака Nova являются (см. *Рис. **2***):

1) Сервис nova-api. Обрабатывает API-запросы от пользователей. Поддерживается несколько API, включая Amazon EC2,

OpenStack API. Инициирует большую часть операций в облаке, а также выполняет проверку квот. Реализация имеет модульную структуру, что позволяет предоставлять только необходимые API.

2) Сервис nova-compute. Данный сервис получает запросы на управление ВМ из очереди сообщений и взаимодействует с конкретным гипервизором на вычислительном узле. Для взаимодействия используется API, специфичное для гипервизора. Поддерживаются все наиболее распространенные гипервизоры (XenServer/XCP, KVM/QEMU, LXC, VMWare ESXi, Hyper-V), а также виртуализация на уровне аппаратуры.
3) Сервис nova-schedule. Планировщик ВМ и хранилища. Определяет, на каком сервере должна быть запущена ВМ и где должно быть выделено блочное хранилище в соответствии с заданными политиками. При разработке системы ВСК для учета специфики высокопроизводительных вычислений был разработан специальный планировщик, учитывающий топологию вычислительной сети. Более подробно см. [37].
4) Сервис nova-network. Процесс, отвечающий за организацию виртуальных сетей для групп ВМ. nova-network поддерживает несколько базовых моделей организации сетей; в рамках проекта OpenStack также разрабатывается компонент Quantum, предоставляющий сервис по управлению виртуальными сетями различной сложности.

   В рамках системы ВСК каждая группа ВМ объединена приватной сетью, к такой сети нельзя получить доступ извне. При этом также поддерживается концепция публичных IP-адресов. Публичный адрес может быть выделен по запросу пользователя из пула доступных адресов и назначен конкретной ВМ.
5) Поддержка удаленного доступа к ВМ по протоколу VNC. В рамках системы ВСК обеспечивается сервисами nova-consoleauth и nova-novncproxy. Сервис nova-novncproxy обеспечивает проксирование между компонентом noVNC, обеспечивающим отображение дисплея в веб-интерфейсе, и VNC-сервером на конкретном сервере ВМ. Сервис nova-consoleauth предназначен для авторизации доступа к дисплею ВМ.
6) Сервис nova-conductor. Сервис отвечает за взаимодействие с БД. Основная задача сервиса – ограничить взаимодействие других компонентов (прежде всего nova-compute) с БД. Это необходимо для повышения безопасности (так как nova-compute является наиболее уязвимым компонентом) и для упрощения обновления системы.

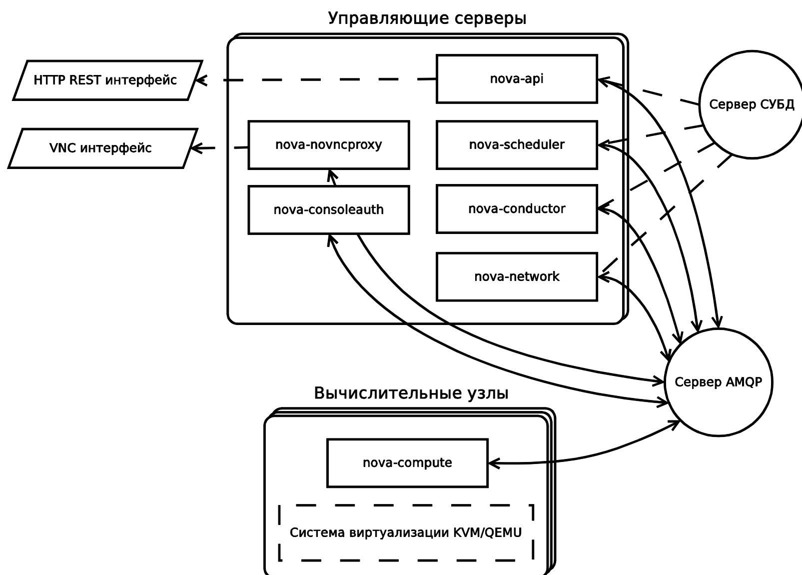


Рис. 2. Архитектура облачного контроллера Nova.

На рис. 2 описаны связи отдельных сервисов облачного контроллера Nova. Как видно, на вычислительных узлах достаточно запуска только сервисов nova-compute, что способствует снижению шума при виртуализации. За счет использования протокола AMQP отдельные сервисы могут быть запущены на произвольных серверах, связанных сетью. Кроме того, число процессов конкретного сервиса может быть любым, что позволяет при необходимости масштабировать производительность отдельных сервисов.

3.2.1 Особенности контроллера, связанные с HPC

В рамках сервиса уровня инфраструктуры основной единицей конфигурации ПО облачного сервера является образ ВМ (image), содержащий в себе данные диска, пригодные для запуска сервера. Образ может содержать предустановленную ОС и ПО. Образ используется для инициализации данных диска запускаемого сервера.

При создании облачного сервера недостаточно наличия только образа, который является только шаблоном и требует дополнительной конфигурации. Облачные системы поддерживают средства дополнительной настройки сервера, некоторые исследователи также используют понятие контекстуализации [38]. Среди необходимых настроек – публичные ключи для осуществления удаленного доступа к серверу. Для передачи настроек в ВМ может использоваться встраивание файлов в диск сервера до запуска (при этом облачная

система должна учитывать особенности конкретной ОС, установленной в образ, и используемой ФС). Кроме того, может быть реализована поддержка специального сервера метаданных либо дополнительного конфигурационного диска сервера (при этом внутри образа должен быть предустановлен агент облачной системы, который может взаимодействовать с такими источниками информации).

В системе ВСК сервис по предоставлению MPI-кластера реализован с использованием специального образа ВМ. После запуска группы ВМ из такого образа, достаточно выбрать какую-либо ВМ в качестве управляющего узла и запускать задачи командой mpirun. Единственное, что необходимо сделать пользователю – подготовить файл с IP-адресами серверов и передать его как параметр команде mpirun.

Концепция групп ВМ была специально разработана для учета специфики высокопроизводительных вычислений. Каждая группа ВМ при запуске планируется как единое целое, что позволяет планировщику располагать ВМ из одной группы оптимальным способом. Для задания характеристик оборудования облачного сервера используется концепция типа ВМ (flavor), позволяющего описать аппаратные требования сервера, например, частоту и число ядер процессора, объем ОЗУ, основного диска. Каждой группе ВМ ставится в соответствие конкретный тип ВМ, что обеспечивает запуск каждой ВМ группы с использованием одного и того же объема ресурсов и обеспечивает однородность виртуального кластера (см.рис. 3).

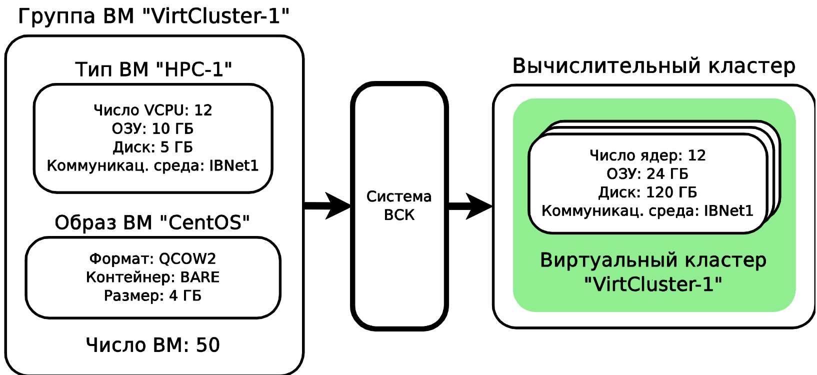


Рис. 3. Концепция Групп ВМ, запуск виртуального кластера.

Концепция типа ВМ была дополнена для обеспечения возможности создания виртуальных машин высокой производительности. Добавлена возможность задавать посредством типа ВМ конфигурацию топологии ВМ (например, параметры NUMA), задавать тип страниц для выделения памяти ВМ.

Также была реализована поддержка проброса устройств сервера в ВМ на основе интерфейса библиотеки libvirt. Проброс устройств необходим для

эффективной работы коммуникационной среды в группе ВМ. На каждом сервере необходимо указать пробрасываемые устройства в конфигурации облачного контроллера Nova. Каждое устройство имеет метку, указывающую его принадлежность к конкретной группе устройств (например, вычислительной сети). Для запроса устройства в ВМ необходимо создать новый тип ВМ, где указывается требуемый для запуска набор меток устройств.

Для конфигурации сетей в рамках системы ВСК используются предустановленные настройки. Виртуальные машины каждой группы объединены в частную сеть, не имеющую доступа в сеть Интернет. Для доступа к сети Интернет облачная система позволяет отдельно назначить публичный IP-адрес конкретной ВМ.

3.3 Особенности архитектуры сервиса OpenFOAM

Разработка и реализация сервиса OpenFOAM была выполнена в целях демонстрации возможностей концепции облачных вычислений в применении к высокопроизводительным задачам. Сервис позволяет решать набор задач в области механики сплошной среды.

Сервис OpenFOAM предоставляет пользователю дополнительную вкладку в веб-интерфейсе, на которой доступны операции по управлению задачами. Каждая задача определяется конкретными входными данными (OpenFOAM case), заданными пользователем. Также параметрами задачи являются метод разбиения сетки, используемый решатель, параметры решателя, число используемых процессов для решения. Число процессов определяет объем создаваемого при запуске виртуального кластера.

После задания входных данных, все стадии выполнения задачи происходят автоматически. Во время работы можно отслеживать текущее состояние задачи в веб-интерфейсе. После окончания выполнения, пользователь может загрузить результаты и журналы работы на свой ПК.

При реализации сервиса были использованы API, предоставляемые системой. При этом все данные о задачах хранятся в объектном хранилище, доступном посредством Swift API. При запуске задачи на управляющем сервере создается процесс, обеспечивающий выполнение жизненного цикла задачи.

Заключение

В данной работе описаны основные проблемы, возникающие при переносе высокопроизводительных вычислений в облако. Ключевой проблемой является падение производительности при использовании виртуализации. В ходе работы были выявлены основные источники накладных расходов и разработаны методы повышения

производительности ВМ. Также существенной проблемой является предоставление общей ФС, подходящей для высокопроизводительного (в том числе параллельного) ввода-вывода в рамках облачной системы.

Рассматривается подход к организации высокопроизводительного облачного сервиса с использованием виртуализации. Описана архитектура разработанной и реализованной облачной системы «виртуальный суперкомпьютер», основанной на платформе OpenStack и системе виртуализации KVM/QEMU. Компоненты системы ВСК доработаны для учета специфики высокопроизводительных вычислений, в частности, выполнена доводка и настройка системы виртуализации, что позволило достичь уровня накладных расходов не более 10% (не более 6% для большинства тестов) при использовании, по крайней мере, 1024 процессорных ядер. В рамках платформы OpenStack выполнена разработка и реализация необходимых концепций, в частности, добавлена возможность передачи устройства сервера в ВМ, а также возможность объединения виртуальных машин в группы для совместного планирования. Разработан специализированный планировщик ВМ, учитывающий топологию «толстое дерево» сети Infiniband.

В рамках разработанной системы ВСК предоставляется доступ к сервису уровня инфраструктуры, на базе которого реализован сервис OpenFOAM, демонстрирующий возможность работы с пакетом OpenFOAM как с облачным приложением.

В перспективе при разработке сервисов, ориентированных на применение в области HPC, необходимо, прежде всего, обеспечивать удобство решения задач для пользователя путем создания сервисов уровня платформы и приложений. Производительность может достигаться с использованием виртуализации платформы и существующих наработок, либо с использованием виртуализации на уровне оборудования. Кроме того, необходимо обеспечить пользователю единую среду для хранения данных, доступную как в облачной инфраструктуре, так и на компьютере пользователя. При этом такая среда должна обладать достаточной производительностью для решения задач пользователя. В целом, все предоставляемые сервисы должны быть объединены в единую экосистему, удобную для пользователя.

Список литературы

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[2] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Technical report, July 2009.

[3] xCAT – Extreme Cloud Administration Toolkit. http://xcat.sourceforge.net/, 1999. [Online; accessed 14-May-2013].

[4] Moab Suite. http://www.adaptivecomputing.com/products/. [Online; accessed 14-May-2013].

[5] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pages 124–131, 2009.

[6] Katarzyna Keahey, Ian Foster, Tim Freeman, and Xuehai Zhang. Virtual workspaces: Achieving quality of service and quality of life in the grid. *Scientific Programming*, 13(4):265–275, 2005.

[7] J Fontán, T Vázquez, L Gonzalez, Ruben S Montero, and IM Llorente. Opennebula: The open source virtual machine manager for cluster computing. In *Open Source Grid and Cluster Software Conference*, 2008.

[8] Apache CloudStack: Open Source Cloud Computing. http://-cloudstack.apache.org/, 2010. [Online; accessed 14-May-2013].

[9] OpenStack Open Source Cloud Computing Software. http://-www.openstack.org/, 2010. [Online; accessed 14-May-2013].

[10] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, 2006. [Online; accessed 14-May-2013].

[11] Windows Azure: Microsoft's Cloud Platform. http://-www.windowsazure.com/en-us/, 2010. [Online; accessed 14-May-2013].

[12] Google Compute Engine. https://cloud.google.com/products/compute-engine, 2012. [Online; accessed 14-May-2013].

[13] Lavanya Ramakrishnan, Piotr T. Zbiegel, Scott Campbell, Rick Bradshaw, Richard Shane Canon, Susan Coghlan, Iwona Sakrejda, Narayan Desai, Tina Declerck, and Anping Liu. Magellan: experiences from a science cloud. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, ScienceCloud '11, pages 49–58, New York, NY, USA, 2011. ACM.

[14] G. von Laszewski, G.C. Fox, Fugang Wang, A.J. Younge, A. Kulshrestha, G.G. Pike, W. Smith, J. Vöckler, R.J. Figueiredo, J. Fortes, and K. Keahey. Design of the futuregrid experiment management framework. In *Gateway Computing Environments Workshop (GCE), 2010*, pages 1–10, 2010.

[15] Bcfg2 – A Configuration Management System. http://trac.mcs.anl.gov/-projects/bcfg2/, 2004. [Online; accessed 14-May-2013].

[16] Paul Marshall, Henry Tufo, Kate Keahey, David LaBissoniere, and H.M. Woitaszek. Architecting a large-scale elastic environment – recontextualization and adaptive cloud services for scientific computing. In *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT)*, Rome, Italy, 2012.

[17] CEI Elastic Processing Unit (EPU) Services and Agents. https://github.com/-ooici/epu, 2011. [Online; accessed 14-May-2013].
[18] TORQUE Resource Manager. http://www.adaptivecomputing.com/products/-open-source/torque/, 2003. [Online; accessed 14-May-2013].
[19] Chef Configuration Management. http://www.adaptivecomputing.com/-products/, 2009. [Online; accessed 14-May-2013].
[20] Kate Keahey, Patrick Armstrong, John Bresnahan, David LaBissoniere, and Pierre Riteau. Infrastructure outsourcing in multi-cloud environment. In *Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit*, FederatedClouds '12, pages 33–38, New York, NY, USA, 2012. ACM.
[21] Jens-Sönke Vöckler, Gideon Juve, Ewa Deelman, Mats Rynge, and Bruce Berriman. Experiences using cloud computing for a scientific workflow application. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, ScienceCloud '11, pages 15–24, New York, NY, USA, 2011. ACM.
[22] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.
[23] K. Keahey, M. Tsugawa, A. Matsunaga, and J. A B Fortes. Sky computing. *Internet Computing, IEEE*, 13(5):43–51, 2009.
[24] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.
[25] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.
[26] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.
[27] Fabrice Bellard. Qemu, a fast and portable dynamic translator. USENIX, 2005.
[28] А. О. Кудрявцев, В. К. Кошелев, А. И. Аветисян. Перспективы виртуализации высокопроизводительных систем архитектуры x64. *Труды Института системного программирования РАН, том 22*, с. 189–209, 2012.
[29] Alexander Kudryavtsev, Vladimir Koshelev, and Arutyun Avetisyan. Modern HPC cluster virtualization using KVM and palacios. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–9, 2012.
[30] Alexander Kudryavtsev, Vladimir Koshelev, Boris Pavlovic, and Arutyun Avetisyan. Virtualizing HPC applications using modern hypervisors. In *Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit*, FederatedClouds '12, pages 7–12, New York, NY, USA, 2012. ACM.
[31] А.О. Кудрявцев, В.К. Кошелев, А.О. Избышев, А.И. Аветисян. Высокопроизводительные вычисления как облачный сервис: ключевые

проблемы. *Параллельные вычислительные технологии (ПаВТ'2013): труды международной научной конференции*, с. 432–438, 2013.
[32] Rob F Van der Wijngaart and Parkson Wong. Nas parallel benchmarks version 2.4. Technical report, NAS technical report, NAS-02-007, 2002.
[33] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213. Citeseer, 2006.
[34] Matthias S Müller, Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C Brantley, Chris Parrott, Tom Elken, et al. Spec mpi2007—an application benchmark suite for parallel systems using mpi. *Concurrency and Computation: Practice and Experience*, 22(2):191–205, 2010.
[35] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 19:1–19:12, Piscataway, NJ, USA, 2008. IEEE Press.
[36] S. Vinoski. Advanced Message Queuing Protocol. *Internet Computing, IEEE*, 10(6):87–89, 2006.
[37] И.А. Дудина, А.О. Кудрявцев. Разработка и реализация облачного планировщика, учитывающего топологию коммуникационной среды при высокопроизводительных вычислениях. *Труды Института системного программирования РАН, том 24*, принято к публикации, 2013.
[38] Katarzyna Keahey and Tim Freeman. Contextualization: Providing One-Click Virtual Clusters. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 301–308, Washington, DC, USA, 2008. IEEE Computer Society.

HPC cloud system design and implementation

Kudryavtsev A.O.: alexk@ispras.ru, ISP RAS, Moscow, Russia
Koshelev V.K.: vedun@ispras.ru, ISP RAS, Moscow, Russia
Izbyshev A.O.: izbyshev@ispras.ru, ISP RAS, Moscow, Russia
Dudina I.A.: eupharina@ispras.ru, ISP RAS, Moscow, Russia
Kurmangaleev Sh.F.: kursh@ispras.ru, ISP RAS, Moscow, Russia
Avetisyan A.I.: arut@ispras.ru, ISP RAS, Moscow, Russia
Ivannikov V.P.: ivan@ispras.ru, ISP RAS, Moscow, Russia
Velikhov V.E.: velikhovve@kiae.ru, NRC KI, Moscow, Russia
Ryabinkin E.A.: rea@grid.kiae.ru, NRC KI, Moscow, Russia

Abstract. There is pronounced interest to cloud computing in the scientific community. However, current cloud computing offerings are rarely suitable for high-performance computing, in large part due to an overhead level of underlying virtualization components. The purpose of this paper is to propose a design and implementation of a cloud system that possesses a small enough overhead level to allow it to be practically used for a wide range of scientific workloads. First, we describe requirements for the desired system and classify workloads to identify those that are practical to transfer to the cloud. Then, we review related work. Finally, we describe our cloud system, "Virtual Supercomputer", which is based on the OpenStack cloud infrastructure and KVM/QEMU hypervisor. Most components of the original infrastructure were modified to satisfy the requirements. In particular, we tuned KVM/QEMU and the host operating system, introduced the concept of virtual machine groups and implemented a topology-aware scheduler to reduce communication overhead between network nodes belonging to the same virtual machine group. Also, we implemented a proof-of-concept web service on top of our system that allows to use OpenFOAM toolbox in software-as-a-service manner. The main result of our work is that "Virtual Supercomputer" achieved the overhead level of less than 10% on industry standard benchmarks when using up to 1024 processor cores. We deem this overhead level as acceptable for practical use.

Keywords: cloud computing; virtualization; hypervisors; high-performance computing; parallel computing

References

[1]. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[2]. Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Technical report, July 2009.

[3]. xCAT – Extreme Cloud Administration Toolkit. http://xcat.sourceforge.net/, 1999. [Online; accessed 14-May-2013].

[4]. Moab Suite. http://www.adaptivecomputing.com/products/. [Online; accessed 14-May-2013].
[5]. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on, pages 124–131, 2009.
[6]. Katarzyna Keahey, Ian Foster, Tim Freeman, and Xuehai Zhang. Virtual workspaces: Achieving quality of service and quality of life in the grid. Scientific Programming, 13(4):265–275, 2005.
[7]. J Fontán, T Vázquez, L Gonzalez, Ruben S Montero, and IM Llorente. Opennebula: The open source virtual machine manager for cluster computing. In Open Source Grid and Cluster Software Conference, 2008.
[8]. Apache CloudStack: Open Source Cloud Computing. http://cloudstack.apache.org/, 2010. [Online; accessed 14-May-2013].
[9]. OpenStack Open Source Cloud Computing Software. http://www.openstack.org/, 2010. [Online; accessed 14-May-2013].
[10]. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, 2006. [Online; accessed 14-May-2013].
[11]. Windows Azure: Microsoft's Cloud Platform. http://www.windowsazure.com/en-us/, 2010. [Online; accessed 14-May-2013].
[12]. Google Compute Engine. https://cloud.google.com/products/compute-engine, 2012. [Online; accessed 14-May-2013].
[13]. Lavanya Ramakrishnan, Piotr T. Zbiegel, Scott Campbell, Rick Bradshaw, Richard Shane Canon, Susan Coghlan, Iwona Sakrejda, Narayan Desai, Tina Declerck, and Anping Liu. Magellan: experiences from a science cloud. In Proceedings of the 2nd international workshop on Scientific cloud computing, ScienceCloud '11, pages 49–58, New York, NY, USA, 2011. ACM.
[14]. G. von Laszewski, G.C. Fox, Fugang Wang, A.J. Younge, A. Kulshrestha, G.G. Pike, W. Smith, J. Vöckler, R.J. Figueiredo, J. Fortes, and K. Keahey. Design of the futuregrid experiment management framework. In Gateway Computing Environments Workshop (GCE), 2010, pages 1–10, 2010.
[15]. Bcfg2 – A Configuration Management System. http://trac.mcs.anl.gov/projects/bcfg2/, 2004. [Online; accessed 14-May-2013].
[16]. Paul Marshall, Henry Tufo, Kate Keahey, David LaBissoniere, and H.M. Woitaszek. Architecting a large-scale elastic environment – recontextualization and adaptive cloud services for scientific computing. In Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT), Rome, Italy, 2012
[17]. CEI Elastic Processing Unit (EPU) Services and Agents. https://github.com/ooici/epu, 2011.[Online; accessed 14-May-2013].
[18]. TORQUE Resource Manager. http://www.adaptivecomputing.com/products/open-source/torque/, 2003. [Online; accessed 14-May-2013].
[19]. Chef Configuration Management. http://www.adaptivecomputing.com/products/, 2009. [Online; accessed 14-May-2013].
[20]. Kate Keahey, Patrick Armstrong, John Bresnahan, David LaBissoniere, and Pierre Riteau. Infrastructure outsourcing in multi-cloud environment. In Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit, FederatedClouds '12, pages 33–38, New York, NY, USA, 2012. ACM.

[21]. Jens-Sönke Vöckler, Gideon Juve, Ewa Deelman, Mats Rynge, and Bruce Berriman. Experiences using cloud computing for a scientific workflow application. In Proceedings of the 2nd international workshop on Scientific cloud computing, ScienceCloud '11, pages 15–24, New York, NY, USA, 2011. ACM.
[22]. Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming, 13(3):219–237, 2005.
[23]. K. Keahey, M. Tsugawa, A. Matsunaga, and J. A B Fortes. Sky computing. Internet Computing, IEEE, 13(5):43–51, 2009.
[24]. Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In Proceedings of the 7th symposium on Operating systems design and implementation, pages 307–320. USENIX Association, 2006.
[25]. Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In Proceedings of the Linux Symposium, volume 1, pages 225–230, 2007.
[26]. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. ACM SIGOPS Operating Systems Review, 37(5):164–177, 2003.
[27]. Fabrice Bellard. Qemu, a fast and portable dynamic translator. USENIX, 2005.
[28]. Kudryavtsev A.O., Koshelev V.K., Avetisyan A.I. Perspektivy virtualizatsii vysokoproizvoditel'nykh sistem arkhitektury x64 [The Prospects for Virtualization of High Performance x64 Systems]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 189–209 (in Russian).
[29]. Alexander Kudryavtsev, Vladimir Koshelev, and Arutyun Avetisyan. Modern HPC cluster virtualization using KVM and palacios. In High Performance Computing (HiPC), 2012 19th International Conference on, pages 1–9, 2012.
[30]. Alexander Kudryavtsev, Vladimir Koshelev, Boris Pavlovic, and Arutyun Avetisyan. Virtualizing HPC applications using modern hypervisors. In Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit, FederatedClouds '12, pages 7–12, New York, NY, USA, 2012. ACM.
[31]. Kudryavtsev A.O., Koshelev V.K., Izbyshev A.O., Avetisyan A.I. Vysokoproizvoditel'nye vychisleniya kak oblachnyj servis: klyuchevye problemy [High Performance Computing as a Cloud Service: Key Issues]. Trudy mezhdunarodnoj nauchnoj konferentsii PaVT'2013 [The Proceedings of International Conference PCT'2013], 2013, pp. 432–438 (in Russian).
[32]. Rob F Van der Wijngaart and Parkson Wong. Nas parallel benchmarks version 2.4. Technical report, NAS technical report, NAS-02-007, 2002.
[33]. Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, page 213. Citeseer, 2006.
[34]. Matthias S Müller, Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C Brantley, Chris Parrott, Tom Elken, et al. Spec mpi2007—an application benchmark suite for parallel systems using mpi. Concurrency and Computation: Practice and Experience, 22(2):191–205, 2010.

[35]. Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC ’08, pages 19:1–19:12, Piscataway, NJ, USA, 2008. IEEE Press.
[36]. S. Vinoski. Advanced Message Queuing Protocol. Internet Computing, IEEE, 10(6):87–89, 2006.
[37]. Dudina I.A., Kudryavtsev A.O., Gaissaryan S.S. Razrabotka i realizatsiya oblachnogo planirovshhika, uchityvayushhego topologiyu kommunikatsionnoj sredy pri vysokoproizvoditel'nykh vychisleniyakh [Topology-aware cloud scheduling for HPC]. Trudy ISP RAN [The Proceedings of ISP RAS], 2013, vol. 24, pp. 189–209 (in Russian).
[38]. Katarzyna Keahey and Tim Freeman. Contextualization: Providing One-Click Virtual Clusters. In Proceedings of the 2008 Fourth IEEE International Conference on eScience, ESCIENCE ’08, pages 301–308, Washington, DC, USA, 2008. IEEE Computer Society.

Разработка и реализация облачного планировщика, учитывающего топологию коммуникационной среды при высокопроизводительных вычислениях[1]

И.А. Дудина, А.О. Кудрявцев, С.С. Гайсарян

Аннотация. Для эффективного решения высокопроизводительных задач в облаке необходимо планирование виртуальных машин на серверах, учитывающее особенности высокопроизводительных вычислений, чтобы уменьшить потери производительности, связанные с задержками в сети. В данной работе рассматривается подход к планированию, основанный на оценке размещения с помощью Hop-Byte метрики. Для конкретного варианта коммуникационной среды (сети Infiniband с топологией «толстое дерево») производится подсчет Hop-Byte метрики в предположении, что взаимодействие между виртуальными машинами одинаково. На основе полученных результатов разработан алгоритм планирования, выполнена его реализация в облачной системе OpenStack. Показано, что для отдельных тестов удается получить прирост производительности, при этом существенно снижается разброс значений при повторных запусках.

Ключевые слова: планирование, виртуализация, облачные вычисления, hop-byte метрика

Введение

Согласно проведенному министерством энергетики США исследованию Magellan User Survey [1] среди ученых, использующих в своей деятельности высокопроизводительные вычисления, возрастает интерес к облачным сервисам. Причиной тому является целый ряд уникальных преимуществ, присущих облачным вычислениям. К наиболее значительным из них можно отнести экономию от масштаба, гибкие условия аренды ресурсов, простоту их приобретения и использования,

[1] Работа выполнена при финансовой поддержке Минобрнауки России по государственному контракту от 15.06.2012 г. № 07.524.11.4018 в рамках ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы»

наличие у пользователей возможности самостоятельного управления ПО, а также удобный пользовательский интерфейс и возможность создания научного портала для организации совместной работы с коллегами. Однако ключевым препятствием для использования облачных систем в этой области является падение производительности приложений в облачной среде. Некоторые успехи в решении этой проблемы уже достигнуты, в частности уменьшение падения производительности, вызванного накладными расходами на виртуализацию [2].

Одним из факторов, ограничивающих производительность приложений не только в облачных системах, но и на суперкомпьютерах, является пропускная способность сети, обеспечивающей взаимодействие между процессами в рамках одной задачи. В то время как собственно вычисления дешевеют, узким местом при масштабировании параллельных приложений является взаимодействие между процессами. Учитывая особенности высокопроизводительных вычислений, даже незначительное снижение производительности может привести к сильному ухудшению при масштабировании числа узлов. Таким образом, для повышения производительности необходимо как можно более эффективно использовать ресурсы сети.

В настоящее время существует несколько облачных платформ, как закрытых, так и с открытым исходным кодом. К закрытым относятся: Amazon EC2, Microsoft Azure, Google Compute Engine. Среди открытых платформ наиболее известны системы OpenNebula, OpenStack, Nimbus, Eucalyptus. Наиболее активно развивающейся из них является OpenStack – комплекс проектов свободного программного обеспечения для создания вычислительных облаков и облачных хранилищ (распространяется под лицензией Apache License 2.0).

Целью данной работы являлась разработка алгоритма планировщика OpenStack для сокращения падения производительности, вызванного задержками сети. Одним из компонентов OpenStack является контроллер вычислительных ресурсов Nova, частью которого является планировщик, отвечающий за выбор узлов для запуска виртуальных машин. Задача заключается в том, чтобы модифицировать имеющийся планировщик с учетом требований для высокопроизводительных вычислений.

1 Обзор работ

В работе [3] были предприняты попытки повышения эффективности использования сети для высокопроизводительных вычислений. Здесь рассматриваются сети InfiniBand с иерархической структурой, организованные в несколько уровней с помощью коммутаторов. Идея оптимизации основана на том, что время на передачу сообщения тем меньше, чем меньше коммутаторов должно оно пройти. Разработанный

метод заключается в определении топологии сети и использовании полученных данных для оптимизации работы коллективных операций. Авторы проанализировали производительность операций MPI Gather и MPI Scatter на таких системах и разработали для них эффективный алгоритм, учитывающий топологию сети. Увеличение производительности на уровне микробенчмарков при использовании этих алгоритмов достигло почти 54%.

Продолжением этого исследования стала работа [4]. Авторы рассматривают случай сети, неоднородной по скорости, и предлагают учитывать это свойство сети наряду с топологией для оптимизации коллективных операций. В результате этого исследования был разработан фреймворк, автоматически определяющий топологию сети и скорость между узлами и предоставляющий эти данные пользователю. Кроме того, были внесены изменения в библиотеку MPI: добавлены возможности динамического получения информации о топологии и скорости сети, а также построения модели сети на основе этих данных; модифицирован алгоритм вещания для оптимизации с учетом имеющейся модели. Эксперименты показали, что для однородной по скорости сети и сообщений большого размера увеличение производительности коллективных операции на уровне микробенчмарков достигает 14%. На уровне приложения использование фреймворка позволяет сократить общее время работы почти на 8%, особенно при увеличении размера задачи. Увеличение производительности сетевых алгоритмов на неоднородной по скорости сети достигает 70%-100%.

Диссертация A. Bhatele [5] посвящена автоматическому размещению процессов на узлах суперкомпьютера с учетом топологии сети. В этой работе предлагается использовать для оценки загруженности сети hop-byte метрику вместо диаметра графа. Показано, что эта метрика более адекватна для поставленной задачи, чем метрики, использованные ранее другими авторами. Разработан фреймворк, позволяющий автоматически размещать процессы на узлах с учетом топологии. В первую очередь происходит определение графа коммуникаций и выделение повторяющихся образцов. Далее для регулярных и нерегулярных графов применяются эвристики, позволяющие найти удачное размещение с низким значением hop-byte метрики. Кроме того, в данной работе рассматриваются перспективы виртуализации высокопроизводительных вычислений, в частности, метод балансировки нагрузки с учетом топологии сети.

К сожалению, задача оптимизации hop-byte метрики является NP-трудной, т.к. сводится к квадратичной задаче о назначениях. В статье [6] приводится эвристика для этой задачи, позволяющая сократить значение этой метрики. Использование эвристики в библиотеке MPI, разрабатываемой фирмой Cray, позволило сократить значение метрики на 75%. Однако временная сложность этой эвристики выше по

сравнению с другими эвристиками для этой задачи, поэтому данный алгоритм разумно использовать в случае статичных, пусть и нерегулярных графов топологии и коммуникаций задачи.

Попытки доработать планировщик Openstack для высокопроизводительных вычислений были предприняты в статье [7]. Первая идея, реализованная в этой работе, заключается в том, что планировщик должен рассматривать размещение N процессов одной задачи на N узлах как единый запрос, а не N последовательных независимых запросов. Это позволяет сразу выбрать группу наиболее подходящих для этой задачи узлов. Далее, решая задачу размещения для группы из N сильно связанных процессов, планировщик выбирает наиболее незагруженную стойку, а в ней отдает предпочтение узлам с наименьшим количеством уже запущенных виртуальных машин. Вторая идея основана на том, что в высокопроизводительных задачах вычисления часто прерываются коллективными операциями, т.е. происходит периодическая синхронизация процессов. Вследствие этого все процессы выполняются со скоростью самого “медленного”. Поэтому для эффективного использования ресурсов разумно размещать всю группу из N процессов на одинаковых серверах.

2 Существующие возможности системы Openstack

Начиная с версии Essex, в OpenStack по умолчанию используется так называемый «фильтрующий» планировщик. Получая запрос на размещение нескольких виртуальных машин, он принимает решение о размещении на конкретных серверах в два этапа.

2.1 *Этап фильтрации*

Первый этап заключается в отсеивании всех серверов, не удовлетворяющих требованиям запроса. Для этого к списку серверов применяется набор фильтров, определенный в конфигурационном файле Nova. Каждый фильтр отсеивает сервера, не удовлетворяющие определенному требованию, и в результате для дальнейшего рассмотрения формируется список серверов, прошедших через все фильтры. Так, например, **ram_filter** исключает из рассмотрения сервера, не обладающие достаточной памятью, **disk_filter** — местом на дисковом пространстве и т. д.

Стоит отметить, что в случае высокопроизводительных вычислений для минимизации накладных расходов разумно размещать на каждом сервере не более одной виртуальной машины и предоставлять ей все ресурсы этого сервера. В нашей задаче будем считать, что среди прошедших фильтрацию серверов останутся только полностью незанятые.

2.2 *Этап взвешивания*

К началу этапа взвешивания планировщик имеет список серверов, на каждом из которых можно разместить запускаемые ВМ. На этом этапе необходимо среди всех допустимых серверов найти наиболее подходящий. Сравнение различных вариантов размещения необходимо проводить по многим факторам одновременно. Для этого каждому из факторов ставится в соответствие оценочная функция, возвращающая численное значение для рассматриваемого сервера, а также коэффициент, отражающий значимость этого фактора. Таким образом вычисляется вес сервера, который равен сумме произведений значений оценочных функций на соответствующие им коэффициенты:

$$W_{h_i} = \sum_{j=1}^{m} c_j f_j(h_i),$$

где h_i – i-ый сервер, W_{h_i} – его вес, f_j – оценочная функция, c_j – коэффициент, m – количество оценочных функций.

Далее серверы сортируются по весу, и для размещения очередной ВМ выбирается сервер с наибольшим весом. После этого этапы фильтрации и взвешивания повторяются, пока не будут размещены все ВМ из запроса [8].

3 Планирование высокопроизводительных задач

Задача планирования распределенных (в том числе параллельных) программ в рамках вычислительного кластера в целом аналогична задаче распределения ВМ по узлам кластера, при условии, что в виртуальных машинах выполняется какая-либо распределенная программа. В данном разделе описывается концепция групп ВМ, аналогичная набору процессов одного приложения. К задаче планирования групп ВМ применимы подходы, разработанные для планирования процессов распределенной программы. Описывается применение подхода, основанного на Hop-Byte метрике.

3.1 *Группы ВМ*

Для того чтобы при планировании учесть взаимодействие ВМ, планировщик должен решать проблему размещения ВМ, решающих конкретную задачу, ориентируясь на положение других ВМ этой задачи. Для того чтобы логически объединить виртуальные машины одной задачи, была введена концепция групп виртуальных машин в OpenStack. Каждая виртуальная машина может принадлежать какой-то группе (соответствующей определенной высокопроизводительной задаче). Таким образом, цель планировщика – разместить виртуальные машины

наиболее “компактно” – может быть поставлена в рамках одной группы виртуальных машин. Пришло время сформулировать критерий “компактности”, т.е. удачности конкретного варианта размещения виртуальных машин одной группы на определенных серверах.

3.2 Hop-Byte метрика

Эффективность конкретного варианта размещения процессов данного приложения на узлах сети зависит от двух параметров: топологии сети и характера взаимодействия процессов задачи между собой. Для оценки этой величины была предложена Hop-Byte (HB) метрика. В данной модели рассматриваются два графа.

Первый взвешенный неориентированный граф $G = (V, E)$ отражает топологию сети. Вершины данного графа соответствуют серверам. Каждое ребро между двумя вершинами i, j имеет вес e_{ij}, равный количеству скачков между двумя серверами, соответствующими данным вершинам (количество скачков равно количеству коммутаторов, которое проходит сообщение при передаче от одного узла к другому).

Второй взвешенный граф $G' = (V', E')$ отражает взаимодействие между процессами. Вершины этого графа соответствуют процессам рассматриваемого приложения. Ребро между двумя вершинами i, j имеет вес e'_{ij}, равный суммарному объему сообщений, посылаемых между этими вершинами в обоих направлениях.

Рассмотрим конкретное размещение процессов из V' на серверах из V:

$$x_{hp} = \begin{cases} 1, \text{если процесс } p \text{ размещен на сервере } h \\ 0, \quad \text{в противном случае} \end{cases}, \quad h \in V, p \in V';$$

$$HB = \sum_{i,j} \sum_{k,l} e_{ik} e'_{jl} x_{ij} x_{kl};$$

$$\sum_i x_{ij} = 1, \text{для всех } j \in V', \qquad \sum_j x_{ij} = 1, \text{для всех } i \in V.$$

Полученное значение **HB** и является показателем, отражающим успешность данного расположения процессов на узлах. Доказано, что задача минимизации этой величины является NP-трудной, т.к. сводится к квадратичной задаче о назначениях. Кроме этого, заранее о приложениях пользователей ничего неизвестно, поэтому необходимо прилагать дополнительные усилия для выяснения значений e'_{ij}.

Для решения поставленной задачи за приемлемое время можно её упростить, сделав некоторые дополнительные предположения. Во-

первых, при разработке алгоритма можно ориентироваться на какой-то конкретный тип топологии сети (предположения относительно графа ***G*** в нашей постановке). Во-вторых, можно отказаться от решения дополнительной весьма сложной задачи по выяснению значений e'_{ij}, приняв их за одинаковые константы (это соответствует ситуации, в которой все процессы равноправны и взаимодействуют между собой с одинаковой интенсивностью).

3.3 *Топология сети*

Мы рассматриваем топологию сети Infiniband «Fat tree», т.к. она является одной из распространенных топологий для высокопроизводительных задач. Такая сеть представляет собой дерево, листьями которого являются вычислительные узлы, а внутренними вершинами – коммутаторы (см. рис. 1).

Рис. 1. Топология Fat Tree

У коммутаторов более высоких уровней пропускная способность каналов больше, т.е. связи с другими вершинами более «толстые». Поэтому эта топология названа «толстое дерево»

3.4 *Алгоритм планирования*

С учетом сделанных предположений о равномерности коммуникаций между процессами и топологии сети можно предложить алгоритм планирования, ориентированный на минимизацию HB-метрики для каждой из групп виртуальных машин.

3.4.1 Применяемая эвристика

Из предположения о равномерном взаимодействии процессов одной задачи следует, что на значение HB-метрики влияет только суммарное значение скачков между всеми парами процессов. С другой стороны, количество скачков между процессами любой пары зависит только от того, с каким из коммутаторов нижнего уровня связан каждый из двух серверов, размещающих рассматриваемые процессы. Для удобства

можно объединить все сервера, связанные с определенным коммутатором нижнего уровня в один “пучок” (см. Рис. 2).

Рис. 2. Объединение серверов в пучки

Очевидно, что количество скачков между любой парой серверов зависит только от того, принадлежат ли эти серверы одному пучку. В связи с этим в планировщик была добавлена новая сущность, хранящая информацию о пучке, принадлежащих ему серверах, о числе виртуальных машин каждой группы, уже запущенных на этих серверах и т.д.

Итак, пусть планировщику поступил запрос на планирование N виртуальных машин одной группы, причем ранее уже запущено M машин этой группы. Задача планировщика разместить N машин так, чтобы значение HB-метрики для $(M+N)$ машин было как можно меньше. Пусть есть P пучков, и на них соответственно размещено $n_1, n_2, \dots, n_P$ машин (см. Рис. 3):

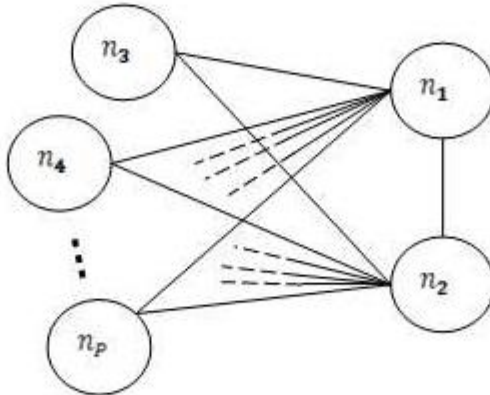


Рис. 3. Подсчет HB-метрики

$$\sum_{i=1}^{P} n_i = M + N.$$

Без ограничения общности выделим первые два пучка и посчитаем значение HB-метрики. Для начала посчитаем вклад взаимодействия внутри пучков (каждая связь – один скачок):

$$HB_{in} = \sum_{\substack{i=1..P,\\ n_i>0}} \frac{n_i(n_i-1)}{2}.$$

Пусть $n_1 + n_2 = s,\ s > 0$. Теперь посчитаем вклад взаимодействия виртуальных машин между пучками (каждое взаимодействие происходит с тремя скачками):

$$HB_{out} = 3\sum_{\substack{i,j=1..P,\\ i\neq j}} n_i n_j$$

$$= 3(\sum_{\substack{i,j=3..P,\\ i\neq j}} n_i n_j + n_1(n_3 + \cdots + n_P)$$

$$+ n_2(n_3 + \cdots + n_P) + n_1 n_2)$$

$$= 3(\sum_{\substack{i,j=3..P,\\ i\neq j}} n_i n_j + s(n_3 + \cdots + n_P) + n_1 n_2).$$

Отсюда видно, что от распределения s виртуальных машин между первым и вторым пучком зависят только слагаемые:

$$\frac{n_1(n_1-1)}{2} + \frac{n_2(n_2-1)}{2} + 3n_1 n_2$$

$$= \frac{n_1(n_1-1)}{2} + \frac{(s-n_1)(s-n_1-1)}{2}$$

$$+ 3n_1(s-n_1) =$$

$$-2n_1^2 + 2sn_1 + \frac{s^2-s}{2}, \text{где } n_1 > 0, n_2 > 0.$$

(В случае $n_2 = 0$ эта сумма становится одним слагаемым $\frac{n_1(n_1-1)}{2}$, случай $n_1 = 0$ аналогичен.)

Меняя значение n_1 можно перераспределить виртуальные машины между пучками так, чтобы минимизировать эту сумму. На **Рис. 4** показана зависимость значения этой суммы от значения n_1. Очевидно, что чем дальше n_1 (а, значит, и n_2) от $s/2$, тем меньше значение рассматриваемой суммы, и, следовательно, меньше значение HB-метрики в целом.

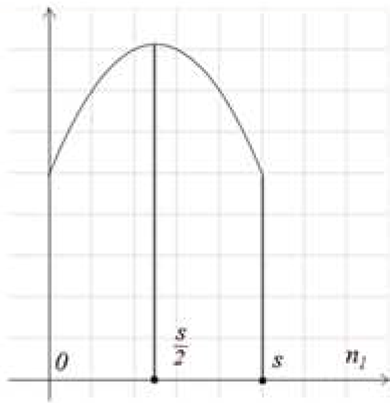


Рис. 4. Изменение значения НВ при перераспределении ВМ

Из этого можно можно предложить следующую стратегию: планировщик не должен заполнять пучки равномерно. Напротив, следует в первую очередь заполнять те из них, которые способны вместить наибольшее количество виртуальных машин данной группы (включая уже запущенные).

3.4.2 Взвешивающая функция

Было решено использовать имеющийся в OpenStack фильтрующий планировщик, дополнив его возможности, сохраняя общую логику планирования. Для того чтобы планировщик имел возможность учитывать особенности высокопроизводительных вычислений (а именно, высокие требования к производительности, зависящей от размещения виртуальных машин), была разработана взвешивающая функция, оценивающая размещение виртуальной машины определенной группы на отфильтрованных серверах-кандидатах. Гибкость данного подхода заключается в том, что он позволяет администратору изменять степень важности этого фактора, меняя коэффициент данной взвешивающей функции.

Значение оценочной функции для всех серверов одного пучка будет одинаковым. Принимая во внимание сделанные выше выводы, в первую очередь следует отсортировать пучки по максимальному числу виртуальных машин рассматриваемой группы, которое может оказаться в этом пучке после выполнения запроса к планировщику. Среди тех пучков, для которых это значение оказалось наибольшим, следует выбрать тот, в котором больше уже запущенных виртуальных машин этой группы. Для реализации сортировки сначала по первому, а затем по второму параметру, к значению второго параметра добавляется значение первого, умноженное на максимальное значение второго, увеличенное на единицу (см. *Листинг 1*).

```
first_ord = bunch.get_group_num(required_group) +
            min(bunch.free_hosts,
      num_instances_awaiting_shedule)
second_ord = bunch.get_group_num(required_group)
result = first_ord * (bunch.maxsize + 1) + second_ord
```

Листинг 1. Вычисление значения взвешивающей функции

4 Результаты тестирования

Для исследования влияния разработанного алгоритма планирования на производительность было проведено тестирование в два этапа. Во время первого запуска группы из 16 виртуальных машин планирование происходило средствами стандартного планировщика, и виртуальные машины были распределены произвольным образом между коммутаторами. Во время второго запуска планировщик учитывал разработанную взвешивающую функцию, и все виртуальные машины оказались в одном коммутаторе.

Исследование производительности было выполнено с использованием NAS Parallel Benchmarks. Это набор из 11 тестов производительности нацеленных на проверку возможностей вычислительной системы. Тесты разработаны и поддерживаются в NASA Advanced Supercomputing (NAS) Division, расположенном в NASA Ames Research Center [9].

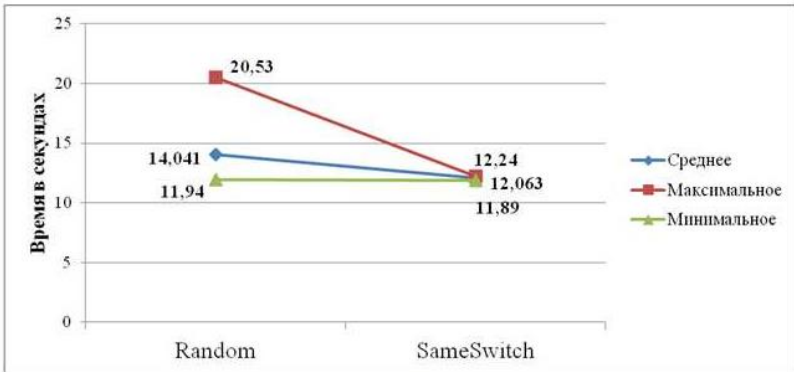


Рис. 5. Тест Block Tridiagonal.

Пакет тестов в обоих случаях запускался по 10 раз. Для 10 из 11 тестов не было выявлено существенных различий в производительности. Однако на тесте Block Tridiagonal среднее значение времени выполнения уменьшилось на 14% при планировании в рамках одного коммутатора, а максимальное значение уменьшилось на 40%. Амплитуда колебаний

производительности снизилась с 42% до 3%, т.е. колебания почти исчезли (см. Рис. 5).

Из этого можно сделать вывод, что разработанный алгоритм планирования для отдельных приложений позволяет предотвратить падение производительности, при этом производительность других приложений не ухудшается.

5 Выводы

Для эффективного решения высокопроизводительных задач в облаке необходимо планирование виртуальных машин на серверах, учитывающее особенности высокопроизводительных вычислений, чтобы уменьшить потери производительности, связанные с задержками в сети. В данной работе был рассмотрен подход к планированию, основанный на оценке размещения с помощью Hop-Byte метрики. Эта задача сводится к NP-трудной. Для конкретного варианта коммуникационной среды (сеть Infiniband с топологией «толстое дерево») был произведен подсчет Hop-Byte метрики в предположении, что взаимодействие между виртуальными машинами одинаково.

Основным вкладом данной работы является разработка алгоритма планирования на основе полученных результатов и его реализация в рамках общего подхода к планированию в OpenStack с использованием концепции групп виртуальных машин. Таким образом, в OpenStack была добавлена возможность планирования виртуальных машин с учетом требований высокопроизводительных вычислений.

Результаты проведенных исследований говорят о том, что разработанный в рамках планировщика OpenStack алгоритм планирования позволяет повысить производительность некоторых приложений. При этом падений производительности, связанных с использованием алгоритма, ни на каких тестах выявлено не было.

Список литературы

[1]. U.S. Department of Energy, «The Magellan Report on Cloud Computing for Science,» Chicago, 2011.

[2]. А. О. Кудрявцев, В. К. Кошелев и А. И. Аветисян, «Перспективы виртуализации высокопроизводительных систем архитектуры x64,» *Труды Института системного программирования РАН,* т. 22, pp. 189-210, 2012.

[3]. K. Kandalla, H. Subramoni, A. Vishnu и D. K. Panda, «Designing Topology-Aware Collective Communication Algorithms for Large Scale InfiniBand Clusters:Case Studies with Scatter and Gather,» в *The 10th Workshop on Communication Architecture for Clusters (CAC 10), Int'l Parallel and Distributed Processing Symposium (IPDPS 2010).*, Ohio, 2010.

[4]. H. Subramoni, K. Kandalla, J. Vienne, S. Sur, B. Barth, K. Tomko, R. McLay, K. Schulz и D. K. Panda, «Design and Evaluation of Network Topology-/Speed-Aware Broadcast Algorithms for InfiniBand Clusters,» 2012.

[5]. A. Bhatele, «Automatic Topology Aware Mapping For Supercomputers,» 2010.

[6]. C. D. Sudheer и A. Srinivasan, «Optimization of the Hop-Byte Metric for Effective Topology Aware Mapping,» 2012.
[7]. A. Gupta, D. Milojicic и L. V. Kalé, «Optimizing VM Placement for HPC in the Cloud,» в *Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit*, San Jose, CA, USA., 2012.
[8]. «Nova Developer guide. Filter Scheduler,» [В Интернете]. Available: http://docs.openstack.org/developer/nova/devref/filter_scheduler.html. [Дата обращения: 16 04 2013].
[9]. «Википедия,» [В Интернете]. Available: http://ru.wikipedia.org/wiki/NAS_Parallel_Benchmarks. [Дата обращения: 27 05 2013].

Topology-aware cloud scheduling for HPC

I.A. Dudina: eupharina@ispras.ru, ISP RAS, Moscow, Russia
A.O. Kudryavtsev: alexk@ispras.ru, ISP RAS, Moscow, Russia
S.S. Gaissaryan: ssg@ispras.ru, ISP RAS, Moscow, Russia

Abstract. For some compute intensive applications cloud computing can be a cost-effective alternative or an addition to supercomputers. However, in the case of high-performance computing, overall application performance depends heavily on how processes are mapped to the network nodes. Therefore a cloud scheduler must be topology-aware to reduce network congestion. In this paper the Hop-Byte metric for the case of "fat tree" network topology was evaluated under the assumption that all pairs of processes communicate evenly. We propose a scheduling algorithm that tries to minimize this metric, which was implemented atop the OpenStack scheduler. All instances are divided into groups according to compute intensive application they belong to. Every time the scheduler receives a request for launching N new instances of the same group, it maps them to the nodes in such a way that entire group (including already running instances) uses as few lower-level switches in “fat tree” as possible.

We measured the impact of topology-aware scheduling on the performance of NASA Advanced Supercomputing Parallel Benchmarks. Results of 10 of 11 benchmarks changed insignificantly. The average time of Block Tridiagonal test decreased by 14%, the maximum time decreased by 40% and the difference between the maximum time and the minimum time decreased from 42% to 3%, that is, fluctuation almost disappeared.

Keywords: scheduling, virtualization, cloud computing, hop-byte metric

References

[1]. U.S. Department of Energy. The Magellan Report on Cloud Computing for Science. Chicago, 2011.
[2]. Kudryavtsev A.O., Koshelev V.K. and Avetisyan A.I. Perspektivy virtualizatsii vysokoproizvoditel'nykh sistem arkhitektury x64 [The prospects for virtualization of high performance x64 systems] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 189-210, (in Russian)
[3]. Kandalla K., Subramoni H., Vishnu A. and Panda D. K. Designing Topology-Aware Collective Communication Algorithms for Large Scale InfiniBand Clusters: Case Studies with Scatter and Gather. The 10th Workshop on Communication Architecture for Clusters (CAC 10), Int’l Parallel and Distributed Processing Symposium (IPDPS 2010), Ohio, 2010.
[4]. Subramoni H., Kandalla K., Vienne J., Sur S., Barth B., Tomko K., McLay R., Schulz K. and Panda D. K., Design and Evaluation of Network Topology-/Speed-Aware Broadcast Algorithms for InfiniBand Clusters, 2012.
[5]. Bhatele A., Automatic Topology Aware Mapping For Supercomputers, Graduate College of the University of Illinois at Urbana-Champaign, 2010.

[6]. Sudheer C. D. and Srinivasan A. Optimization of the Hop-Byte Metric for Effective Topology Aware Mapping. 19th International Conference on High Performance Computing, 2012
[7]. Gupta A., Milojicic D. and Kalé L. V., Optimizing VM Placement for HPC in the Cloud, Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit, San Jose, CA, USA., 2012.
[8]. Filter Scheduler. Nova Developer guide. Accessed: 16 Apr. 2013. http://docs.openstack.org/developer/nova/devref/filter_scheduler.html
[9]. NAS Parallel Benchmarks. Wikipedia. Accessed: 27 May 2013. http://ru.wikipedia.org/wiki/NAS_Parallel_Benchmarks

Исследование и разработка шаблонов неэффективного поведения в параллельных MPI, UPC приложениях

М.С. Акопян, Н.Е. Андреев [1]
manuk@ispras.ru, andreev.nikita@gmail.com

Аннотация. В данной статье рассматриваются шаблоны в параллельных программах, приводящих к потере производительности. Рассматриваются шаблоны как в параллельных MPI приложениях для вычислительных систем с распределенной памятью, так и в параллельных UPC программах для систем с разделенным глобальным адресным пространством (PGAS). В работе предложен метод автоматизированного обнаружения шаблонов неэффективного поведения в параллельных MPI приложениях и UPC программах. Это позволяет сократить время доводки параллельных приложений.

Ключевые слова: параллельное программирование, семантические ошибки, шаблоны неэффективности, MPI, UPC

1. Введение

Большинство разработанных на сегодняшний день инструментов анализа производительности для различных библиотек и языков параллельного программирования используют низкоуровневые подходы к анализу производительности параллельных программ. Чаще всего это пролифилировочные утилиты, либо визуализаторы трасс. На выходе, в результате анализа, программист получает таблицы и графики со статистикой о выполнении программы. Подобная информация не дает четкого представления о возможных проблемах и узких местах работы приложения. Разработчик вручную просматривает графики в поисках причин замедления

---

1 Работа проводится при финансовой поддержке Министерства образования и науки Российской Федерации, ГК от «07» марта 2013 г. № 14.514.11.4061

программы и потенциальных возможностей оптимизации. В условиях быстрого роста количества ядер в современных высокопроизводительных вычислительных системах, количество информации, которую необходимо обрабатывать программисту, становится неприемлемо большим, а ручные методы анализа - неприменимыми. Поэтому для оптимизации параллельных приложений в современных условиях, необходимы новые методы анализа производительности, выполняющие полную или частичную автоматизацию обработки получаемой информации.

В работе предложен метод автоматизированного обнаружения шаблонов неэффективного поведения в параллельных MPI [1] приложениях и UPC [2] программах. Метод базируется на анализе данных полученных во время исполнения параллельной программы в режиме сбора информации (post-mortem анализе). В работе приводится описание шаблонов для программ с использованием MPI и UPC.

В разделе 2 приводится описание аналогичных работ. В разделе 3 рассматривается метод автоматизированного обнаружения шаблонов в MPI, UPC программах. Раздел 4 содержит описание шаблонов неэффективного поведения в MPI программах, использующих коммуникации точка-точка. В разделе 5 приводится описание шаблонов в UPC программах приводящих к потере производительности.

2. Обзор работ

Одна из наиболее известных систем для улучшения производительности параллельных приложений является TAU [3]. TAU – это набор инструментов для анализа производительности параллельных программ, является результатом усилий исследователей Университета Орегона и Исследовательского центра Juelich из Лос-Аламоса. TAU предоставляет набор статических и динамических инструментов, которые с помощью взаимодействия с пользователем производят комплексный анализ параллельных приложений на языках Fortran, C++, C, Java, и Python. Также в TAU разработан статический инструмент автоматического инструментирования. В системе TAU инструмент Hercule [4] представляет собой прототип модуля, использующего базу знаний для выявления и определения причины «узких мест» производительности в соответствии с парадигмой программирования (таких как master-worker, pipeline и.т.д.) вместо модели программирования (MPI, OpenMP). Инструмент Hercule позволяет анализировать приложения, написанные в любой модели программирования. Однако данный инструмент не может обрабатывать приложения, разработанные с применением комбинации разных парадигм.

Система PPW [5] разработана в лаборатории HCS (High-performance Computing and Simulation) в университете Флориды. Система была создана для анализа производительности параллельных PGAS программ (в частности UPC и SHMEM программ). Вначале программа инструментируется и запускается.

В результате работы инструментированной программы собирается профиль программы (статистические данные времени выполнения) и трасса программы (трасса создается в собственном формате). Собранная трасса может быть использована для анализа и выявления узких мест параллельной программы. Также существуют конверторы трассы параллельной программы в известные форматы, что позволяет использовать известные инструменты визуализации (Vampir, JumpShot, и.т.д.) для ручной оптимизации пользователем. PPW является активно разрабатываемым пакетом, имеет графический интерфейс и богатый функционал. Однако методы, лежащие в основе пакета, являются низкоуровневыми и не используют автоматизированных подходов к анализу.

Система Scalasca [6] представляет собой набор инструментов предназначенных для анализа производительности и был спроектирован специально для использования на больших системах с десятками тысяч ядер, но она также хорошо зарекомендовала себя для маленьких и средних HPC платформ. Scalasca поддерживает измерение и анализ конструкций MPI, OpenMP и гибридные программные конструкции, широко используемые в HPC приложения написанных на языках C, C++, Fortran. Система бала разработана в суперкомпьютерном центре Julich и в немецкой исследовательской школе наук моделирования (Jülich Supercomputing Centre and the German Research School for Simulation Sciences). Scalasca поддерживает инкрементальный анализ производительности, который совмещает информацию времени выполнения с глубоким анализом характера параллелизма с помощью трассы событий. Вначале параллельное приложение инструментируется. При запуске каждый процесс создает файл трассы, содержащий записи для локальных событий данного процесса. После того как выполнение параллельной программы завершилось, Scalasca позволяет проводить post-mortem анализ трассы событий. Вначале локальные трассы разных процессов сливаются в единую трассу. При этом для синхронизации часов различных процессов используется метод, описанный в [7]. После слияния локальных трасс в глобальную, можно использовать инструмент EXPERT [8, 9] для выявления шаблонов неэффективности. Инструмент EXPERT последовательно сканирует события в глобальной трассе и пытается отыскать предопределенные шаблоны, входящие в дистрибутив системы. В трассе могут встречаться только терминальные события (SEND, RECV и.е.д.). Каждое событие помимо других свойств содержит также временную метку. Шаблоном называется комбинация терминальных событий удовлетворяющих определенным предикатам (обычно предикат включает в себя условия относительно временных меток событий с разных процессов). В системе определены порядка тридцати шаблонов для MPI, OpenMP и SHMEM программ.

3. Описание метода автоматизированного обнаружения шаблонов

Метод автоматизированного обнаружения шаблонов неэффективного поведения в параллельных MPI-программах и UPC программах базируется на анализе данных полученных во время исполнения параллельной программы в режиме сбора информации (post-mortem анализе). Для автоматизированного обнаружения шаблонов необходимо вначале получить информацию времени выполнения о критических конструкциях-функциях, которые потенциально могут привести к шаблонам определенных типов. После чего проводится анализ собранной информации на предмет наличия того или иного шаблона. Разрабатываемый подход базируется на использовании свободно распространяемых библиотек из системы Scalasca [6].

Таким образом, алгоритм автоматизированного обнаружения семантических ошибок в параллельных MPI-программах состоит из следующих этапов:

Этап 1. Сбор данных времени выполнения параллельной программы.

Этап 2. Анализ данных полученных на Этапе 1 и выявление шаблонов в параллельной программе.

Этап 3. Создание отчета о выявленных ошибках с привязкой к исходному коду параллельной программы.

Построение трассы параллельного приложения состоит из этапа инструментирования и выполнения инструментированной программы на целевой платформе. Инструментирование программы представляет собой добавление в определенных позициях оригинальной программы вызовы к инструментальной библиотеке. Во время выполнения программы эти вызовы регистрируют наступление определенного события и производят запись в трассу. После этого инструментированная программа переносится на целевую платформу и производится запуск параллельной программы. В результате для каждого процесса программы создается его трасса.

На втором этапе, после получения трассы событий применяется post-mortem анализ - трасса параллельной программы анализируется с целю выявления предопределенных семантических ошибок. Каждому шаблону соответствует определенный критерий (предикат от временной метки события, временной метки соответствующего парного события и.т.д.). Для выявления шаблонов перебираются события из трассы и, при выполнении определенного предиката, регистрируется соответствующий шаблон.

На третьем этапе собранные данные передаются генератору отчетов, который создает итоговый отчет в удобном формате. Итоговый отчет о выявленных ошибках содержит список описателей ошибок. Каждый элемент в списке представляет собой кортеж {type, node, process/thread, file, line, comment,…}, где type – тип ошибки, comment – диагностическое текстовое сообщение об ошибке (при возможности рекомендации). node, process/thread, file, line

определяют адрес узла, идентификатор процесса/потока, имя файла, номер строки в файле, где проявляется данная ошибка.

4. Выявляемые шаблоны в MPI приложениях

Рассмотрим шаблоны с применением коммуникаций точка-точка в параллельных MPI-программах. Пусть F представляет собой коммуникационную функцию MPI. Обозначим через Time_start(F) начальный момент времени непосредственно перед началом функции F. Time_end(F) представляет собой временную метку события после функции F. Обозначим через $I_T(pid, p_i, c_j)$ время простоя процесса с идентификатором pid в результате коммуникации c_j={sendId,recvId} вызванное обнаруженной семантической ошибкой p_i. sendId и recvId представляют собой внутренние идентификаторы посылки и приема соответственно. Пусть ε пороговое значение (на данный момент получено экспериментальным путем).

Замечание. Если $I_T (pid, p_i, c_j) < \varepsilon$, будем считать, что при коммуникации c_j шаблон p_i не проявился.

В текущих реализациях библиотеки MPI (например, MPICH2 [10], MVAPICH2 [11], OpenMPI [12]…) при использовании блокирующих функций пересылок точка-точка используются следующие внутренние протоколы:

а) При размере пересылаемого сообщения меньше чем предопределенная константа используется протокол опережающей посылки. В этом случае пересылка не блокируются. Верхняя граница размера для небольших сообщений отличается в разных реализациях MPI, также как и при разных настройках библиотеки MPI. В реализации MVAPICH2-1.7 при текущих настройках верхняя граница составляет 64KB.

б) При размере пересылаемого сообщения больше границы описанной в пункте а) используется протокол рандеву.

4.1 Шаблоны, связанные с простоем при использовании блокирующих точка-точка коммуникаций

При пересылке сообщений от одного процесса к другому возникают ситуации простоя (idle) либо на одной, либо на другой стороне. Данный эффект на корректность вычислений не влияет, однако он отрицательно отражается на скорости выполнения программы. Устранение этих простоев (при возможности) приведет к увеличению производительности параллельной программы.

Ранняя стандартная посылка. Рассмотрим случай, когда посылка начинается раньше, чем прием (рис. 1). И в этом случае процесс-отправитель теряет время. Критерий шаблона:

$$Time_start(MPI_Send) < Time_start(MPI_Recv)$$

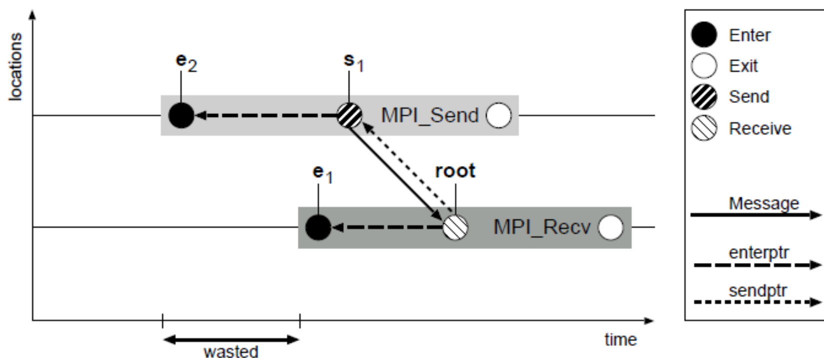


Рис. 1. Поздний прием с передачей MPI_Send.

Замечание. Здесь следует учесть особенности связанные с реализацией MPI (также об этом упоминается в стандарте MPI). Если реализация библиотеки MPI использует протокол опережающей посылки, то функция MPI_Send будет локальной (не блокирующей). Следовательно, простоя в этом случае наблюдаться не будет и необходимо исключить этот случай во избежание ложных срабатываний.

Поздняя стандартная посылка. Рассмотрим случай, когда получение сообщения начинается раньше, чем посылка (рис. 2). И В этом случае процесс-получатель теряет время. Критерий шаблона:

$$\text{Time_start(MPI_Recv)} < \text{Time_start(MPI_Send)}$$

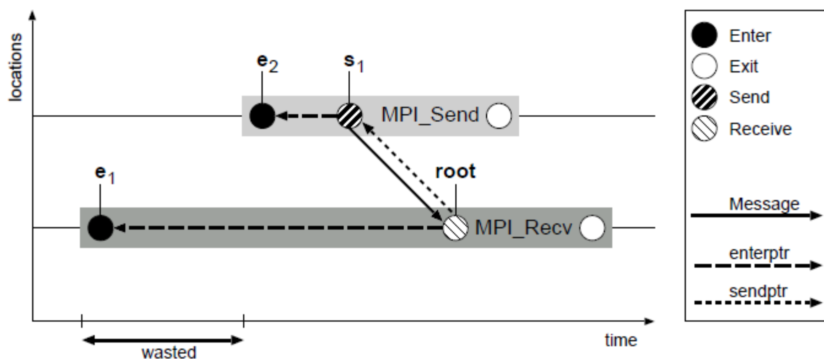


Рис. 2. Ранний прием с передачей MPI_Send.

Поздняя буферизированная посылка. Функция MPI_Bsend является локальной функцией (отправитель копирует сообщение в буфер и возвращает управление, а система времени выполнения MPI занимается отправкой сообщения из буфера). Пусть имеется ситуация как на рис. 2, но вместо

функции MPI_Send используется MPI_Bsend. Прием сообщения начинается раньше, чем соответствующая посылка MPI_Bsend. В этом случае принимающая сторона простаивает в ожидании. Критерий шаблона:

Time_start(MPI_Recv) < Time_start(MPI_Bsend)

$$\text{I_T(pid, pi, cj)} = \begin{cases} 0, & \text{если (Time_start(MPI_Bsend)-Time_start(MPI_Recv))} < \varepsilon \\ \text{Time_start(MPI_Bsend)-Time_start(MPI_Recv)}, & \text{иначе} \end{cases}$$

Ранняя буферизированная посылка. Посылка сообщения начинается раньше, чем соответствующий прием. Однако в данном случае процесс-отправитель не простаивает, потому что функция MPI_Bsend является локальной – функция копирует сообщение в буфер и возвращает управление, а система времени выполнения MPI занимается отправкой сообщения из буфера.

Поздняя синхронная посылка. Пусть имеется ситуация как на рис. 1, но вместо функции MPI_Send используется MPI_Ssend. Прием сообщения начинается раньше, чем соответствующая посылка MPI_Ssend. В этом случае принимающая сторона (процесс 1) простаивает в ожидании. Критерий шаблона:

Time_start(MPI_Recv) < Time_start(MPI_Ssend)

$$\text{I_T(pid, pi, cj)} = \begin{cases} 0, & \text{если (Time_start(MPI_Ssend)-Time_start(MPI_Recv))} < \varepsilon \\ \text{Time_start(MPI_Ssend)-Time_start(MPI_Recv)}, & \text{иначе} \end{cases}$$

Ранняя синхронная посылка. Пусть имеется ситуация как на рис. 2, но вместо функции MPI_Send используется MPI_Bsend. Посылка сообщения начинается раньше, чем соответствующий прием. Синхронная посылка использует протокол рандеву – отправляющий ждет, пока сообщение будет принята на стороне принимающего и после чего возвращает управление. То есть из-за того что MPI_Recv начинается позже, отправитель простаивает. Критерий шаблона:

Time_start(MPI_Ssend) < Time_start(MPI_Recv)

$$\text{I_T(pid, pi, cj)} = \begin{cases} 0, & \text{если (Time_start(MPI_Recv) - Time_start(MPI_Ssend))} < \varepsilon \\ \text{Time_start(MPI_Recv) - Time_start(MPI_Ssend)}, & \text{иначе} \end{cases}$$

Поздняя посылка по готовности. Пусть имеется ситуация как на рис. 2, но вместо функции MPI_Send используется MPI_Rsend. Прием сообщения начинается раньше, чем соответствующая посылка MPI_Rsend. В этом случае принимающая сторона (процесс 1) простаивает в ожидании. Критерий шаблона:

Time_start(MPI_Recv) < Time_start(MPI_Rsend)

$$I_T(pid, pi, cj) = \begin{cases} 0, & \text{если } (Time_start(MPI_Rsend)-Time_start(MPI_Recv)) < \varepsilon \\ Time_start(MPI_Rsend)-Time_start(MPI_Recv), & \text{иначе} \end{cases}$$

Ранняя посылка по готовности. Пусть имеется ситуация как на рис. 1, но вместо функции MPI_Send используется MPI_Rsend. Посылка сообщения начинается раньше, чем соответствующий прием. Согласно стандарту MPI [1] данная ситуация является ошибочной. Однако в текущей реализации MPICH2 [10], MVAPICH2 [11] MPI_Rsend отображается на MPI_Send. В результате ошибки не выдается, а сообщение передается получателю. Таким образом, шаблоны ранней посылки по готовности можно разбить на два случая:

а) Размер сообщения меньше определенной константы. Тогда применяется опережающая посылка, и операция MPI_Rsend является локальной. Следовательно, в этом случае простоя не будет и такая ошибка не приводит к снижению эффективности программы.
б) Размер пересылаемого сообщения больше определенной константы. В этом случае будет применяться протокол рандеву, что при поздней инициализации операции приема (MPI_Recv) приведет к простою процесса отправителя.

На рис. 1 приводится графическое представление выше описанного шаблона. Критерий шаблона:

$$Time_start(MPI_Rsend) < Time_start(MPI_Recv)$$

$$I_T(pid, pi, cj) = \begin{cases} 0, & \text{если } (Time_start(MPI_Recv) - Time_start(MPI_Rsend)) < \varepsilon \\ Time_start(MPI_Recv) - Time_start(MPI_Rsend), & \text{иначе} \end{cases}$$

4.2 Шаблоны, связанные с «неправильным порядком сообщений»

Эффект с «неправильным порядком сообщений» может возникнуть когда в процессе-получателе сообщения ожидаются в одном порядке, а процесс-отправитель посылает сообщения в обратном порядке (рис. 3*Рис.*). Переупорядочив сообщения можно не только добиться ускорения программы, но также потребуется меньший размер буфера для хранения необработанных сообщений.

Если посылка-прием используют протокол рандеву, то очевидно программа попадает в дедлок. Но если посылка локальная (буферизированная посылка, либо обычный send, но размер сообщения маленький и сообщение в реализации MPI отправляется через внутренний MPI буфер), то блокировки не будет, а будет неэффективная организация коммуникаций.

Неправильный порядок с применением MPI_Send. При применении пары функций {MPI_Send, MPI_Recv} учитывая особенности реализации MPI_Send возможны два варианта:

а) При маленьких сообщениях вызовы MPI_Send не блокируются и можно получить семантическую ошибку «неправильный порядок».
б) При больших сообщениях для завершения функции MPI_Send необходимо, чтобы соответствующая функция MPI_Recv была начата. Но поскольку сообщения принимаются в обратном порядке, то данная программа останется в дедлоке.

Рис. 3*Рис.* представляет графическое представление шаблона неэффективного поведения при пересылке сообщений в неправильном порядке с применением функции MPI_Send.

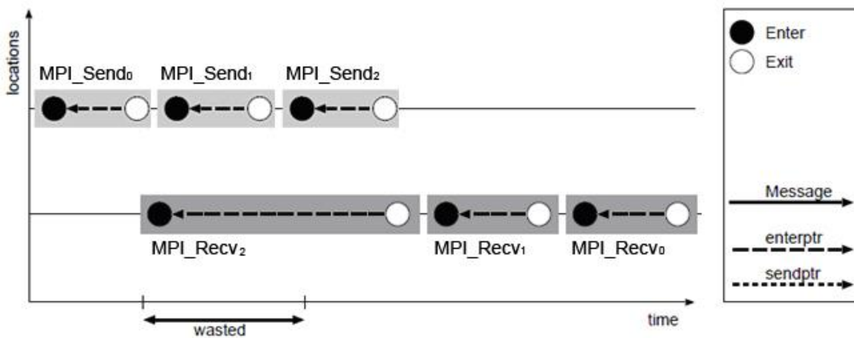


Рис. 3- Шаблон «неправильный порядок» при применении пары функций {MPI_Send, MPI_Recv}.

Ниже представлен критерий шаблона:

$Time_end(MPI_Send_0) < Time_end(MPI_Send_1) < Time_end(MPI_Send_2)$ и
$Time_end(MPI_Recv_2) < Time_end(MPI_Recv_1) < Time_end(MPI_Recv_0)$ и
$Time_start(MPI_Recv_2) < Time_start(MPI_Send_2)$

Неправильный порядок с применением MPI_Ssend. Шаблон неправильный порядок сообщений не возможен при применении пары функций {MPI_Ssend, MPI_Recv}, поскольку в этом случае последовательные вызовы MPI_Ssend не встретив соответствующих MPI_Recv-ов заблокируются.

Неправильный порядок с применением MPI_Bsend. Пусть имеется ситуация как на рис. 3, но вместо функции MPI_Send используется MPI_Bsend. Шаблон «неправильный порядок» может возникнуть при использовании пары функций {MPI_Bsend, MPI_Recv }. Ниже представлен критерий шаблона:

$Time_end(MPI_Bsend_0) < Time_end(MPI_Bsend_1) < Time_end(MPI_Bsend_2)$ и
$Time_end(MPI_Recv_2) < Time_end(MPI_Recv_1) < Time_end(MPI_Recv_0)$ и
$Time_start(MPI_Recv_2) < Time_start(MPI_Bsend_2)$

Неправильный порядок с применением MPI_Rsend. Пусть имеется ситуация как на рис. 3*Рис.* , но вместо функции MPI_Send используется MPI_Rsend. Как показали дальнейшие исследования в реализации MVAPICH-а при работе с {MPI_Rsend,MPI_Recv}-ом важно не абсолютное запаздывание соответствующего MPI_Recv-а, а относительный порядок относительно остальных MPI_Recv-ов в очереди сообщений на стороне процесса-получателя. На принимающей стороне (p_1) существует очередь сообщений, где хранятся все сообщения, которые прибыли (либо прибыло служебное сообщение-запрос на начало пересылки), но еще не были обработаны (не было вызова соответствующего MPI_Recv-а). После того как в p_1 был вызван MPI_Recv_2 система времени выполнения берет первый элемент в очереди и поскольку MPI_Rsend_0 не соответствует MPI_Recv_2, то система регистрирует эту ошибку, но выдача ошибки откладывается. Далее подбирается следующий элемент из очереди (MPI_Rsend_1) и поскольку соответствие найдено производится прием сообщения в буфер MPI_Rsend_1. Пользовательская программа продолжает выполнять инструкции. Если в программе не будет вызова MPI_Recv_0, то программа завершится, и ошибки не будет. Если же где то дальше находится MPI_Recv_0, то программа аварийно завершается с выдачей соответствующего сообщения. Следовательно, шаблон «неправильный порядок» не применим к случаю, когда используется последовательность передачи сообщений посредством MPI_Rsend-ов и обратная последовательность приема сообщений MPI_Recv-ами.

4.3 Шаблоны, связанные с не-блокирующими точка-точка коммуникациями

Пусть имеется пара вызовов {MPI_Isend,MPI_Wait} в процессе p_0 и {MPI_Irecv, MPI_Wait} в процессе p_1. Рассмотрим семантические ошибки, которые возникают в этом случае процессах pid_0 и pid_1.

Ожидание на стороне отправителя при использовании {MPI_Isend, MPI_Irecv}. Рассмотрим процесс pid_0. В этом случае после вызова функции MPI_Isend управление возвращается в процесс pid_0 и выполняются вычислительные инструкции, после чего вызывается функция MPI_Wait. Если вызов MPI_Wait был произведен слишком рано, то процесс блокируется и простаивает. В трассе событий содержится также временные метки для каждого события, следовательно, разница временных меток событий после и перед вызовом MPI_Wait позволит вычислить время простоя процесса. Критерий шаблона:

$$I_T(pid_0, pi, cj) = \begin{cases} 0, & \text{если } (\text{Time_end(MPI_Wait) - Time_start(MPI_Wait)}) < \varepsilon \\ \text{Time_end(MPI_Wait) - Time_start(MPI_Wait)}, & \text{иначе} \end{cases}$$

$I_T(pid_0,p_i,c_j) > 0$

Помимо определения прогноза можно выдать диагностическое сообщение с оценкой оптимальной дистанции(O_D(pid,p_i,c_j)) для вызова MPI_Wait.

$$O_D(pid_0, pi, cj) = \begin{cases} \Delta Ti + T_{send}, & \text{если Time_start(MPI_Isend) > Time_start(MPI_Irecv)} \\ (\text{Time_start(MPI_Irecv)-Time_start(MPI_Isend)}) + \Delta Ti + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Isend, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне получателя при использовании {MPI_Isend, MPI_Irecv}. При приеме сообщения в процессе pid_1 возникает аналогичная ситуация. После вызова функции MPI_Irecv управление возвращается в процесс pid_1 и выполняются вычислительные инструкции, после чего вызывается функция MPI_Wait. Если вызов MPI_Wait был произведен слишком рано, то процесс блокируется и простаивает. В трассе событий содержится также временные метки для каждого события, следовательно, разница временных меток событий после и перед вызовом MPI_Wait позволит вычислить время простоя процесса. Критерий шаблона:

$$I_T(pid_1, pi, cj) = \begin{cases} 0, & \text{если } (\text{Time_end(MPI_Wait) - Time_start(MPI_Wait)}) < \varepsilon \\ \text{Time_end(MPI_Wait) - Time_start(MPI_Wait)}, & \text{иначе} \end{cases}$$

I_T(pid1,pi,cj) > 0

Оценка оптимальной дистанции(O_D(pid,p_i,c_j)) для вызова MPI_Wait:

$$O_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, & \text{если Time_start(MPI_Isend) < Time_start(MPI_Irecv)} \\ (\text{Time_start(MPI_Isend)-Time_start(MPI_Irecv)}) + \Delta T_i + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Irecv, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне отправителя при использовании {MPI_Ibsend, MPI_Irecv}. В этом случае ошибки не будет, потому что не-блокирующая функция MPI_Ibsend является локальной – функция копирует сообщение в буфер и возвращает управление, а система времени выполнения MPI занимается отправкой сообщения из буфера.

Ожидание на стороне получателя при использовании {MPI_Ibsend, MPI_Irecv}. На стороне отправителя используется пара функций {MPI_Ibsend, MPI_Wait}, на стороне получателя {MPI_Irecv, MPI_Wait}. В этом случае шаблон может проявиться только на стороне получателя. По аналогии с

шаблоном при использовании пары **{MPI_Isend, MPI_Irecv}** критерий шаблона представляется формулами:

$$I_T(pid_1, pi, cj) = \begin{cases} 0, & \text{если } (Time_end(MPI_Wait) - Time_start(MPI_Wait)) < \varepsilon \\ Time_end(MPI_Wait) - Time_start(MPI_Wait), & \text{иначе} \end{cases}$$

$$I_T(pid_1,pi,cj) > 0$$

Оценка оптимальной дистанции($O_D(pid,p_i,c_j)$) для вызова MPI_Wait:

$$O_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, & \text{если } Time_start(MPI_Ibsend) < Time_start(MPI_Irecv) \\ (Time_start(MPI_Ibsend) - Time_start(MPI_Irecv)) + \Delta T_i + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Irecv, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне отправителя при использовании {MPI_Issend, MPI_Irecv}. На стороне отправителя используется пара функций {MPI_Issend,MPI_Wait}, на стороне получателя {MPI_Irecv, MPI_Wait}. В этом случае после вызова не-блокирующей синхронной функции MPI_Issend управление возвращается в процесс pid_0. Если вызов соответствующей функции MPI_Wait был произведен слишком рано, то процесс блокируется и простаивает. По аналогии с шаблоном при использовании пары **{MPI_Issend, MPI_Irecv}** критерий шаблона представляется формулами:

$$I_T(pid_0, pi, cj) = \begin{cases} 0, & \text{если } (Time_end(MPI_Wait) - Time_start(MPI_Wait)) < \varepsilon \\ Time_end(MPI_Wait) - Time_start(MPI_Wait), & \text{иначе} \end{cases}$$

$$I_T(pid_0,p_i,c_j) > 0$$

Оценка оптимальной дистанции($O_D(pid,p_i,c_j)$) для вызова MPI_Wait:

$$O_D(pid_0, pi, cj) = \begin{cases} \Delta Ti + T_{send}, & \text{если } Time_start(MPI_Issend) > Time_start(MPI_Irecv) \\ (Time_start(MPI_Irecv) - Time_start(MPI_Issend)) + \Delta Ti + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Issend, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне получателя при использовании {MPI_Issend, MPI_Irecv}. При приеме сообщения в процессе pid_1 возникает аналогичная ситуация. После вызова функции MPI_Irecv управление возвращается в процесс pid_1 и выполняются вычислительные инструкции, после чего вызывается функция MPI_Wait. Если вызов MPI_Wait был произведен слишком рано, то процесс блокируется и простаивает. По аналогии с шаблоном при использовании пары **{MPI_Issend, MPI_Irecv}** критерий шаблона вычисляется как:

$$I_T(pid_1, pi, cj) = \begin{cases} 0, & \text{если } (Time_end(MPI_Wait) - Time_start(MPI_Wait)) < \varepsilon \\ Time_end(MPI_Wait) - Time_start(MPI_Wait), & \text{иначе} \end{cases}$$

$I_T(pid1,pi,cj) > 0$

Оценка оптимальной дистанции($O_D(pid,p_i,c_j)$) для вызова MPI_Wait:

$$O_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, & \text{если } Time_start(MPI_Issend) < Time_start(MPI_Irecv) \\ (Time_start(MPI_Issend) - Time_start(MPI_Irecv)) + \Delta T_i + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Irecv, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне отправителя при использовании {MPI_Irsend, MPI_Irecv}. На стороне отправителя используется пара функций {MPI_Irsend, MPI_Wait}, на стороне получателя {MPI_Irecv, MPI_Wait}. Данный случай является пересечением шаблонов с использованием блокирующих функция {MPI_Rsend,MPI_Recv} и шаблонов ожидание на стороне получателя при использовании {MPI_Isend, MPI_Irecv}. Следовательно, шаблон может проявиться только тогда, когда размер пересылаемого сообщения больше определенной константы. В этом случае будет применяться протокол рандеву, что при поздней инициализации операции приема (MPI_IRecv) приведет к простою процесса отправителя. Таким образом, критерий шаблона представляется формулами:

$$I_T(pid_0, pi, cj) = \begin{cases} 0, & \text{если } (Time_end(MPI_Wait) - Time_start(MPI_Wait)) < \varepsilon \\ Time_end(MPI_Wait) - Time_start(MPI_Wait), & \text{иначе} \end{cases}$$

$I_T(pid_0,p_i,c_j) > 0$

Оптимальная дистанция ($O_D(pid,p_i,c_j)$) для вызова MPI_Wait оценивается формулой:

$$O_D(pid_0, pi, cj) = \begin{cases} \Delta Ti + T_{send}, & \text{если } Time_start(MPI_Irsend) > Time_start(MPI_Irecv) \\ (Time_start(MPI_Irecv) - Time_start(MPI_Irsend)) + \Delta Ti + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Irsend, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне получателя при использовании {MPI_Irsend, MPI_Irecv}. На стороне отправителя используется пара функций {MPI_Irsend, MPI_Wait}, на стороне получателя {MPI_Irecv, MPI_Wait}. В этом случае шаблон может возникнуть только на стороне получателя. По аналогии с шаблоном при использовании пары **{MPI_Isend, MPI_Irecv}** критерий шаблона представляется формулами:

$$I_T(pid_1, pi, cj) = \begin{cases} 0, & \text{если } (Time_end(MPI_Wait) - Time_start(MPI_Wait)) < \varepsilon \\ Time_end(MPI_Wait) - Time_start(MPI_Wait), & \text{иначе} \end{cases}$$

$I_T(pid_1,pi,cj) > 0$

Оптимальная дистанция ($O_D(pid,p_i,c_j)$) для вызова MPI_Wait оценивается формулой:

$$O_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, & \text{если Time_start(MPI_Irsend)} < \text{Time_start(MPI_Irecv)} \\ (\text{Time_start(MPI_Irsend)-Time_start(MPI_Irecv)}) + \Delta T_i + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Irecv, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

4.4 Близкая пересылка, передача сообщений

Рассмотрим программу, где в процессе pid_i применяется посылка и прием сообщения с процессом pid_j и операция посылки и приема находятся близко (рис. 4).

```
        if( rank == pidi )
        {
        int *send_buf = (int *)malloc(sizeof(int) * 12001);
        int *recv_buf = (int *)malloc(sizeof(int) * 12001);

        MPI_Send(send_buf,12001,MPI_INT, pidj,0,MPI_COMM_WORLD);
…
        MPI_Recv(recv_buf,12001,MPI_INT, pidj,0,MPI_COMM_WORLD);
        }
        else if( rank == pidj )
        {
        int * send_buf = (int *)malloc(sizeof(int) * 12001);
        int * recv_buf = (int *)malloc(sizeof(int) * 12001);
               MPI_Status stat;
               MPI_Status stat2;

        MPI_Recv(recv_buf,12001,MPI_INT, pidi,0,MPI_COMM_WORLD);
…
        MPI_Send(send_buf,12001,MPI_INT, pidi,0,MPI_COMM_WORLD);
        }
```

Рис. 4- Пример использования тесной посылки и приема сообщений процессами.

Если был найден такой шаблон и логика программы позволяет, то можно объединить функции MPI_Send, MPI_Recv в функцию MPI_Sendrecv, что позволит получить достаточно серьезный выигрыш по времени выполнения. Происходит это из-за того, что в реализациях MPI достаточно хорошо реализована функция MPI_Sendrecv. А также на нижнем уровне современные высокопроизводительные коммуникационные сети (Infiniband [13]) поддерживают дуплексную связь, что позволяет одновременно посылать и получать сообщение одному HCA. И в этом случае вместо того, чтобы pid_i ждал окончания операции MPI_Send и только после этого начал ждать

окончания MPI_Recv, посылка и получение производятся одновременно. Пусть Δ_{SR} некоторая предопределенная константа. Критерий шаблона:

$$(Time_start(MPI_Recv) - Time_end(MPI_Send)) < \Delta_{SR}$$

При нахождении этого шаблона выдается диагностическое сообщение. И пользователь, убедившись, что не существует зависимости по данным для буферов send_buf и recv_buf, может заменить пару вызовов функций {MPI_Send, MPI_Recv} на вызов функции MPI_Sendrecv.

5. Выявляемые шаблоны в UPC приложениях

Набор из двадцати шаблонов неэффективного поведения был разбит на четыре группы. В первую группу входят шаблоны для обнаружения задержек в коллективных операциях передачи данных, так же называемых операциями релокализации в языке UPC [2]. В первую группу входят восемь шаблонов. Во вторую группу вошли семь шаблонов, связанных с операциями явной и неявной синхронизации, которые присутствуют практически во всех языках параллельного программирования. Третья группа состоит из трех шаблонов, выполняющих анализ односторонних и коллективных операций передачи данных.

Последняя четвертая группа, состоящая из двух шаблонов, связана с моделью параллельного программирования «Главный-Подчиненные», в которой «главный» (master) поток создает набор «подчиненных» (slave) потоков и работа распределяется между ними.

5.1 Блокировки в коллективных операциях передачи данных

В отличие от односторонних коммуникационных операций *upc_put()/upc_get()*, в коллективных операциях присутствуют сложные зависимости по данным, требующие синхронизации. Если программа написана неудачно и четкая координация между потоками нарушена, то неизбежно возникнут дополнительные задержки. Чтобы свести эту проблему до минимума, в UPC для всех операций релокализации были выделены три типа синхронизации, которые указываются последним аргументом в вызове функций отдельно для синхронизации на входе в операцию и на выходе из нее. С учетом этих особенностей, были сформулированы восемь шаблонов для анализа следующих операций релокализации данных: *upc_all_broadcast(), upc_all_scatter(), upc_all_gather(), upc_all_gather_all(), upc_all_exchange(), upc_all_permute(), upc_all_reduce() и upc_all_prefix_reduce()*.

Шаблон «Синхронизация на входе в коллективную операцию». Данный шаблон справедлив для всех операций релокализации, в которых используется тип синхронизации UPC_IN_ALLSYNC. Предположим для данного примера, что это операция широковещательной рассылки upc_all_broadcast(). В данном

случае каждый поток обязан дождаться на входе всех остальных. Когда потоки входят в операцию релокализации в разные моменты времени - это вносит нежелательные накладные расходы на синхронизацию. Данный шаблон не встречается в программной модели передачи сообщений и характерен только для языка UPC. Это объясняется тем, что стандарт MPI не накладывает строгих ограничений на порядок входа потоков в коллективную операцию. Будут ли потоки ожидать друг друга, как при использовании барьерной синхронизации, или же будет использован более оптимальный алгоритм, определяется реализацией, и этого нельзя отследить. В случае с UPC, синхронизация указывается программистом явно, что позволяет сформулировать новые более точные шаблоны.

Шаблон «Синхронизация на выходе из коллективной операции». Данный шаблон справедлив для всех операций релокализации, в которых используется тип синхронизации UPC_OUT_ALLSYNC. Шаблон также характерен только для языка UPC. Как и в шаблоне с синхронизацией на входе, все потоки обязаны дождаться друг друга, только теперь на выходе из операции.

Шаблон «Поздняя рассылка». Данный шаблон возникает при использовании синхронизации UPC_IN_MYSYNC на входе в операции один ко многим. К ним относятся такие операции языка UPC, как: upc_all_broadcast() и upc_all_scatter(). Если поток, рассылающий данные, входит в операцию позднее потоков, принимающих данные, то последние обязаны приостановить свое выполнение. Шаблон отражает время, потерянное в результате возникновения такой ситуации.

Шаблон «Ранняя сборка». Данный шаблон присущ операциям, выполняющим сборку данных, таким как upc_all_gather() и upc_all_reduce(), если для синхронизации на входе используется UPC_IN_MYSYNC. Данный шаблон аналогичен шаблону «Поздняя рассылка» за тем исключением, что причиной его возникновения является поток получатель данных, в том случае, если он входит в операцию позднее других.

Шаблон «Ранняя префиксная редукция». Данный шаблон уникален для операции префиксной редукции upc_all_prefix_reduce(). Возникает в случае использования синхронизации UPC_IN_MYSYNC. В данной операции результат редукции в потоке n зависит от редукции, выполненной в потоке n-1. Если хотя бы один из потоков 0..n-1 не вошел в операцию, поток n обязан ждать.

Шаблон «Синхронизация на входе в коллективную операцию многие ко многим». Данный шаблон в отличие от шаблона «Синхронизация на входе в коллективную операцию» возникает при использовании синхронизации UPC_IN_MYSYNC и только в операциях, которые отправляют данные от многих потоков ко многим, к которым относятся upc_all_gather_all() и upc_all_exchange(). Если в первом случае все потоки ждут друг друга, то в данном шаблоне они имеют право работать с данными потоков, которые уже

вошли в коллективную операцию. Тем не менее, если некоторые потоки еще не вошли в операцию, а с остальными обмен уже произошел, то потоку придется остановиться. Шаблон описывает потерянное в такой ситуации время. Искомые данные и потерянное время находятся аналогично первому случаю. Важно лишь отметить, что данный шаблон не достаточно точен, так как часть времени, попадающего в категорию потерянного, является полезным. Это время, потраченное на обмен с потоками, уже вошедшими в операцию. На практике невозможно определить, какую часть времени поток обменивался данными, а какую часть находился в ожидании. Поэтому данный шаблон можно рассматривать лишь как подозрение на неэффективное поведение. Если программа затратила значительную часть времени в такой ситуации, то нужно более подробно взглянуть на проблему.

Шаблон «Синхронизация на выходе из коллективной операции многие ко многим». Данный шаблон аналогичен предыдущему шаблону, с тем отличием, что синхронизация и соответственно потеря времени происходит на выходе из операции.

Шаблон «Ожидание внутри коллективной операции». Данный шаблон возникает в операциях upc_all_broadcast(), upc_all_scatter(), upc_all_gather() и upc_all_reduce() при использовании типа синхронизации UPC_IN_MYSYNC. Этот шаблон является дополнением к шаблонам «Поздняя рассылка» и «Ранняя сборка». При выполнении коллективной операции в языке UPC может возникнуть ситуация, когда поток находится в коллективной операции один. Это происходит, например, если поток входит в операцию первым, либо выходит последним или если часть потоков уже выполнили вычисления и вышли, а некоторые потоки еще не успели дойти до коллективной операции. Для нахождения потерянного времени в общем случае, достаточно из времени выполнения операции корневым потоком, вычесть время выполнения операции другими потоками, пересекающееся с корневой.

5.2 Операции синхронизации

В языке UPC явная синхронизация представлена операциями *upc_barrier(), upc_notify()/upc_wait() и upc_fence()*. В первом случае программа блокируется до тех пор пока все потоки не достигнут операции *upc_barrier()*, во втором случае, который называется барьером с расщепленной фазой, синхронизация разбивается на два этапа и накладывает менее строгие требования к синхронизации. Операция *upc_fence()* заставляет поток дождаться завершения всех неявных односторонних операций обращения к памяти других потоков. Кроме того, неявная синхронизация потоков происходит при выполнении атомарных операций работы с памятью *bupc_atomic*()*, а так же в операции динамического выделения памяти *upc_all_alloc()*.

Шаблон «Конкуренция за блокировку». Данный шаблон возникает, если один из потоков пытается захватить блокировку, которой в данный момент владеет другой поток. Шаблон является типичным для модели общей памяти.

Однако он хорошо ложится на программную модель PGAS, так как в ней используется модель распределенной общей памяти. Если в момент вызова функции upc_lock() блокировка уже захвачена, то поток останавливает свое выполнение до освобождения блокировки. Время, потраченной в данной ситуации, соответствует времени, проведенном в функции upc_lock().

Шаблон «Ожидание в барьере». Данный шаблон возникает в ситуациях, когда для синхронизации потоков в программе используется барьерная синхронизация. Если в приложении встретился барьер, то все потоки обязаны остановиться и дождаться друг друга. Шаблон фиксирует время, затраченное на выполнение барьерной синхронизации.

Шаблон «Завершение барьера». Это достаточно специфический шаблон, в том смысле, что в нормальной ситуации все потоки должны выходить из барьера в один и тот же момент времени. Любое, даже незначительное время, проведенное в данном шаблоне, может означать неэффективность реализации PGAS языка, либо помехи со стороны других процессов, работающих на том же расчетном узле.

Шаблон «Ожидание в операции динамического выделения памяти». Данный шаблон возникает в операции upc_all_alloc(). В языке UPC присутствуют ряд операций для динамического выделения общей памяти. Если память выделяется коллективно при помощи функции upc_all_alloc(), то возникают требования к синхронизации. В действительности выделение памяти происходит в потоке 0, после чего результат операции рассылается по всем потокам, аналогично операции upc_all_broadcast(). Если поток 0 входит в операцию позже других, то остальные потоки обязаны ждать рассылки результата, поэтому описание данного шаблона аналогично шаблону «Поздняя рассылка».

Шаблон «Ожидание в решетке». Если в момент выполнения операции upc_fence() существуют незавершенные односторонние операции обращения данного потока к памяти других потоков, то его выполнение блокируется до их завершения. Операция upc_fence(), вызванная в одном потоке, не влияет на другие. Так же, на блокировку выполнения потока операцией upc_fence() не влияют обращения других потоков в его память. Потерянное время расчитывается, как время выполнения операции upc_fence().

Шаблон «Ожидание в расщепленном барьере». Расщепленный барьер отличается от обычного барьера upc_barrier() тем, что он выполняется в два этапа. Когда поток завершает выполнение операций, требующих синхронизации, он вызывает upc_notify(). После чего поток может продолжать выполнять часть локального для потока программного кода, не требующего синхронизации. Когда его выполнение так же завершено, поток выполняет операцию upc_wait(), которая блокирует выполнение до тех пор, пока все остальные не выполнят операцию upc_notify(). Расщепленный барьер накладывает менее строгие требования на синхронизацию, но так же вызывает

блокировку потоков. Потерянное в шаблоне время рассчитывается, как время выполнения операции upc_notify().

Шаблон «Ожидание в атомарной операции». Атомарность выполнения даже таких операций, как увеличение значения переменной на величину другой переменной, часто не гарантируется в параллельном программировании. Такая операция как x += y трансформируется в несколько операций: READ(x) => t1, READ(y) => t2, t1 + t2 => t3, t3 => WRITE(x). Ни что не мешает втором потоку изменить значение y, после того, как операция уже начала выполнятся и значение x было уже считано в первом потоке. Спецификация языка UPC версии 1.3 вводит новый набор функций, которые позволяют атомарно выполнять ряд составных операций над переменными, например таких как «сравнение и замена». Для обеспечения атомарности этих операций, используется синхронизация. Данный шаблон описывает время, потерянное на синхронизацию в атомарной функции.

5.3 Операции передачи данных

Шаблоны из данной группы позволяют обнаружить «горячие» точки и узкие места программы, с точки зрения объема передаваемых между потоками данных.

Шаблон «Частая коммуникация». Данный шаблон позволяет идентифицировать блоки кода программы генерирующие наибольшее количество операций коммуникации. Так как коммуникации являются наиболее медленными операциями параллельного приложения, обнаружение «горячих» точек программы позволяет выявить места исходного кода, наиболее выгодные с точки зрения оптимизации.

Шаблон «Большие сообщения». В шаблоне анализируется размер отправленных и полученных сообщений, в результате выполнения коллективных и односторонних коммуникационных операций. Использование более крупных посылок данных в одном из блоков программы, по сравнению с другими, может создать «узкое» место в работе приложения. Шаблон позволяет обнаружить подобные операции передачи данных, которые в дальнейшем могут быть оптимизированы, например, путем использования неблокирующих коммуникационных операций.

Шаблон «Несбалансированные коммуникации». В шаблоне сравнивается время, проведенное разными потоками в одной и той же коммуникационной операции. В лучшем случае это время должно совпадать во всех потоках. Если потоки затрачивают существенно разное время в одной и той же операции, то это означает, что присутствует разбалансировка нагрузки, которая требует дальнейшей оптимизации.

5.4 Модель «Главный-Подчиненные»

Задача шаблонов из данной группы выявление возможной разбалансировки нагрузки между потоками и связанную с этим потерю процессорного времени.

Шаблон «Медленные подчиненные». Часто в параллельных программах используется подход главный-подчиненные, когда один из потоков выполняет генерацию данных, их отправку подчиненным потокам и последующую сборку результатов. Шаблон описывает ситуацию, когда главный поток блокирует свое выполнение в ожидании результатов от подчиненных потоков, вместо выполнения полезной работы.

Шаблон «Перегруженный главный». Данный шаблон описывает противоположную ситуацию, когда подчиненные ожидают данные от главного потока, вместо выполнения вычислений. Главный поток может затрачивать излишне большое время на сбор результатов, либо не успевать генерировать новые данные для расчетов. В обоих случаях имеет место разбалансировка нагрузки между потоками, являющаяся потенциальным местом для оптимизации.

6. Заключение

Инструменты анализа производительности оказывают помощь разработчикам параллельных приложений в оптимизации параллельного кода и ускоряют процесс доводки параллельных приложений. Большинство существующих инструментов основаны на методах ручного анализа. В статье приведен метод автоматизированного обнаружения шаблонов неэффективного поведения в параллельных MPI приложениях и UPC программах. Это становится особенно актуальным в условиях постоянного роста количество ядер в современных кластерах и увеличения сложности параллельных комплексов. В работе приводится описание шаблонов неэффективного поведения для MPI-программ и шаблонов разработаных для языка Unified Parallel C, который является одним из представителей программной модели Partitioned Global Address Space (PGAS).

Список литературы

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

[2]. W. Chen, C. Iancu, K. Yelick. Communication Optimizations for Fine-grained UPC Applications. //14th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2005.

[3]. Sameer S. Shende Allen D. Malony. “The Tau Parallel Performance System”, International Journal of High Performance Computing Applications, Volume 20 , Issue 2, Pages: 287 – 311, May 2006.

[4]. L. Li and A.D. Malony, “Model-Based Performance Diagnosis of Master-Worker Parallel Computations,” Lecture Notes in Computer Science, Number 4128, Pages 35-46, 2006.
[5]. H. Su, M. Billingsley III, and A. George. "Parallel Performance Wizard: A Performance System for the Analysis of Partitioned Global Address Space Applications," International Journal of High-Performance Computing Applications, Vol. 24, No. 4, Nov. 2010, pp. 485-510.
[6]. Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, Bernd Mohr. The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience, 22(6):702–719, April 2010.
[7]. Felix Wolf. Automatic Performance Analysis on Parallel Computers with SMP Nodes. PhD thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003, ISBN 3-00-010003-2.
[8]. Wolf, F., Mohr, B. Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture 49(10-11) (2003) 421–439.
[9]. Wolf, F., Mohr, B., Dongarra, J., Moore, S. Efficient Pattern Search in Large Traces through Successive Refinement. In: Proc. European Conf. on Parallel Computing (Euro-Par, Pisa, Italy), Springer (2004).
[10]. MPICH. http://www.mpich.org
[11]. MVAPICH. http://mvapich.cse.ohio-state.edu.
[12]. OpenMPI. http://www.open-mpi.org.
[13]. Infiniband. http://www.infinibandta.org.

Research and development of inefficiency patterns in MPI, UPC applications

M.S. Akopyan, N.E. Andreev
ISP RAS, Moscow, Russia, Thomas Duryea Consulting, Melbourne, Australia
manuk@ispras.ru, andreev.nikita@gmail.com

Abstract. Most of developed tools for analysis for various libraries (MPI, OpenMP) and languages for parallel programming use low level approaches to analyze the performance of parallel applications. There are a lot of profiling tools and trace visualizers which produce tables, graphs with various statistics of executed program. In most cases developer has to manually look for bottlenecks and opportunities for performance improvement in the produced statistics and graphs. The amount of information developer has to handle manually, increase dramatically with number of cores, number of processes and size of problem in application. Therefore new methods of performance analysis fully or partially handling output information will be more beneficial. To apply the same analysis tool to various parallel paradigm (MPI applications, UPC programs) paradigm-specific inefficiency patterns has been developed. In this paper code patterns resulting in performance penalties are discussed. Patterns of parallel MPI applications for parallel computing systems with distributed memory as well as for parallel UPC programs for systems with partial global address space (PGAS) are considered. A method for automatic detection of inefficiency patterns in parallel MPI applications and UPC programs is proposed. It allows to reduce the tuning time of parallel application.

Keywords: parallel programming, semantic errors, inefficiency patterns, MPI, UPC

References

[1]. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. MPI – The complete Reference, Volume 1, The MPI Core, Second edition. The MIT Press. 1998.
[2]. W. Chen, C. Iancu, K. Yelick. Communication Optimizations for Fine-grained UPC Applications. 14th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2005.
[3]. Sameer S. Shende Allen D. Malony. “The Tau Parallel Performance System”, International Journal of High Performance Computing Applications, Volume 20 , Issue 2, Pages: 287 – 311, May 2006.
[4]. L. Li and A.D. Malony, “Model-Based Performance Diagnosis of Master-Worker Parallel Computations,” Lecture Notes in Computer Science, Number 4128, Pages 35-46, 2006.
[5]. H. Su, M. Billingsley III, and A. George. "Parallel Performance Wizard: A Performance System for the Analysis of Partitioned Global Address Space Applications," International Journal of High-Performance Computing Applications, Vol. 24, No. 4, Nov. 2010, pp. 485-510.
[6]. Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, Bernd Mohr. The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience, 22(6):702–719, April 2010.

[7]. Felix Wolf. Automatic Performance Analysis on Parallel Computers with SMP Nodes. PhD thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003, ISBN 3-00-010003-2.
[8]. Wolf, F., Mohr, B. Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture 49(10-11) (2003) 421–439.
[9]. Wolf, F., Mohr, B., Dongarra, J., Moore, S. Efficient Pattern Search in Large Traces through Successive Refinement. In: Proc. European Conf. on Parallel Computing (Euro-Par, Pisa, Italy), Springer (2004).
[10]. MPICH. http://www.mpich.org
[11]. MVAPICH. http://mvapich.cse.ohio-state.edu.
[12]. OpenMPI. http://www.open-mpi.org.
[13]. Infiniband. http://www.infinibandta.org

Анализ эффективности итерационных методов решения систем линейных алгебраических уравнений, реализованных в пакете OpenFOAM

И. К. Марчевский, В. В. Пузикова
МГТУ им. Н.Э. Баумана
iliamarchevsky@mail.ru, valeria.puzikova@gmail.com

Аннотация. Для выбора оптимального в смысле вычислительной эффективности итерационного метода решения систем линейных алгебраических уравнений, возникающих при дискретизации дифференциальных уравнений в частных производных, помимо скорости сходимости следует учитывать такие характеристики системы и метода, как число обусловленности, коэффициент сглаживания, показатель «затратности». Последние две характеристики вычисляются по коэффициентам усиления гармоник, которые позволяют судить о сглаживающих свойствах итерационного метода и его «затратности», т.е. о том, насколько медленнее метод подавляет низкочастотные компоненты ошибки по сравнению с высокочастотными. Предложен способ определения коэффициентов усиления гармоник, основанный на использовании дискретного преобразования Фурье. В качестве примера приведён анализ эффективности метода BiCGStab с ILU и многосеточным предобуславливанием при решении разностных аналогов уравнений Гельмгольца и Пуассона, возникающих при моделировании течения вязкой несжимаемой жидкости в квадратной каверне.[1]

Ключевые слова. Разреженные линейные системы; предобуславливание; сглаживатели; коэффициенты усиления гармоник; многосеточные методы.

1. Введение

Как правило, значительная доля всего объёма вычислительной работы при численном моделировании технических систем, физических явлений и технологических процессов приходится на решение систем линейных алгебраических уравнений (СЛАУ), возникающих при дискретизации

[1] Работа проводилась при финансовой поддержке Министерства образования и науки Российской Федерации

соответствующих дифференциальных или интегро-дифференциальных уравнений.

Существует несколько классов итерационных методов решения СЛАУ [1], различающихся самим подходом к построению очередного итерационного приближения. Это методы, основанные на расщеплении, а также методы вариационного и проекционного типов. Необходимо отметить, что это исторически сложившееся деление весьма условно, поскольку любой итерационный метод можно записать в форме метода, основанного на расщеплении.

Во многих программных пакетах, используемых при решении широкого класса задач механики сплошной среды (ANSYS, NASTRAN, FLUENT, STAR-CD, COSMOSFloWorks и др.), имеется возможность выбора итерационного метода, который будет использоваться при проведении расчетов. В некоторых случаях методы решения линейных систем в программном пакете выбраны «по умолчанию» и их изменение пользователями осуществляется крайне редко, что не всегда благоприятным образом сказывается на времени решения задач. Особого внимания заслуживает открытый свободно распространяемый пакет OpenFOAM [2], представляющий собой платформу для решения задач механики сплошной среды. В нем установки «по умолчанию» отсутствуют, и требуется явно указывать методы решения линейных систем, возникающих в результате аппроксимации соответствующих уравнений. От того, насколько удачно будет выбран итерационный метод и, если необходимо, его параметры, во многом будет зависеть время выполнения расчета.

В пакете OpenFOAM реализованы следующие методы решения СЛАУ:

1) *релаксационные методы*:

а) метод Гаусса-Зейделя (GS);

б) метод неполного разложения Холецкого (DIC) /

метод неполного LU-разложения (DILU);

в) метод Гаусса-Зейделя с DIC-предобуславливанием (GS + DIC) /

метод Гаусса-Зейделя с DILU-предобуславливанием (GS + DILU);

2) проекционные методы:

а) метод сопряжённых градиентов (CG) / метод бисопряжённых градиентов (BiCG);

б) метод бисопряжённых градиентов со стабилизацией (BiCGStab) [3];

в) метод обобщенных минимальных невязок (GMRES);

3) *многосеточные методы*: геометро-алгебраический многосеточный метод (GAMG) и его модификации [4];

4) *диагональный решатель* (для диагональных систем).

Запись вида A / B означает, что метод A является частным случаем метода B для систем с симметричной матрицей (соответственно для систем с несимметричной матрицей применяется метод B). Курсивом выделены группы методов, которые могут быть использованы как предобуславливатели. Отметим, что методы BiCGStab и GMRES реализованы только в расширенной версии пакета (OpenFOAM-ext [5]).

Часто выбору итерационного метода решения СЛАУ уделяют мало внимания, оставляя неизменными те настройки, которые в соответствующих программных комплексах установлены по умолчанию, либо используются в похожих примерах в руководствах, как это часто делается при работе с OpenFOAM. Однако даже при решении простых тестовых задач время счёта можно сократить в несколько раз, если выбирать методы с учётом специфики конкретной задачи.

Целью данной работы является построение методики анализа вычислительной эффективности итерационных методов решения СЛАУ, возникающих при численном решении разностных аналогов уравнений механики сплошной среды, а также методики априорного выбора метода решения таких СЛАУ, обладающего высокой вычислительной эффективностью.

С практической точки зрения наиболее важной характеристикой итерационного метода решения СЛАУ является время, затрачиваемое на решение системы с заданной точностью. В то же время при теоретическом исследовании итерационных методов основное внимание, как правило, уделяется скорости сходимости, т.е. скорости уменьшения ошибки. Однако опыт применения различных итерационных методов при решении задач механики сплошной среды показывает, что прямой связи между скоростью сходимости метода и его вычислительной эффективностью для конкретной задачи может не быть. При этом речь идет не только о различающихся трудоемкостях выполнения одной итерации при использовании различных методов, но и о необходимости учета таких характеристик, как коэффициент сглаживания и показатель «затратности», которые определяются не только численным методом, но и решаемой задачей.

2. Постановка задачи

Рассмотрим систему линейных алгебраических уравнений

$$Ax = b \quad (x, b \in R^{N_d}, \quad \det A \neq 0), \qquad (1)$$

возникающую при дискретизации уравнения $Lu = f$, где L – дифференциальный либо интегро-дифференциальный оператор, f – известная функция, u – искомая функция. Будем считать, что матрица A

является разреженной и не обладает специальными свойствами (симметрией, положительной определенностью).

В данной работе в качестве примера будет рассмотрена модельная задача о расчёте течения вязкой несжимаемой жидкости в квадратной каверне, которое описывается уравнениями неразрывности и Навье-Стокса:

$$\begin{cases} \nabla \cdot \vec{v} = 0, \\ \dfrac{\partial \vec{v}}{\partial t} + (\vec{v} \cdot \nabla)\vec{v} + \nabla p = \dfrac{1}{\mathrm{Re}} \Delta \vec{v}. \end{cases}$$

Здесь Re – число Рейнольдса, $\vec{v}$ – скорость течения, p – давление; все величины являются безразмерными.

Задача решается в пространственной области $(x; y) \in [0;1] \times [0;1]$ при $t \geq 0$. На границах каверны (при $x = 0$, $x = 1$, $y = 0$ и $y = 1$) выполняются граничные условия прилипания и равенства нулю нормальной производной давления, при этом верхняя «крышка» каверны ($y = 1$) движется с постоянной скоростью $\vec{v}_\infty = (1;0)^{\mathrm{T}}$, остальные «стенки» неподвижны.

Давление в такой постановке определяется с точностью до константы, поэтому для выделения единственного решения в нижнем правом углу каверны давление полагается постоянным и равным константе. Разностная схема для решения задачи строится интегро-интерполяционным методом LS-STAG [6] на прямоугольной структурированной сетке.

На каждом шаге расчёта по времени решение задачи сводится к решению уравнения Гельмгольца

$$(\Delta + k^2) u_1 = f_1 \qquad (2)$$

для прогноза скорости u_1 и уравнения Пуассона

$$\Delta u_2 = f_2 \qquad (3)$$

для поправки давления u_2 (здесь Δ – оператор Лапласа). Разностные аналоги уравнений (2) и (3) представляют собой системы линейных алгебраических уравнений вида (1). Как показывает практика, при решении этих СЛАУ итерационными методами вычислительная сложность процедуры решения разностного аналога уравнения Пуассона (3) оказывается существенно выше, чем для уравнения Гельмгольца (2).

3. Предобуславливание

Поскольку любой итерационный метод решения системы линейных алгебраических уравнений (1) может быть представлен как метод, основанный на расщеплении, запишем его в виде

$$Mx^{n+1} = Nx^n + b, \quad M - N = A.$$

Здесь x^n – n-е итерационное приближение к искомому решению x. Пусть $r^n = b - Ax^n$ – вектор невязки, $z^n = x - x^n$ – итерационная ошибка, $p^n = x^{n+1} - x^n$ – вектор коррекции. Легко заметить, что закон изменения итерационной ошибки при переходе от итерации к итерации имеет следующий вид: $z^{n+1} = M^{-1}Nz^n$. Это соотношение показывает, что скорость сходимости определяется нормой матрицы $M^{-1}N$ перехода от итерации к итерации, которую можно оценить сверху при помощи спектрального радиуса [1, 4]. Ясно, что чем точнее M приближает A, тем выше скорость сходимости. Таким образом вводится понятие предобуславливания: если M близка к A в смысле некоторой нормы, она является предобуславливателем. Также отметим, что любой итерационный метод, основанный на расщеплении, может быть переписан как метод простой итерации (с параметром $\tau = 1$) для решения предобусловленной системы. Понятно, что можно использовать не один предобуславливатель, т.е. можно заменить метод простой итерации на другой итерационный метод и использовать его для решения предобусловленной системы. Например, на рис. 1 показано убывание нормы невязки при решении разностного аналога уравнения Пуассона (3) на сетке 16×16 методом Якоби без предобуславливания, методом простой итерации с ILU-предобуславливанием и методом Якоби с ILU-предобуславливанием [1]. Точность 10^{-16} выбрана исключительно для наглядности.

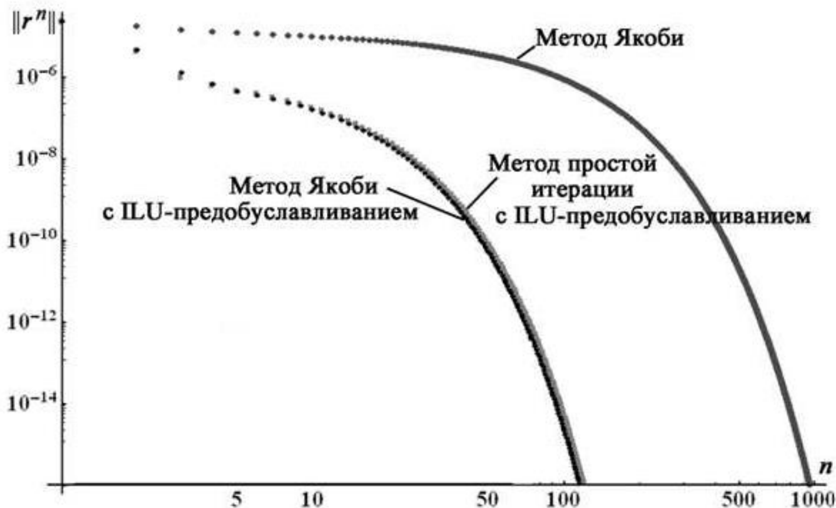


Рис. 1. Зависимость нормы невязки от номера итерации

В дальнейших примерах в качестве базового метода будем использовать метод BiCGStab [3], который относится к методам крыловского типа и обладает наиболее быстрой и гладкой сходимостью из всех методов данного класса. Алгоритм метода BiCGStab имеет вид:

$$r^0 = b - Ax^0,\ p^0 = r^0,\ \forall r_*^0 : (r_*^0, r^0) \neq 0$$

$$for\ \ n = 0,1,\ldots,\quad while\ (\| r^n \|_2 \geq \varepsilon),\ do$$

$$\alpha_n = \frac{(r_*^0, r^n)}{(Ap^n, r_*^0)}$$

$$s^n = r^n - \alpha_n Ap^n$$

$$\omega_n = \frac{(As^n, s^n)}{(As^n, As^n)}$$

$$x^{n+1} = x^n + \alpha_n p^n + \omega_n s^n$$

$$r^{n+1} = s^n - \omega_n As^n$$

$$\beta_n = \frac{\alpha_n (r^{n+1}, r_*^0)}{\omega_n (r^n, r_*^0)}$$

$$p^{n+1} = r^{n+1} + \beta_n (p^n - \omega_n Ap^n).$$

здесь p^n – вектор коррекции, r^n – вектор невязки на n-й итерации. При использовании правого предобуславливания [1] вместо исходной системы $Ax = b$ последовательно решаются системы $AM^{-1}y = b$ и $Mx = y$. Такой последовательности действий соответствует алгоритм метода BiCGStab с предобуславливанием:

$$r^0 = b - Ax^0,\ p^0 = r^0,\ \forall r_*^0 : (r_*^0, r^0) \neq 0$$

$$\mathit{for} \quad n = 0,1,\ldots, \quad \mathit{while}\ (\| r^{n+1} \|_2 \geq \varepsilon),\ \mathit{do}$$

$$y^n = M^{-1} p^n$$

$$\alpha_n = \frac{(r_*^0, r^n)}{(Ay^n, r_*^0)}$$

$$s^n = r^n - \alpha_n A y^n$$

$$z^n = M^{-1} s^n$$

$$\omega_n = \frac{(Az^n, s^n)}{(Az^n, Az^n)}$$

$$x^{n+1} = x^n + \alpha_n y^n + \omega_n z^n$$

$$r^{n+1} = s^n - \omega_n A z^n$$

$$\beta_n = \frac{\alpha_n (r^{n+1}, r_*^0)}{\omega_n (r^n, r_*^0)}$$

$$p^{n+1} = r^{n+1} + \beta_n (p^n - \omega_n A y^n).$$

Видно, что отличие от исходного алгоритма состоит только в двух шагах ($y^n = M^{-1} p^n$ и $z^n = M^{-1} s^n$); при этом явного обращения матрицы M при проведении вычислений не производится, вместо этого решаются системы $My^n = p^n$ и $Mz^n = s^n$. На рис. 2 показано, что предобуславливание позволяет многократно увеличить скорость сходимости метода BiCGStab как для СЛАУ с симметричной матрицей, так и с несимметричной матрицей.

При этом существенный выигрыш получается не только по числу итераций (табл. 1), но и по числу умножений (табл. 2). Отметим, что α ILU-предобуславливанием здесь называется модификация ILU-предобуславливания, для которой элементы матрицы U вычисляются по

формуле $u_{kj} = a_{kj} - (1+\alpha)\sum_{i=1}^{k-1} l_{kj} u_{ij}$ [7]. Использование такой модификации целесообразно, если имеется серия однотипных задач: на одной из них подбирается оптимальный параметр α, что позволяет сократить вычислительную трудоемкость процедуры решения по сравнению с «базовым» вариантом (при $\alpha = 1$) примерно на 10 %.

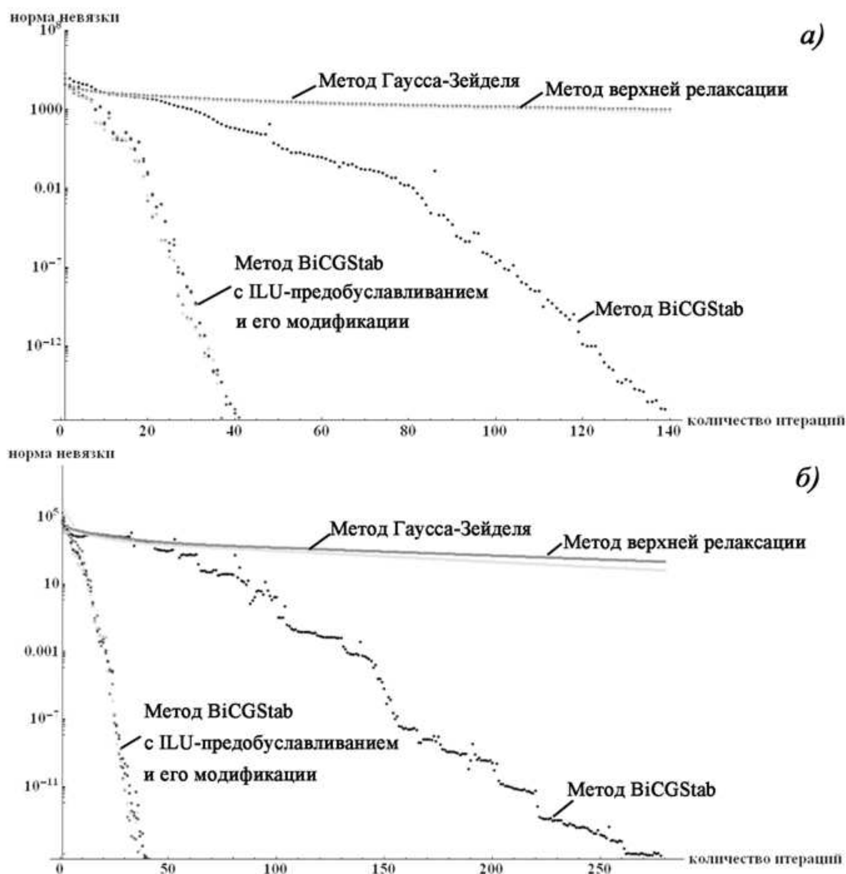


Рис. 2. Убывание норм невязок при решении различными методами тестовых СЛАУ 809×809: *а)* симметричной; *б)* несимметричной.

Табл. 1. Количество итераций при при решении нескольких тестовых СЛАУ размерности 809×809 различными итерационными методами.

Метод	Количество итераций			
	«Тест 1»	«Тест 2»	«Тест 3»	«Тест 4»
Метод Зейделя	3313	3069	2987	3015
Метод верхней релаксации ($w=1.4$)	2384	2241	2168	2219
BiCGStab	139	138	240	283
BiCGStab+ ILU	41	38	30	40
BiCGStab+ ILU (ван дер Ворст)	40	42	30	42
BiCGStab+ αILU	37 ($\alpha=0.15$)	36 ($\alpha=0.11$)	27 ($\alpha=0.3$)	37 ($\alpha=0.1$)
BiCGStab+ αILU (ван дер Ворст)	37 ($\alpha=0.15$)	36 ($\alpha=0.11$)	27 ($\alpha=0.3$)	38 ($\alpha=0.05$)

Табл. 2. Количество умножений при решении тестовых СЛАУ размерности 809×809 различными итерационными методами (M_e – число ненулевых элементов в матрице СЛАУ).

Метод	Количество умножений				
	в 1 итерацию	«Тест 1»	«Тест 2»	«Тест 3»	«Тест 4»
Метод Зейделя	M_e	17 754 367	16 152 147	14 388 379	15 038 820
Метод верхней релаксации ($w=1.4$)	M_e	12 775 856	11 794 383	10 443 256	11 068 372
BiCGStab	$2M_e + 11N_d$	2 726 763	2 680 650	4 447 920	5 341 625
BiCGStab+ ILU	$4M_e + 9N_d$	1 177 397	1 076 654	796 470	1 089 320
BiCGStab+ ILU (ван дер Ворст)	$5M_e + 7N_d$	1 298 320	1 343 076	892 440	1 285 326
BiCGStab+ αILU	$4M_e + 9N_d$	1 062 529	1 019 988	716 823	1 007 621
BiCGStab+ αILU (ван дер Ворст)	$5M_e + 7N_d$	1 200 946	1 151 208	803 196	1 162 914

4. Анализ изменения итерационной ошибки и коэффициенты усиления гармоник

Для дальнейшего анализа эффективности методов необходимо ввести некоторые дополнительные понятия. Представим вектор итерационной ошибки z^n в виде разложения по гармоникам, возникающим из решения разностного аналога одномерной спектральной задачи

$$-u''(x) = \lambda u(x)\,, \quad x \in (0,1)\,; \quad u(0) = u(1) = 0 \qquad (4)$$

на равномерной сетке с шагом h. Собственные векторы матрицы СЛАУ, соответствующей разностному аналогу задачи (4), – гармоники – имеют вид

$$\psi_j^k = \sin \pi kjh\,, \quad j = \overline{1, N_d}\,, \quad k = \overline{1, N_d}\;.$$

Таким образом, гармоники с номерами $k > N_d$ неразрешимы на такой сетке, поскольку длина волны для любой такой гармоники меньше $2h$. Остальные гармоники (с номерами $1 \le k \le N_d$) естественным образом делятся на низкочастотные $(1 \le k \le [N_d / 2])$ и высокочастотные ($[N_d / 2] + 1 \le k \le N_d$; запись $[\cdot]$ означает целую часть числа). Низкочастотные гармоники являются длинноволновыми (длина волны больше $4h$), а высокочастотные – коротковолновыми.

Наличие длинноволновых и коротковолновых компонент определяет жёсткость задачи, заключающуюся в различной скорости убывания соответствующих компонент ошибки ($z^{n+1} = M^{-1} N z^n$) при выполнении одной итерации. Скорость убывания *p*-й компоненты ошибки можно оценить коэффициентом усиления соответствующей гармоники

$$g_p = \sum_{k=1}^{N_d} \lambda_k^{M^{-1}N} \left| \dot{\Psi}_{p+1}^k \right|, \quad p = \overline{1, N_d}\,,$$

где $\lambda_k^{M^{-1}N}$ – k-е собственное число матрицы $M^{-1}N$ (предполагаем, что все $\lambda_k^{M^{-1}N}$ действительные), $\dot{\Psi}^k$ – дискретное преобразование Фурье (ДПФ) [8] k-го собственного вектора матрицы $M^{-1}N$, дополненного нулями.

Такие итерационные методы, как метод Якоби, метод Гаусса – Зейделя и т.п., редко используются для решения жёстких задач, поскольку за первые

несколько итераций они практически полностью устраняют высокочастотные компоненты ошибки, тогда как низкочастотные подавляются очень медленно. Таким образом, указанные методы действуют как фильтры низких частот, поэтому их называют сглаживателями, или релаксационными методами, а максимальный коэффициент усиления высокочастотных гармоник – коэффициентом сглаживания ρ_{sm} [4].

При больших N_d вычисление собственных чисел, собственных векторов и их ДПФ превращается в громоздкую задачу, поэтому для рассматриваемой СЛАУ и итерационного метода важно иметь характеристику, которая определяется видом дискретизируемого оператора и мало зависит от величины шага по пространству. Как показывают вычислительные эксперименты, такой характеристикой является отношение наибольшего коэффициента усиления низкочастотных гармоник к коэффициенту сглаживания, который обозначим β и назовём показателем «затратности» итерационного метода для рассматриваемой задачи, поскольку он показывает, во сколько раз медленнее подавляются низкочастотные гармоники по сравнению с высокочастотными, т.е. насколько плохо метод преодолевает жёсткость задачи. Чем ближе β к единице и чем ближе к нулю спектральный радиус матрицы $M^{-1}N$ перехода от итерации к итерации, тем лучше метод подходит для решения данной задачи.

5. Вычислительные эксперименты

Как было отмечено выше, в качестве примеров СЛАУ рассмотрим разностные аналоги уравнений Гельмгольца (2) и Пуассона (3), полученные при решении хорошо известной тестовой задачи о моделировании течения в каверне методом погруженных границ с функциями уровня (LS-STAG) [6]. Как видно из табл. 3, для решения разностного аналога уравнения Гельмгольца рассматриваемый метод BiCGStab с ILU-предобуславливанием весьма эффективен, в то время как для плохо обусловленной системы, полученной при дискретизации уравнения Пуассона, спектральный радиус $\rho(M^{-1}N)$ матрицы перехода от итерации к итерации и показатель «затратности» β далеки от нуля и единицы соотвественно, поэтому для численного решения второго уравнения необходимо подобрать более эффективный метод.

Табл. 3. Характеристики метода BiCGStab с ILU-предобуславливанием при решении разностных аналогов уравнений Гельмгольца и Пуассона.

Разностный аналог уравнения	Гельмгольца			Пуассона		
Размерность СЛАУ (N_d)	12	56	240	16	64	256
Число обусловленности	1.00	1.00	1.00	85.88	589.89	3292.74
$\rho(M^{-1}N)$	$1.5\cdot10^{-9}$	$2.8\cdot10^{-8}$	$1.7\cdot10^{-6}$	0.82	0.97	0.99
ρ_{sm}	$1.7\cdot10^{-9}$	$7.3\cdot10^{-8}$	$1.5\cdot10^{-5}$	0.13	0.24	0.30
β	1.01	1.14	1.09	3.34	3.25	3.32

Поскольку наиболее трудоемкой оказывается «борьба» с низкочастотными составляющими ошибки, весьма перспективной представляется многосеточная стратегия. Сначала выполняется несколько сглаживающих итераций, подавляющих высокочастотные составляющие, а затем осуществляется переход на более грубую сетку, на которой высокочастотные гармоники неразрешимы, а половина низкочастотных становятся высокочастотными для нового сеточного уровня. На самой грубой сетке задача, как правило, решается методом Гаусса, а затем решение переносится на исходную сетку. При переносе на подробную сетку выполняются постсглаживающие итерации.

Многосеточный метод [4, 9–11] является хорошим предобуславливателем. Здесь для примера рассмотрим модификацию многосеточного метода, описанную в [7]. В ней присутствуют три параметра, значения которых можно варьировать. Это параметр релаксации ω в используемом в качестве сглаживателя методе ADLJ, число предсглаживаний n_{pre} и число постсглаживаний n_{post}. В [12] указано, что при решении СЛАУ, возникающих при дискретизации эллиптических уравнений в двумерном случае, для подобного решателя оптимальными будут следующие параметры: $\omega \approx 0.71$, $n_{pre} = 0$, $n_{post} = 1$ или $n_{post} = 2$. Однако, рассчитав такие характеристики данного итерационного метода при решении разностного аналога уравнения Пуассона, как спектральный радиус $\rho(M^{-1}N)$ и показатель «затратности» β, был сделан вывод о том, что в двумерном

случае наилучшим выбором являются такие значения параметров: $\omega \approx 0.86$, $n_{pre} = 1$, $n_{post} = 1$. При них наблюдается наименьший спектральный радиус матрицы перехода от итерации к итерации $M^{-1}N$ при близком к единице показателе «затратности». Собственные числа матрицы перехода от итерации к итерации и коэффициенты усиления гармоник для данных параметров представлены на рис. 3.

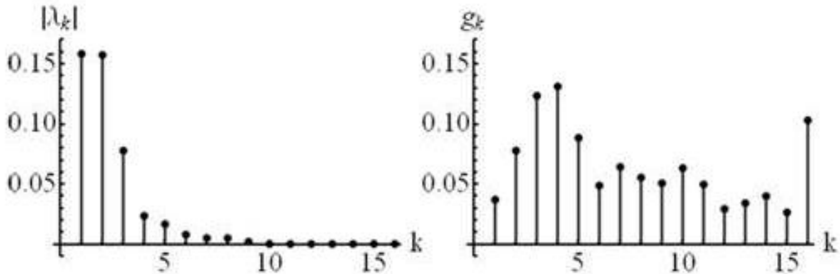


Рис. 3. Собственные числа матрицы перехода от итерации к итерации и коэффициенты усиления гармоник при решении разностного аналога уравнения Пуассона на сетке 16×16 методом BiCGStab с многосеточным предобуславливанием ($\omega \approx 0.86$, $n_{pre} = 1$, $n_{post} = 1$).

Отметим, что при $N_d = 17760$ и ограничении на норму невязки $\varepsilon = 10^{-6}$ на первом шаге по времени система решается методом BiCGStab с ILU-предобуславливанием за 158 итераций, а методом BiCGStab с многосеточным прелобуславливанием с указанными выше параметрами – за 9. Для сравнения, если использовать параметры, приведенные в [12] ($\omega \approx 0.71$, $n_{pre} = 0$, $n_{post} = 1$), число итераций увеличивается до 11, при иных параметрах ($\omega \approx 0.10$, $n_{pre} = 0$, $n_{post} = 2$), число итераций увеличивается до 23.

Решая тестовую задачу о моделировании течения в квадратной каверне в пакете OpenFOAM на сетке 100×100, наименьшее время счёта также получается при использовании ILU-предобуславливания для разностного аналога уравнения Гельмгольца и многосеточного предобуславливания для разностного аналога уравнения Пуассона (рис. 4).

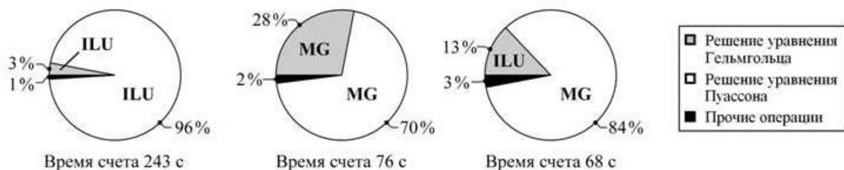


Рис. 4. Временные затраты при решении тестовой задачи о моделировании течения в каверне методом BiCGStab с различными предобуславливателями (OpenFOAM, $N_d = 10000$*).*

6. Заключение

Определение коэффициентов усиления гармоник позволяет получить ряд важных характеристик итерационного метода, по которым можно судить о его эффективности при решении той или иной задачи. Для определения этих коэффициентов предлагается использовать дискретное преобразование Фурье. Как показывают вычислительные эксперименты, для итерационного метода отношение наибольшего коэффициента усиления низкочастотной гармоники к наибольшему коэффициенту усиления высокочастотной гармоники, названное показателем «затратности», определяется видом дискретизируемого оператора и мало зависит от N_d. Благодаря введению такой характеристики, которая может быть вычислена при решении задачи малой размерности (на грубой сетке), стало возможным определение оптимальных параметров при использовании многосеточного предобуславливания.

Работа проводится при финансовой поддержке Министерства образования и науки Российской Федерации

Список литературы

[1]. Saad Y. Iterative Methods for Sparse Linear Systems. N.Y.: PWS Publ., 1996. 547 p.
[2]. OpenFOAM // The OpenFOAM Foundation. URL: http://www.openfoam.org (дата обращения: 30.04.2013).
[3]. Van der Vorst H.A. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for solution of non-symmetric linear systems// SIAM J. Sci. Stat. Comp. 1992. № 2. P. 631-644.
[4]. Wesseling P. An introduction to multigrid methods. Chichester: John Willey & Sons Ltd., 1991. 284 p.
[5]. The OpenFOAM® Extend Project. URL: http://www.extend-project.de (дата обращения: 30.04.2013).

[6]. Cheny Y., Botella O. The LS-STAG method: A new immersed boundary/level-set method for the computation of incompressible viscous flows in complex moving geometries with good conservation properties // J. Comput. Phys. 2010. № 229. P. 1043-1076.
[7]. Пузикова В.В. Решение систем линейных алгебраических уравнений методом BiCGStab с предобуславливанием // Вестник МГТУ им. Н.Э. Баумана. Естественные науки. 2011. Спец. выпуск «Прикладная математика». С. 124-133.
[8]. Сергиенко А.Б. Цифровая обработка сигналов. СПб.: Питер, 2002. 608 с.
[9]. Федоренко Р.П. Введение в вычислительную физику. М.: Изд-во Моск. физ.-техн. ин-та, 1994. 528 с.
[10]. Галанин М.П., Савенков Е.Б. Методы численного анализа математических моделей. М.: Изд-во МГТУ им. Н.Э. Баумана, 2010. 590 с.
[11]. Ольшанский М.А. Лекции и упражнения по многосеточным методам. М.: Изд-во ЦПИ при механико-математическом факультете МГУ им. М.В. Ломоносова, 2003. 163 с.
[12]. Van Kan J., Vuik C., Wesseling P. Fast pressure calculation for 2D and 3D time dependent incompressible flow // Numer. Linear Algebra Appl. 2000. № 7. P. 429-447.

OpenFOAM iterative methods efficiency analysis for linear systems solving

I. K. Marchevsky, V. V. Puzikova
iliamarchevsky@mail.ru, valeria.puzikova@gmail.com
Bauman Moscow State Technical University, Moscow, Russia
105005 Russia, Moscow, 2nd Baumanskaya st., 5

Abstract. Significant part of the computational work in numerical simulations of technical systems, physical phenomena and technological processes is solving of linear algebraic equations systems arising from the discretization of the corresponding differential or integro-differential equations. There are several classes of iterative methods for linear algebraic equations systems solving, which differ by the approach to the construction of the next iterative approximation. These classes are methods based on splitting, variational-type methods and projection-type methods. The aim of this study is the approach development for computational efficiency analysis of iterative methods for linear algebraic equations systems solving, which are difference analogues of continuum mechanics equations, and the approaches for method a'priori choice for linear algebraic equation solving with high computational efficiency. To choose the optimal numerical method for linear systems solving, in addition to the rate of convergence such characteristics of a linear system and numerical method, as condition number, smoothing factor and cost-coefficient should be considered. The smoothing factor and cost-coefficient can be computed through the amplification factors of the modes. The performance of a smoothing method is measured by its smoothing factor, but the cost of a numerical method is measured through its cost-coefficient which shows the difference between amplitudes vanishing speeds of smooth modes and rough modes. The method for modes amplification factors computing using the discrete Fourier transform is proposed. The cost-coefficient usage allows to choose the optimal parameters of the multigrid preconditioner. Some test problems are considered and the efficiency of BiCGStab (BiConjugate Gradient Stabilized) method with the Incomplete LU and multigrid preconditioners is investigated for linear systems solving which follow from discrete forms of Helmholtz and Poisson equations. These linear algebraic equations systems arise in numerical simulation of incompressible viscous flow in a square cavity by using the LS-STAG cut-cell immersed boundary method with level-set function.

Keywords. Sparse linear systems; preconditioning; smoothers; modes amplification factors; multigrid methods.

References

[1]. Saad Y. Iterative Methods for Sparse Linear Systems. New York, PWS Publ., 1996. 547 p.
[2]. OpenFOAM // The OpenFOAM Foundation. URL: http://www.openfoam.org.
[3]. Van der Vorst H.A. Bi-CGSTAB: a Fast and Smoothly Converging Variant of Bi-CG for Solution of Non-Symmetric Linear Systems. SIAM J. Sci. Stat. Comp., 1992, no. 2, pp. 631-644.
[4]. Wesseling P. An Introduction to Multigrid Methods. Chichester, John Willey & Sons Ltd., 1991. 284 p.

[5]. The OpenFOAM® Extend Project. URL: http://www.extend-project.de.
[6]. Cheny Y., Botella O. The LS-STAG Method: A New Immersed Boundary/Level-Set Method for the Computation of Incompressible Viscous Flows in Complex Moving Geometries with Good Conservation Properties. J. Comput. Phys., 2010, no. 229, pp. 1043-1076.
[7]. Puzikova V.V. Reshenie sistem linejnykh algebraicheskikh uravnenij metodom BiCGStab s predobuslavlivaniem [Solving of Linear Algebraic Equation Systems Using the BiCGStab Method with Preconditioner]. Vestnik MGTU im. N.EH. Baumana. Estestvennye nauki [Herald of the Bauman Moscow State Technical University. Natural Sciences], 2011, Spets. vypusk «Prikladnaya matematika» [Special issue "Applied mathematics"], pp. 124-133 (in Russian).
[8]. Sergienko A.B. TSifrovaya obrabotka signalov [Digital Signal Processing]. SPb.: Piter [Saint Petersburg, Piter], 2002. 608 p.
[9]. Fedorenko R.P. Vvedenie v vychislitel'nuyu fiziku [Introduction to Computational Physics]. M.: Izd-vo Mosk. fiz.-tekhn. in-ta [Moscow, Publishing house of Moscow Physics and Technology Institute], 1994. 528 p.
[10]. Galanin M.P., Savenkov E.B. Metody chislennogo analiza matematicheskikh modelej [Methods of Numerical Analysis of Mathematical Models]. M.: Izd-vo MGTU im. N.EH. Baumana [Moscow, Publishing houce of the Bauman Moscow State Technical University], 2010. 590 p.
[11]. Ol'shanskij M.A. Lektsii i uprazhneniya po mnogosetochnym metodam [Lectures and Exercises on Multigrid Methods]. M.: Izd-vo TSPI pri mekhaniko-matematicheskom fakul'tete MGU im. M.V. Lomonosova [Moscow, Publishing house at the Mechanics and Mathematics Faculty of Lomonosov Moscow State University], 2003. 163 p.
[12]. Van Kan J., Vuik C., Wesseling P. Fast Pressure Calculation for 2D and 3D Time Dependent Incompressible Flow. Numer. Linear Algebra Appl., 2000, no. 7, pp. 429-447.

Расчет течений непрерывно стратифицированной жидкости с использованием открытых вычислительных пакетов на базе технологической платформы UniHUB

Загуменный Ярослав Викторович
(Институт гидромеханики НАН Украины, Киев),
e-mail: zagumennyi@gmail.com
Юлий Дмитриевич Чашечкин (Институт проблем механики РАН, Москва),
e-mail: yulidch@gmail.com

Аннотация. В работе представлен авторский опыт использования технологической платформы UniHUB при численном моделировании и проведении расчетов течений непрерывно стратифицированной жидкости с применением свободных прикладных вычислительных пакетов OpenFOAM, Salome и ParaView. Внимание уделяется вопросам построения высокоразрешающих расчетных сеток, постановки сложных граничных с помощью встроенных и расширенных утилит пакета OpenFOAM, разработки собственных решателей, обработки и визуализации расчетных данных, а также проведения расчетов в параллельном режиме на вычислительном кластере МСЦ РАН. Демонстрируются некоторые результаты численных расчетов – картины стратифицированных течений около непроницаемых наклонной и горизонтальной пластин, симметричного клина, горизонтального диска и кругового цилиндра. Приводятся сравнения данных аналитического, численного и лабораторного моделирования стратифицированных течений около различных препятствий, оговариваются условия согласия и расхождения результатов, обсуждаются проблемы переноса данных моделирования на природные системы.

Ключевые слова: численное моделирование; открытые вычислительные пакеты; стратифицированные течения; тонкая структура; внутренние волны; теневая визуализация

1. Введение

Данная работа посвящена одному из наиболее активно развиваемых направлений – построению и численной реализации адекватных моделей природных процессов, протекающих в атмосфере, гидросфере, геосфере Земли, и, в частности, исследованию волн и вихрей – одному из

традиционных разделов классической механики жидкости, сохраняющему свою актуальность на протяжении последних столетий.

Интерес к теме поддерживается логикой развития научных исследований – волновая тематика служит одним из стимулов развития фундаментальной и прикладной математики и объектом интенсивным экспериментальных исследований. Стимулирующими факторами являются и практические потребности – разрушения и катастрофы волновой природы (цунами, наводнения, волны экстремальной высоты) наносят наиболее ощутимый экономический ущерб и сопровождаются человеческими жертвами. Пристальное внимание на сегодняшний день также уделяется решению экологических проблем – развитию технологий мониторинга и прогнозирования состояния окружающей среды, предотвращения и ликвидации ее загрязнения. Не теряет своей актуальности и проблема управления аэродинамическим качеством крыловых профилей или иных аэродинамических компоновок ввиду всевозрастающей необходимости экономичного расходования энергоресурсов.

Заметный прогресс в решении перечисленных проблем в последние годы обусловлен развитием компьютерных технологий, которые позволили реализовать более точные методы построения решений и высокоразрешающие численные модели, в которых все большее внимание уделяется изучению влияния эффектов нелинейности, стратификации, вращения, диффузии, теплопроводности, как по отдельности, так и в общей постановке. Учет тонкоструктурных эффектов обычно вносит сравнительно небольшие поправки в локальные значения динамических и кинематических характеристик течений, однако в интегральных масштабах могут давать существенные корректировки в результаты расчетов, проведенных на основе более грубых приближений.

Как правило, жидкости и газы в природных системах неоднородны по составу и концентрации растворенных веществ и взвешенных частиц, температуре, давлению, и, как следствие, плотности. Действие массовых сил (гравитационных, центробежных, электрических) ведет к перераспределению веществ и формированию стратификации – увеличению плотности среды в выделенном направлении. Устойчивая стратификация обеспечивает существование тонкоструктурных течений, индуцированных диффузией на неровностях рельефа [1–6, 12], внутренних волн [7, 8], высокоградиентных квазистационарных областей, разделяющих различные виды возмущений, в частности, волн и вихрей. И расчет, и измерения всех макро- и микрокомпонент течений представляют сложные задачи, все еще не решенные с практически необходимой степенью точности [4–6, 8].

Адекватное математическое моделирование течений жидких сред накладывает необходимость использования фундаментальной системы уравнений механики неоднородных жидкостей, учитывающей влияние свойств среды и градиентов термодинамических потенциалов на структуру и динамику течений. Анализ свойств линеаризованных фундаментальных уравнений и

результаты прецизионного лабораторного моделирования показывают, что полные решения содержат как регулярные возмущенные компоненты, характеризующие волны или вихри, так и сингулярные, описывающие сопутствующие тонкоструктурные элементы течений, которые в приближении однородной жидкости теряют индивидуальные признаки и становятся тождественными, что ограничивает возможности расчета течений [9].

При проведении расчетов на основе фундаментальной системы уравнений механики неоднородных жидкостей полный учет диссипативных факторов затрудняет разработку и реализацию численных схем и предъявляет высокие требования к быстродействию и объему памяти компьютера, вынуждая программиста в ряде случаев пренебрегать эффектами диффузии или загрублять исходную расчетную сетку, что часто приводит к появлению численных погрешностей в тех областях, где в лабораторных наблюдениях выражены тонкоструктурные вихревые течения [6]. Поэтому для обеспечения достаточной точности и высокого пространственного разрешения численных расчетов необходимо прибегать к использованию высокопроизводительных суперкомпьютерных систем с применением различных технологий распараллеливания алгоритмов. Одной из наиболее удобных, эффективных и перспективных вычислительных платформ, предназначенных для моделирования и проведения расчетов задач механики сплошных сред, является виртуальная вычислительная лаборатория UniHUB, созданная на базе вычислительного кластера МСЦ РАН.

Целью данной работы является развитие методики численного моделирования на основе системы фундаментальных уравнений механики неоднородных жидкостей и выполнение расчетов стратифицированных течений около препятствий выбранной формы в физически достижимом диапазоне параметров задачи с учетом переноса вещества, как стратифицирующего компонента, так и визуализирующей примеси. Достоинство такого подхода – возможность получения результатов в естественных физических переменных без привлечения дополнительных констант или связей, а также возможность одновременного изучения крупных и наиболее тонких структурных компонент в рамках единого описания.

2. Постановка задачи

Рассматривается система уравнений механики несжимаемой линейно стратифицированной жидкости, невозмущенное распределение плотности которой $\rho_0(z)$ задается профилем солености $S_0(z)$ (где ось $0z$ направлена вертикально вверх, $\Lambda = (d \ln \rho_0 / dz)^{-1}$ – масштаб, $N = 2\pi/T_b = \sqrt{g/\Lambda}$ – частота и T_b – период плавучести, $\mathbf{g}$ – ускорение свободного падения), включающая уравнения состояния $\rho(S(z))$, неразрывности, Навье-Стокса в

приближении Буссинеска, диффузии стратифицирующей компоненты и визуализирующей примеси:

$$\rho = \rho_{00}\left(\exp\left(-z/\Lambda\right) + s + s_0\right), \ \operatorname{div}\mathbf{v} = 0,$$

$$\frac{\partial \mathbf{v}}{\partial t} + \left(\mathbf{v}\nabla\right)\mathbf{v} = -\frac{1}{\rho_{00}}\nabla P + \nu\Delta\mathbf{v} - \left(s + s_0\right)\mathbf{g}, \qquad (2.1)$$

$$\frac{\partial s}{\partial t} + \mathbf{v}\cdot\nabla s = \kappa_S \Delta s + \frac{\mathrm{v}_z}{\Lambda}, \quad \frac{\partial s_0}{\partial t} + \mathbf{v}\cdot\nabla s_0 = \kappa_{S_0}\Delta s_0 \ .$$

Здесь s – возмущение солености (стратифицирующего компонента), включающее коэффициент солевого сжатия, s_0 – концентрация пассивной примеси, $\mathbf{v} = \left(\mathrm{v}_x, \mathrm{v}_y, \mathrm{v}_z\right)$ – индуцированная скорость, P – давление за вычетом гидростатического, ν, κ_S и κ_{S_0} – коэффициенты кинематической вязкости, диффузии соли и пассивной примеси, соответственно, t – время, ∇ и Δ – операторы Гамильтона и Лапласа.

Физически обоснованные начальные условия и граничные условия задачи (прилипания для скорости и непротекания для вещества) имеют вид

$$\mathbf{v},\ s\big|_{t\le 0} = 0, \quad \mathrm{v}_x\big|_\Sigma = \mathrm{v}_z\big|_\Sigma = 0, \quad \left[\frac{\partial s}{\partial \mathbf{n}}\right]\Bigg|_\Sigma = \frac{1}{\Lambda}\frac{\partial z}{\partial \mathbf{n}}, \quad \mathbf{v},\ s\big|_{x,\, z\to\infty} = 0 \qquad (2.2)$$

где $\mathbf{n}$ – внешняя нормаль к поверхности препятствия Σ. На бесконечности все возмущения затухают.

Система (2.1) с начальными и граничными условиями (2.2) характеризуется большим числом собственных масштабов: длины (плавучести Λ, характерного размера препятствия L, скоростного $\delta_N^\nu = \sqrt{\nu/N}$ и диффузионного $\delta_N^{\kappa_S} = \sqrt{\kappa_S/N}$ микромасштабов); скорости ($U_N^\nu = \sqrt{\nu N}$, $U_N^{\kappa_S} = \sqrt{\kappa_S N}$) и времени $t = T_b$. Существенные различия в значениях масштабов длины указывают на сложность внутренней структуры даже такого медленного течения, которое порождается малыми силами плавучести, возникающими вследствие неоднородности молекулярного потока стратифицирующего компонента.

3. Метод решения

Поставленная задача решается путем прямого численного моделирования с использованием свободно распространяемого инструментария

вычислительной гидродинамики OpenFOAM с открытым исходным кодом. На основе стандартного решателя icoFoam, предназначенного для расчета течений вязкой несжимаемой однородной жидкости методом конечного объема, был разработан собственный солвер stratifiedFlow, позволяющий расчитывать течения непрерывно стратифицированных жидкостей в соответствии с системой дифференциальных уравнений (2.1). Новый решатель реализован путем введения дополнительных переменных и соответствующих уравнений для их расчета – переменной плотности, концентрации стратифицирующего компонента и визуализирующей примеси, а также новых параметров – частоты плавучести, масштаба стратификации, коэффициентов диффузии, ускорения свободного падения и других вспомогательных параметров, управляющих учетом эффектов нелинейности, стратификации и диффузии. В уравнение Навье-Стокса для вертикальной компоненты скорости введен дополнительный член, учитывающий влияние стратифицирующего компонента, а в уравнение для возмущения солености – дополнительные слагаемые, определяющие фоновую стратификацию и диффузию растворенного вещества (см. систему (2.1)). С учетом введенных модификаций также были внесены необходимые изменения в алгоритм PISO для расчета поля давления и удовлетворения условия несжимаемости. На базе основных рассчитываемых переменных – компонент скорости, давления, плотности, солености, с использованием встроенных утилит OpenFOAM проводились расчеты других не менее важных физических величин – полной плотности, функции тока, динамической завихренности, скорости бароклинной генерации завихренности, компонент тензора вязких напряжений, скорости диссипации механической энергии, распределения различных динамических характеристик, воздействующих на препятствие.

Построение расчетных сеток осуществлялось как на основе стандартных утилит blockMesh, snappyHexMesh, так и с использованием независимого сеточного генератора Salome. Процедура построения структурированных расчетных сеток с помощью утилиты blockMesh была автоматизирована благодаря применению макропроцессора m4, который позволил достаточно быстро перестраивать сетку при изменении геометрических параметров расчетной области и препятствия – пластины, диска, клина либо цилиндра. С целью адекватного разрешения диффузионных микромасштабов вблизи непроницаемых границ препятствий и областей, в которых выражены тонкоструктурные вихревые компоненты течений, дополнительно использовались стандартные утилиты topoSet и refineMesh, позволяющие на основе геометрических либо параметрических признаков выделять подобласти расчетной сетки и измельчать их в соответствии с заданными масштабами и выбранными направлениями.

Для рассматриваемого класса задач механики неоднородных многокомпонентных жидкостей важным элементом является постановка граничных условий, которые оказываются зависящими как от направления локальной нормали к границе препятствия, так и от значения глобальных

координат элемента поверхности. Для задания неоднородных граничных условий для нормального градиента возмущения солености на непроницаемой поверхности и источника визуализирующей примеси, размещаемого либо в толще жидкости, либо на границе препятствия, были использованы стандартные и расширенные утилиты пакета OpenFOAM, такие как setFields, topoSets (с дополнительным использованием расширенных библиотек swakSourceFields и swakTopoSources), swak4Foam, groovyBC, funkySetFields, funkySetBoundaryField, которые позволяют задавать аналитические выражения для различных физических переменных с использованием программы GNU bison, предназначенной для автоматического создания синтаксических анализаторов по данному описанию грамматики.

Расчеты проводились в параллельном режиме с использованием различных методов декомпозиции расчетной области на вычислительном сегменте кластера МСЦ РАН на базе технологической платформы UniHUB [10]. Визуализация результатов выполнялась с использованием графического интерфейса ParaView, а также пакета программ для численного анализа и научной графики Origin, в рамках которого был реализован высокоразрешающий метод построения полихромной карты изолиний.

4. Результаты расчетов

В качестве иллюстрации работоспособности разработанного решателя пакета OpenFOAM в данной работе приводятся результаты расчета течений, индуцированных диффузией на непроницаемой пластине, симметричном клине, горизонтальном диске и круговом цилиндре. Такие течения формируются в результате прерывания молекулярного потока стратифицирующего компонента на непроницаемой границе и, как следствие, образования дефицита (избытка) плотности над (под) неподвижным препятствием, погруженным в непрерывно стратифицированную жидкость.

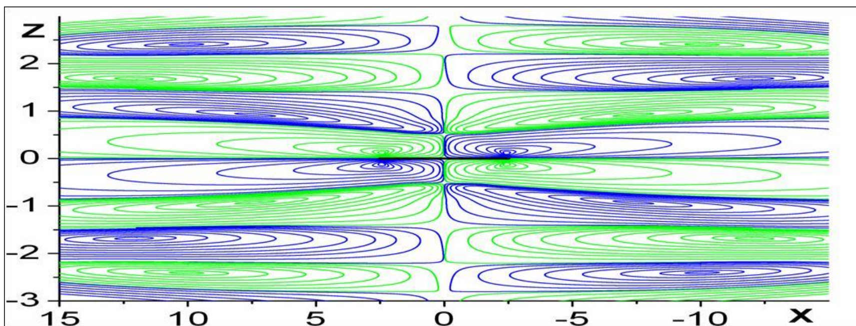
Впервые на возможность формирование течения на неподвижной наклонной поверхности в покоящейся устойчиво стратифицированной воздушной среде указал Л.Прандтль применительно к задаче формирования горных (долинных) ветров [1]. Устойчивый интерес к изучению данного вида течений сформировался после появления работы [2], в которых идеи расчета были перенесены на стратифицированные жидкости и дополнены лабораторными опытами [12], показавшими наблюдаемость тонкого процесса. В дальнейшем течения, индуцированные диффузией на непроницаемых поверхностях, были прослежены более детально применительно и к атмосфере, где они достигают больших значений на спадающих ледниках, и к гидросфере [3–6]. Такие течения формируют тонкую структуру среды и служат пропульсивными механизмами, обеспечивающими самодвижение тел (например, клиновидного тела нейтральной плавучести в лабораторном бассейне [3]), непроницаемых

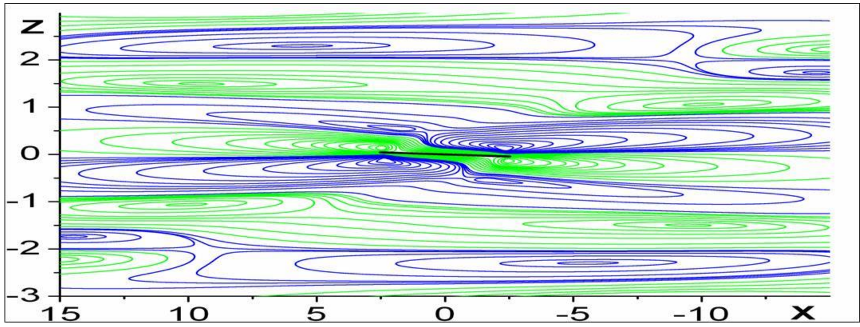
частиц в устойчиво стратифицированной по солености морской среде и в пресноводных водоемах с температурной стратификацией.

4.1. Структура стратифицированного течения на неподвижной пластине

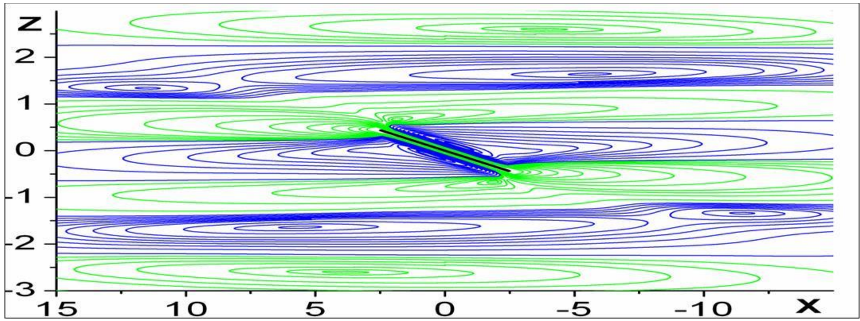
Картина течения, индуцированного диффузией около горизонтальной пластины, моделирующей центральное сечение непроницаемого препятствия произвольной формы, состоит из восьмиуровневой последовательности симметрично расположенных горизонтальных вихревых ячеек (рис. 1, *а*) [5].

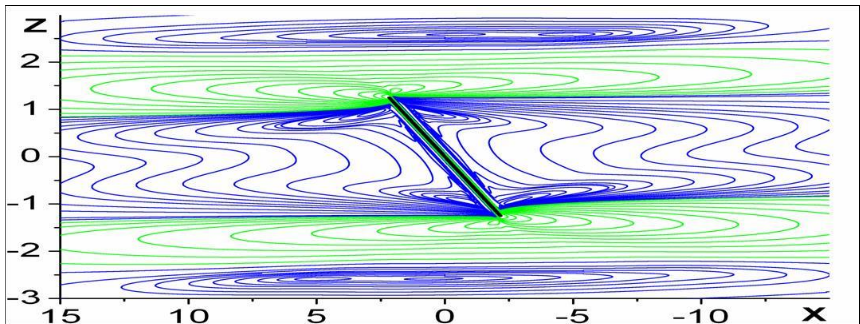
Скорость и завихренность течения резко убывает с удалением от поверхности пластины. Даже малое отклонение пластины от горизонтального положения приводит к нарушению симметрии течения и формированию новых циркуляционных систем, включая восходящее и нисходящее струйные течения вдоль, соответственно, верхней и нижней сторон пластины и систему компенсационных циркуляционных ячеек (рис. 1, *б*). С дальнейшим увеличением угла наклона линии тока непосредственно вблизи пластины располагаются параллельно ее поверхности (рис. 1, *в*). Однородность картины линий тока свидетельствует о неизменности профиля скорости вдоль большей части длины пластины за исключением узких переходных областей в окрестности ее кромок. При дальнейшем увеличении угла наклона пластины к горизонту происходит зарождение новых сложных вихревых систем вблизи внутренних краев циркуляционных ячеек, непосредственно примыкающих к главным струйным течениям (рис. 1, *г*). Внешние контуры примыкающих ячеек имеют периодичную структуру с длиной волны порядка $16\,\delta_N$, что наиболее отчетливо заметно при больших углах наклона пластины к горизонту, когда вертикальный масштаб препятствия максимален.





а) *б)*





в) г)

Рис.1. Структура течения, индуцированного диффузией на наклонной пластине:

а) – $\varphi = 0^\circ$, б) – $\varphi = 1^\circ$, в) – $\varphi = 10^\circ$, г) – $\varphi = 30^\circ$, $N = 1.26$ с$^{-1}$, $L = 5$ см,

$\tau = 50$.

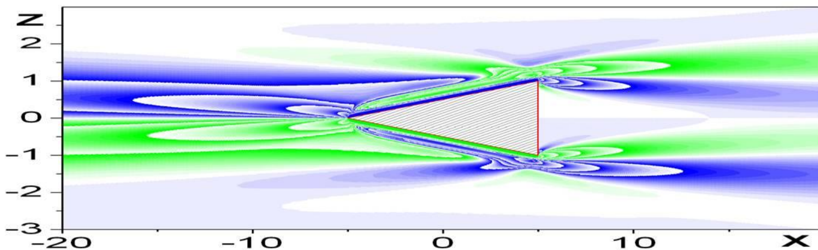
Во всех случаях, и при горизонтальном, и наклонном положении пластины, циркуляция в соприкасающихся ячейках над и под пластиной, как и в соседних ячейках на одном горизонте, имеет противоположные знаки, т.е. скорость на границах соседних ячеек однонаправлена. Такие пространственно-периодические течения можно отнести к классу диссипативно-гравитационных волн, в которых скорость на границах достигает максимального значения, в отличие от ячеек многокомпонентной конвекции, где скорости разнонаправлены и границы ячеек отмечены слоем покоящейся жидкости.

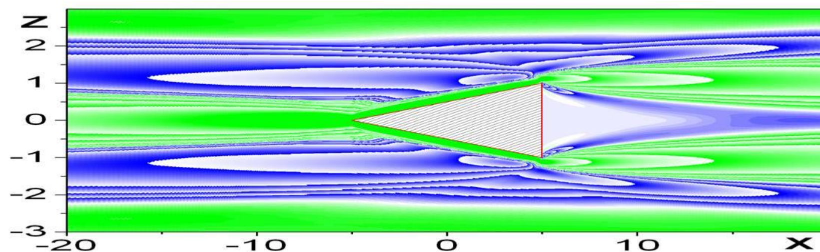
Расчеты полных сил и моментов, действующих на наклонную пластину в покоящейся непрерывно стратифицированной жидкости, показывают, что формирующиеся струйные течения жидкости создают момент сил, стремящийся повернуть наклонную пластину в устойчивое горизонтальное положение [5].

4.2. Индуцированное диффузией течение на клине

Непроницаемое клиновидное препятствие, погруженное в толщу непрерывно стратифицированной жидкости, блокирует фоновый диффузионный перенос стратифицирующей компоненты, в результате чего вблизи его боковых поверхностей формируется тонкий слой дефицита солености на верхней грани и избытка солености на нижней (рис. 2, *а*). Это приводит к тому, что жидкость начинает оттекать от поверхности клина в слоях почти однородной толщины, формируя восходящие струйные течения вдоль наклонных граней и систему примыкающих противотечений (рис. 2, *б*). Отрываясь от препятствия у его основания, главные струи формируют сложную систему компенсационных циркуляционных ячеек, интенсивность которых резко убывает с расстоянием от препятствия. Окружающая жидкость подтекает к вершине клина вдоль центральной горизонтальной струи, которая утончается к препятствию [4].

В донной области жидкость заблокирована и чрезвычайно медленно подтекает к клину по всей высоте основания, при этом скорость возрастает только при подходе к вершинам клина. В полях всех переменных выражены элементы структуры, дополняющие препятствие до симметричного тела – от треугольника к призме.





а) *б)*

Рис. 2. Картина индуцированного диффузией течения на симметричном клине: поля а) – возмущения солености, б) – горизонтальной компоненты скорости,

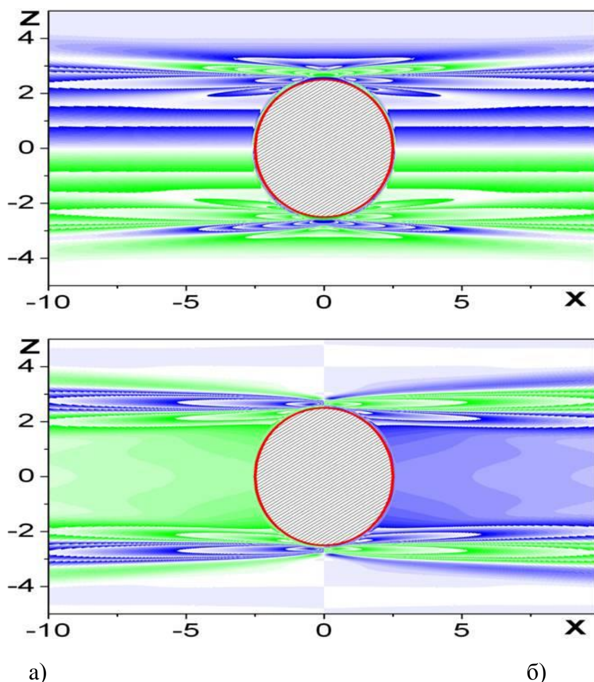
$$L_x = 10\,\text{см}, T_b = 7.5\,\text{с},\ N = 0.84\,\text{с}^{-1},\ \kappa_S = 1.4\cdot 10^{-5}\,\text{см}^2/\text{с}.$$

В картине поля возмущений давления область дефицита располагается в толстом слое перед телом и в тонкой прослойке, примыкающей к боковым сторонам клина. Разность давлений – подпор у основания и дефицит перед клином, а также в тонком слое вдоль его боковых сторон и создает интегральную силу, толкающую горизонтальный клин в направлении его вершины – пропульсивную силу самодвижения свободного тела нейтральной плавучести [3, 4]. Основным является дефицит давления, достаточный для описания наблюдаемого перемещения тела со скоростью порядка сантиметра в час в условиях лабораторного эксперимента.

4.3. Течение, индуцированное диффузией на горизонтальном цилиндре

Картина течения, индуцированного диффузией, около непроницаемого цилиндра качественно отличается от структуры течений около наклонной пластины и горизонтального клина. Это обусловлено зависимостью величины нормального градиента возмущения солености от значения угла между вертикальной осью и вектором нормали к поверхности препятствия. Толщина формирующихся слоев дефицита и переизбытка солености ниже и выше препятствия оказывается существенно неоднородной вдоль поверхности цилиндра (рис. 3, *а*), в отличие от рассмотренных ниже случаев наклонной пластины и клина, где существенные изменения геометрии основных структурных компонент течения наблюдаются лишь в окрестностях острых кромок препятствий. Интенсивность склоновых струйных течений, формирующихся на поверхности цилиндра достигают максимальных значений в сечении, образующем угол 60° с горизонтальной осью (рис. 3, *б*), в отличие от результатов расчета параметров главных струй на наклонной

пластине, которые показывают пропорциональное увеличение их скорости и толщины с уменьшением угла наклона пластины к горизонту [5].



а) б)

Рис.3. Картина индуцированного диффузией течения на горизонтальном цилиндре: поля а) – возмущения солености, б) – горизонтальной компоненты скорости,

$$D = 5\,\text{см},\ \ T_b = 7.5\,\text{с},\ \ N = 0.84\,\text{с}^{-1},\ \ \kappa_S = 1.4 \cdot 10^{-5}\ \text{см}^2/\text{с}.$$

Численное решение описывает структуру основных конвективных ячеек, включающие тонкие струи вдоль поверхности и растекающиеся факелы от области конвергенции течения над верхним и нижним полюсами цилиндра, постепенно возвращающие частицы жидкости на горизонты нейтральной плавучести [6]. Это характерным образом проявлено в картине поля возмущения давления, где преобладают области с отрицательными значениями величины, что обусловлено компенсационным подтеканием окружающей жидкости к поверхности цилиндра, за исключением областей растекания струй вблизи полюсов препятствия.

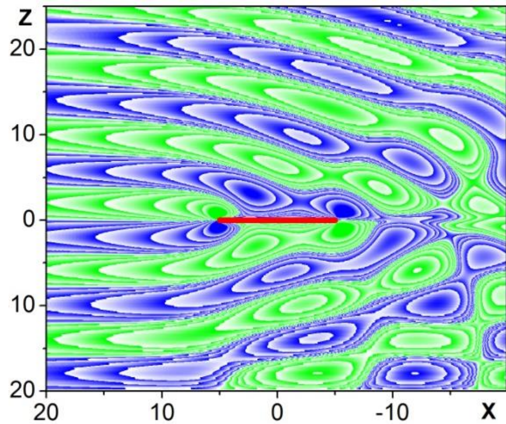
Расчеты динамических характеристик цилиндра показывают, что суммарные силы и моменты, действующие на симметричные части поверхности препятствия, взаимно уравновешиваются, что подтверждает отсутствие

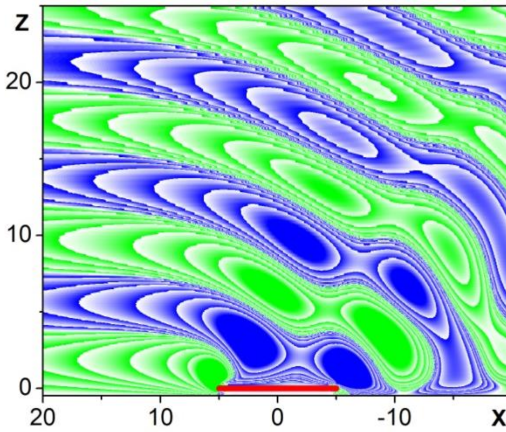
самопроизвольных движений погруженного в покоящуюся непрерывно стратифицированную жидкость цилиндра нейтральной плавучести.

4.4. Экранный эффект для полосы в стратифицированной жидкости

На сегодняшний день проблема управления аэродинамическим качеством крыловых профилей или иных аэродинамических компоновок является насущной ввиду всевозрастающей необходимости экономичного использования энергоресурсов. Одним из решений данной задачи является создание «экранного эффекта», который возникает при движении летательных аппаратов вблизи Земли или поверхности воды. При полете экраноплана на небольшом расстоянии от поверхности возникает дополнительная подъёмная сила, наиболее полный эффект которой проявляется при высоте движения не превосходящей длину хорды крыла. Набегающий поток, отражаясь, успевает дойти до поверхности и вернуться обратно, то есть крыло уплотняет под собой набегающий поток, превращая его в динамическую воздушную подушку [11].

Численное моделирование обтекания горизонтальной пластины конечной толщины – простейшей модели препятствия вблизи подстилающей твердой стенки выполнено на основе фундаментальной системы уравнений механики неоднородных жидкостей с применением построенного решателя пакета OpenFOAM. Рассчитанная картина стратифицированного течения оказывается существенно асимметричной относительно плоскости препятствия даже при его расположении на сравнительно большой высоте от подстилающей поверхности (рис. 4, *а*). Структура области течения, расположенной ниже плоскости полосы, существенно зависит от соотношения величин $h/\lambda=h/UT_b$ – высоты расположения препятствия и длины внутренней присоединенной волны. Отраженные от жесткой подстилающей поверхности внутренние волны искажают не только область течения под полосой, но и оказывают заметное влияние на волновую картину в верхней полуплоскости.

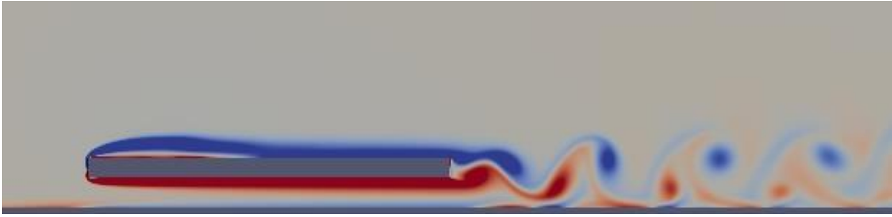




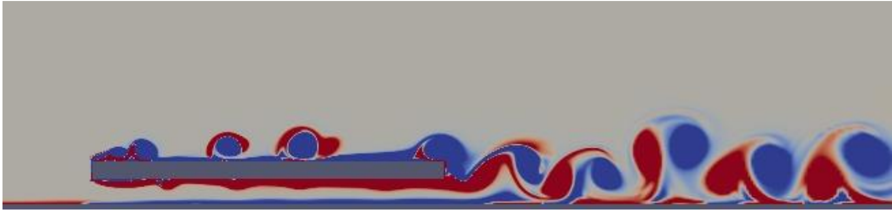
а) *б)*

Рис. 4. Картины дальнего поля установившегося течения при обтекании горизонтальной полосы потоком непрерывно стратифицированной жидкости:

U = 1 см/с, N = 0.83 $с^{-1}$, L = 10 см, τ = 20, *а)* – h = 20 см, *б)* – h = 0.5 см.



а)



б)

Рис. 5. Картины обтекания горизонтальной полосы потоком стратифицированной жидкости:

а) – U = 4 см/с, *б)* – U = 24 см/с, N = 0.83 $с^{-1}$, L = 10 см, h = 1 см, τ = 20.

В предельном случае малого расстояния между пластиной и донной поверхностью (рис. 4, *б*) рассчитанные картины течения согласуются с результатами численной визуализации точного решения задачи генерации внутренних волн полосой, движущейся на подстилающей плоскости [8]. Сопоставления рассчитанных картин обтекания горизонтальной полосы потоком стратифицированной жидкости с результатами теневой визуализации в лабораторном бассейне демонстрируют их достаточно хорошую согласованность при малых и умеренных скоростях движения.

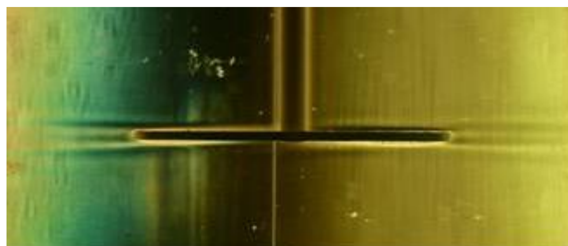
С увеличением скорости набегающего потока за обтекаемой пластиной начинает формироваться спутный след, структура которого существенно зависит от расстояния до подстилающей плоскости (рис. 5, *а*). Наличие ограничивающей поверхности нарушает регулярность следа вниз по потоку, наблюдаемую в случае обтекания пластины в свободном пространстве. Дальнейший рост скорости характеризуется интенсификацией вихревого следа и появление дополнительных вихревых структур, генерируемых передними острыми кромками пластины с частотой пропорциональной скорости обтекания пластины (рис. 5, *б*). Формирование вихревых структур вдоль нижней стороны пластины подавляется подстилающей поверхностью, что оказывает стабилизирующее влияние на течение в целом и характер распределения динамических характеристик вдоль сторон препятствия.

5. Тестирование результатов расчета на лабораторных данных

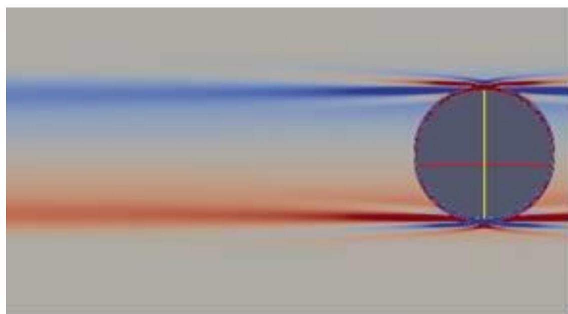
Рассчитанные поля возмущений градиента плотности $\nabla\rho$ для течений, индуцированных диффузией на горизонтальной и наклонной пластине, горизонтальном диске и цилиндре, в которых проявляются и крупномасштабные компоненты, размер которых задается длиной пластины, и тонкие прослойки с масштабами $\delta_N^{\nu} = \sqrt{\nu/N}$ и $\delta_N^{\kappa_S} = \sqrt{\kappa_S/N}$, на больших временах согласуются с картинами визуализации («цветной теневой метод» с горизонтальной щелью и решеткой) распределения градиента коэффициента преломления около пластины в лабораторном бассейне (плотность и коэффициент преломления водных растворов поваренной соли связаны линейным соотношением) [12].

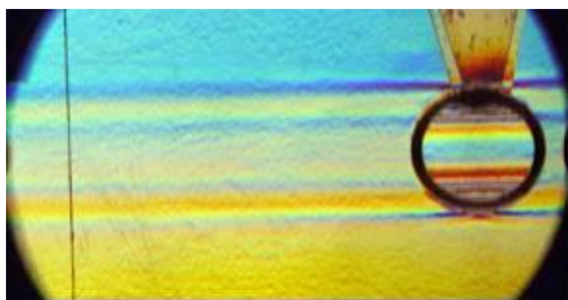
В рассчитанной и теневой картинах (рис. 6) выделяются протяженные горизонтальные полосчатые структуры, примыкающие непосредственно к экстремальным точкам препятствий, разнесенным по вертикали, – острым кромкам диска и полюсам цилиндра. Длина полосок растет с повышением чувствительности метода регистрации: структура течения около кромок наклонной пластины выражена более отчетливо при использовании цветного теневого метода с горизонтальным положением осветительной щели и визуализирующей решетки. Течения, индицированные диффузией, существуют в стратифицированной среде при произвольной геометрии препятствия и его ориентации в пространстве, и отсутствуют в однородной жидкости.

С началом движения пластины структура стратифицированного течения меняется кардинально: возникают опережающие возмущения, присоединенные внутренние волны и спутный след. Горизонтальные прослойки, существующие в покоящейся стратифицированной среде на неподвижном препятствии, трансформируются в квазистационарные высокоградиентные области, разделяющие разные типы возмущений, в частности, спутный след и внутренние волны. Рассчитанные поля течения, возникающего при движении горизонтальной пластины в толще непрерывно стратифицированной жидкости, сравниваются с картинами теневой визуализации, получаемыми в лабораторных опытах с применением метода “вертикальная щель – вертикальный нож Фуко” (Рис.7). Численное решение содержит все структурные компоненты поля возмущений и достаточно хорошо согласуется с наблюдениями позади пластины [8].



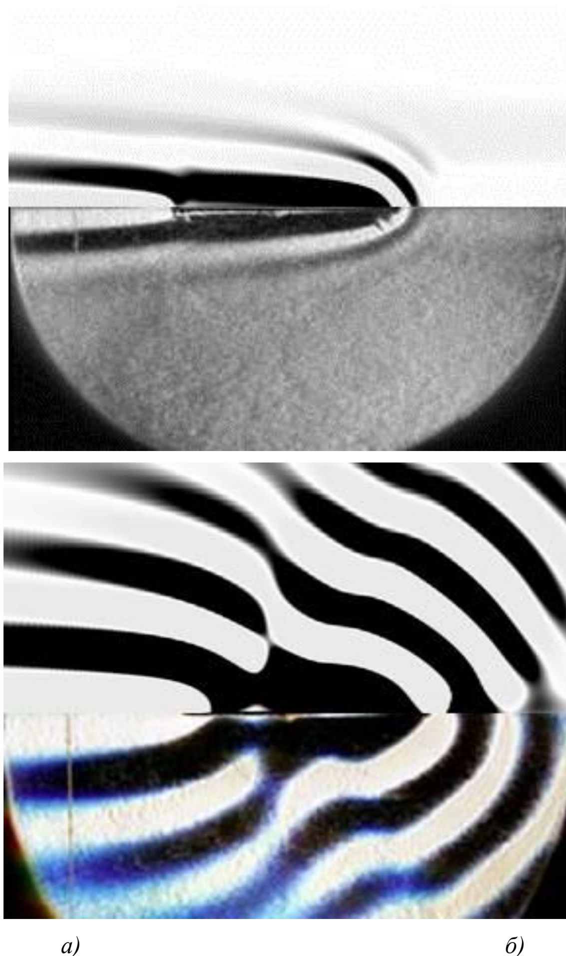
а) *б)*





в) *г)*

Рис. 6. Рассчитанные (а,в) и теневые (б,г) картины стратифицированных течений около непроницаемых препятствий: а,б) – горизонтального диска и в,г) – кругового цилиндра.



а) *б)*

Рис. 7. Рассчитанные поля вертикальной компоненты скорости (верхняя половина изображений) над теневой картиной течения, возникающего при равномерном движении горизонтальной пластины в непрерывно стратифицированной жидкости: *а)* – U=0.1 см/с, T_b=7.6 с, L_x=7.5 см ; *б)* – U=0.32 см/с, T_b=7.6 с, L_x=7.5 см.

5. Заключение

Разработан собственный решатель открытого пакета OpenFOAM, позволяющий рассчитывать поля всех физических величин стратифицированных течений на основе фундаментальной системы уравнений механики неоднородных жидкостей. Расчеты практически ценных задач проведены на базе технологической платформы UniHUB, предоставляющей все необходимые инструментарии для численного моделирования с прямым доступом на вычислительный сегмент кластера МСЦ РАН.

Проведенные расчеты позволили выявить сложную ячеистую структуру даже в самых медленных течениях, индуцированных диффузией на непроницаемых препятствиях различной геометрической формы. Эти структуры только усложняются и утончаются с увеличением характерных скоростей, трансформируясь в квазистационарные высокоградиентные области, разделяющие разные типы возмущений.

Тестирование результатов расчетов на лабораторных данных показало хорошее согласие рассчитанных и теневых картин течений стратифицированной жидкости около неподвижных и движущихся препятствий. Численные расчеты адекватно отражают основные элементы структуры стратифицированных течений: высокоградиентные прослойки, опережающие возмущения, внутренние присоединенные волны и спутный след.

Благодарности

Автор выражает глубокую признательность разработчикам вычислительной платформы UniHUB и Web-лаборатории UniCFD за предоставленную возможность удобно и продуктивно работать в виртуальной вычислительной среде, а также академику РАН В.П. Иванникову за ценные советы по совершенствованию методик высокопроизводительных вычислений и профессору Ю.Д. Чашечкину за многолетнюю поддержку усилий по разработке высокоразрешающих численных моделей течений неоднородных жидкостей.

Список литературы

[1]. Прандтль Л. Гидроаэромеханика. М.: ИЛ. 1949. 520 с.

[2]. Phillips O.M. On flows induced by diffusion in a stably stratified fluid // Deep-Sea Res. 1970. V. 17. P. 435–443.

[3]. Allshouse M.R., Barad M.F., Peacock T. Propulsion generated by diffusion-driven flow // Nature Physics. 2010. V.6. P. 516-519.

[4]. Загуменный Я.В., Чашечкин Ю.Д. Индуцированное диффузией течение на клине // Доклады НАН Украины. 2013. № 3. С. 31–39.

[5]. Чашечкин Ю.Д., Загуменный Я.В. Структура течения, индуцированного диффузией на наклонной пластине // Доклады РАН. 2012. Т. 444, № 2. С. 165–171.

[6]. Байдулов В.Г., Матюшин П.В., Чашечкин Ю.Д. Эволюция течения, индуцированного диффузией на сфере, погруженной в непрерывно стратифицированную жидкость // Механика жидкости и газа. 2007. №2. С. 130–143.
[7]. Chashechkin Yu.D., Mitkin V.V. A visual study on flow pattern around the strip moving uniformly in a continuously stratified fluid // J. Visualization. 2004. V.7, №2. P.127–134.
[8]. Чашечкин Ю.Д., Бардаков Р.Н., Загуменный Я.В. Расчет и визуализация тонкой структуры полей двумерных присоединенных внутренних волн // Морской гидрофизический журнал. 2010. № 6. С. 3–15.
[9]. Чашечкин Ю.Д. Иерархия моделей классической механики неоднородных жидкостей // Морской гидрофизический журнал. 2010. №5. С.3–10.
[10]. Загуменный Я.В. Использование технологической платформы UniHUB в расчетах тонкой структуры стратифицированных течений на базе открытых пакетов // III Международная конференция «Облачные вычисления: образование, исследования, разработки», 6–7 декабря 2012, Москва, http://www/unicluster.ru/events/139-conference-cloud-computing-2012.html.
[11]. Zagumennyi Ia.V., Bardakov R.N. Ground effect in hydrodynamics of a strip in a stratified fluid // IUTAM Symposium 12–3 = GA.10-08 “Waves in fluids: effects of non-linearity, rotation, stratification and dissipation”, June 18-22, 2012, Moscow, Russia, P. 172–175.
[12]. Chashechkin Yu.D. Schlieren Visualization of a Stratified Flow around a Cylinder // J. Visualization. 1999. V. 1, N 4. P. 345–354.

Calculations of continuously stratified fluid flows using open source computational packages based on the technological platform UniHUB

Ia.V. Zagumennyi, Yu.D. Chashechkin
e-mails: zagumennyi@gmail.com, chakin@ipmnet.ru
Institute of Hydromechanics, National Academy of Sciences of Ukraine,
IHM NASU, 8/4, Zheliabova street, 03680, Kiev, Ukraine
Institute for Problems in Mechanics of the Russian Academy of Sciences,
IPMech RAS, 101/1, prospect Vernadskogo, 119526, Moscow, Russia

Abstract. The paper presents the authors' experience in usage of the technological platform UniHUB for numerical simulation and computations of continuously stratified fluid flows based on the open source computational packages OpenFOAM, Salome and ParaView. Special attention is paid to the problems of high-resolution computational grids construction, complex boundary conditions setting using the standard and extended OpenFOAM utilities, own solvers development, numerical data processing and visualization and running program codes in parallel on the JSCC RAS Cluster, as well. Some physical results are demonstrated on the stratified flows structure and dynamics around impermeable sloping plate, symmetrical wedge, horizontal disc and circular cylinder. The obtained numerical results have direct application to the natural systems since the Earth's atmosphere and hydrosphere are mostly stably stratified due to non-uniformity of distributions in space and time of dissolved or suspended matters, gas bubbles, temperature, medium compressibility and effects of external forces. The numerical study of diffusion-induced flows on an impermeable obstacle reveals a system of jet-like flows formed along its sloping boundaries and a complicated structure of circulation cells attached to the surface of the obstacle. The most intensive structures are clearly registered experimentally by Schlieren techniques in form of horizontally extended high gradient interfaces attached to extreme points of an obstacle, i.e. sharp edges of a plate, poles of a cylinder, vertices of a wedge, etc. With increase of typical velocities these structures do not disappear but are transformed into a complicated system of thin interfaces separating different kinds of disturbances, e.g. internal waves and a vortex sheet. The structural elements of extremely slow flows of non-homogeneous fluids form a flow fine structure in rapidly changing environments. The analytical, numerical and laboratory data are compared with each other, conditions of their agreement being discussed together with possibility of their application to the natural systems.

Keywords: direct numerical simulation; open source computational packages; stratified flows; flow fine structure; internal waves; Schlieren visualization.

References

[1]. Prandtl L. The Essentials of Fluid Dynamics. London: Blackie and Son, 1952.
[2]. Phillips O.M. On Flows Induced by Diffusion in a Stably Stratified Fluid. Deep-Sea Research, 1970, vol. 17, pp. 435–443.

[3]. Allshouse M.R., Barad M.F., Peacock T. Propulsion Generated by Diffusion-Driven Flow. Nature Physics, 2010, vol. 6, pp. 516–519.
[4]. Zagumennyi Ia.V., Chashechkin Yu.D. Indutsirovannoe diffuziey techenie na kline [Diffusion Induced Flow on a Wedge]. Doklady NAN Ukrainy [Reports NAS of Ukraine], 2013, no. 3, pp. 31–39 (in Russian).
[5]. Chashechkin Yu.D, Zagumennyi Ya.V. Structure of Diffusion-Induced Flow on an Inclined Plate. Doklady Physics, 2012, vol. 57, no. 5, pp. 210–216.
[6]. Baydulov V.G., Matyushin P.V., Chashechkin Yu.D. Structure of a Diffusion-Induced Flow Near a Sphere in a Continuously Stratified Fluid. Doklady Physics, 2005, vol. 50, no. 4, pp. 195–199.
[7]. Chashechkin Yu.D., Mitkin V.V. A Visual Study on Flow Pattern around the Strip Moving Uniformly in a Continuously Stratified Fluid. J. Visualization, 2004, vol. 7, no. 2, pp. 127–134.
[8]. Chashechkin Yu.D., Bardakov R.N., Zagumennyi Ia.V. Numerical Analysis and Visualization of Fine Structures of the Fields of Two-Dimensional Internal Waves. Physical Oceanography, 2011, vol. 20, no. 6, pp. 397–409.
[9]. Chashechkin Yu.D. Hierarchy of the Models of Classical Mechanics of Inhomogeneous Fluids. Physical Oceanography, 2011, vol. 20, no. 5, pp. 317–324.
[10]. Zagumennyi Ia.V. Ispol'zovanie tekhnologicheskoi platformy UniHUB v raschyetakh tonkoi struktury stratifitsirovannykh techeniy na baze otkrytykh paketov [Usage of the Technological Platform UniHUB in Calculations of Continuously Stratified Fluid Flows Based on the Open Source Computational Packages]. Trudy III Mezhdunarodnoi Konferenzii "Oblachnye vychisleniya: obrazovanie, issledovaniya, razrabotki" [Proc. 3d International Conference on Cloud Computing: Education, Research, Development], 2012. p. 10 (in Russian). http://www/unicluster.ru/events/139-conference-cloud-computing-2012.html
[11]. Zagumennyi Ia.V., Bardakov R.N. Ground Effect in Hydrodynamics of a Strip in a Stratified Fluid. Proc. IUTAM Symposium 12–3 = GA.10-08 on Waves in Fluids: Effects of Non-linearity, Rotation, Stratification and Dissipation, 2012. pp. 172–175.
[12]. Chashechkin Yu.D. Schlieren Visualization of a Stratified Flow around a Cylinder. J. Visualization, 1999, vol. 1, no. 4, pp. 345–354.

Прямая передача данных между ПЛИС Virtex-7 по шине PCI Express

Ю. А. Румянцев, ООО НПО «Роста»,
rumyantsev@rosta.ru

Аннотация. В данной статье рассматривается передача данных по шине PCI Express с одновременным участием нескольких ПЛИС. В компьютерной системе, к PCI Express шине которой подключено несколько (в нашем случае 8) оконечных устройств (PCIe endpoints) ПЛИС запускается одновременно несколько транзакций передачи данных двух типов: А) DMA передача между ОЗУ и ПЛИС (чтение/запись) и Б) прямая передача данных между двумя ПЛИС (запись). Используя соединение PCI Express x4 Gen 2.0 при обращении в память была получена скорость записи 1451 МБ/с (90% от максимальной). Скорость записи данных между ПЛИС была равна 1603 МБ/с (99 % от максимальной) при длине пакетов 128 байт и 1740 МБ/с (99% от максимальной) при длине пакета 256 байт. Латентность передачи данных между ПЛИС зависит от количества промежуточных коммутаторов, и была равна 0,7 мкс для одного коммутатора и 1 мкс для трех. Также показано, что при одновременных передачах через общий канал скорость отдельных передач не уменьшается до тех пор, пока суммарная скорость передачи не превышает пропускную способность общего канала; затем канал используется на 100%, а его пропускная способность делится поровну между устройствами.

Ключевые слова: ПЛИС; FPGA; PCI Express;

1. Введение

PCI Express де факто стало стандартом передачи данных между CPU, системной памятью и аппаратными ускорителями (GPU, ПЛИС) в задачах High Performance Computing (HPC). Во-первых, шина PCI Express обладает небольшой латентностью, во-вторых, имеет высокую скорость передачи данных (около 7 ГБайт/с при соединении PCI Express x8 Gen 3.0). Наконец, шине PCIe присуща хорошая масштабируемость: обычно на материнских платах нет недостатка в разъемах шины PCI Express, к которой можно

подключить несколько плат ускорителей GPU или ПЛИС. Также в последнее время появились технические решения, позволяющие расширить шину PCI Express через кабельные соединения и подключать дополнительные периферийные устройства вне корпуса компьютера (1).

В современных HPC системах недостаточно иметь единственный аппаратный ускоритель. Уже стало привычным видеть две платы GPU на локальной PCI Express шине вычислительного узла. Для обеспечения обмена данными напрямую между GPU была разработана технология GPUDirect (2). Используя эту технологию, можно организовать обмен данными между устройствами GPU по шине PCI Express напрямую без использования оперативной памяти в качестве буфера, что позволяет существенно снизить накладные расходы на передачу данных.

Другие примеры множества ускорителей на шине PCI Express включают в себя системы, в которых одновременно работают и GPU и ПЛИС. В первом примере команда исследователей из Австралии собрали персональный компьютер из материнской платы Intel, процессора Core i7, платы GPU nVidia Tesla C2070 и платы Altera DE-530 с установленным кристаллом ПЛИС Stratix-IV (3). Они назвали его «Химера» в честь мифического чудовища Древней Греции Химеры, имеющий 3 головы (козел, змея, лев) на одном теле. Они успешно решили несколько задач (интегрирование методом Монте-Карло, поиск шаблона в 2D массиве) и работают над применением этой системы для анализа непрерывных гравитационных волн. Ключевой особенностью их проекта было то, что GPU и ПЛИС работали одновременно над одной задачей, и данные передавались от GPU к ПЛИС по шине PCI Express. Однако следует отметить, что этот обмен шел под управлением центрального процессора и через буфер в оперативной памяти.

Другая команда исследователей из Брюсселя собрала гибридный компьютер с платами GPU nVidia Tesla C2050 и ПЛИС Pico Computing EX-500 (4). Последняя плата может включать в себя от 1 до 6 кристаллов ПЛИС Xilinx Virtex6, каждая со своим собственным PCI Express интерфейсом к хосту. Подробности проекта пока неясны, сообщение доступно только в виде препринта.

Наконец команда разработчиков из компании Microsoft исследовала передачу данных напрямую между GPU и ПЛИС по шине PCI Express (5). В их системе были установлены платы GPU nVidia GeForce GTX 580 и ПЛИС Xilinx ML605 с 1 кристаллом Virtex6. Разработчики нашли способ инициировать прямую передачу данных между GPU и ПЛИС, используя CUDA API, технологию GPUDirect и доработки Linux драйвера устройства ПЛИС. Это позволило увеличить скорость и снизить латентность передачи данных по сравнению с подходом, использующим оперативную память в качестве промежуточного буфера. В данном случае GPU было ведущим устройством, а ПЛИС ведомым.

Шина PCI Express может быть использована и для передачи данных напрямую между ПЛИС. Компания Xilinx демонстрировала эту возможность (6). Инженеры Xilinx соединяли два кристалла ПЛИС напрямую по шине PCI Express без использования коммутаторов и вообще без компьютера с центральным процессором. Один кристалл самостоятельно конфигурировал свой PCI Express интерфейс, устанавливал связь со вторым и конфигурировал его. После чего было возможно передавать данные в обе стороны между двумя кристаллами ПЛИС. Этот подход демонстрирует принципиальную возможность передачи данных между ПЛИС по шине PCI Express, однако не может быть использован в случае, когда к компьютеру с центральным процессором подключены несколько ПЛИС через PCI Express коммутаторы.

В данной статье описывается передача данных в системе, содержащей множество ПЛИС на PCI Express шине. Обсуждаются вопросы одновременной передачи данных между системной памятью и несколькими ПЛИС, а также одновременной передачи данных между несколькими ПЛИС напрямую друг с другом. По нашим сведениям данное сообщение является первым в части описания передачи данных между ПЛИС через PCI Express шину компьютера.

2. Описание системы

Эксперименты проводились в следующей системе. В материнскую плату с процессором Intel Core i7 в разъем PCI Express 2.0 x8 устанавливался адаптер RHA-25 производства фирмы Роста (1), расширяющий PCI Express шину через кабельные соединения. На адаптере RHA-25 установлен PCI Express коммутатор (PLX Technology), три порта которого используются для внешних соединений: один ножевой upstream порт x8 Gen 3.0 и два кабельных downstream порта x4 Gen 3.0. К этой системе через 2 кабельных соединения PCI Express x4 Gen 3.0 подключался вычислительный блок RB-8V7 (1). Блок RB-8V7 имеет симметричную архитектуру и конструктивно состоит из двух плат RC-47. Каждая плата RC-47 имеет коммутатор PCI Express фирмы PLX с одним кабельным upstream портом и четырьмя downstream портами, каждый из которых соединен со своим ПЛИС Xilinx Virtex-7 (XC7V585T). Таким образом, в нашей системе к хосту по шине PCI Express с помощью одного адаптера RHA-25 подключалось 8 ПЛИС Virtex-7 (V7). Все ПЛИС соединялись с коммутатором PLX по интерфейсу PCI Express x4 Gen 2.0.

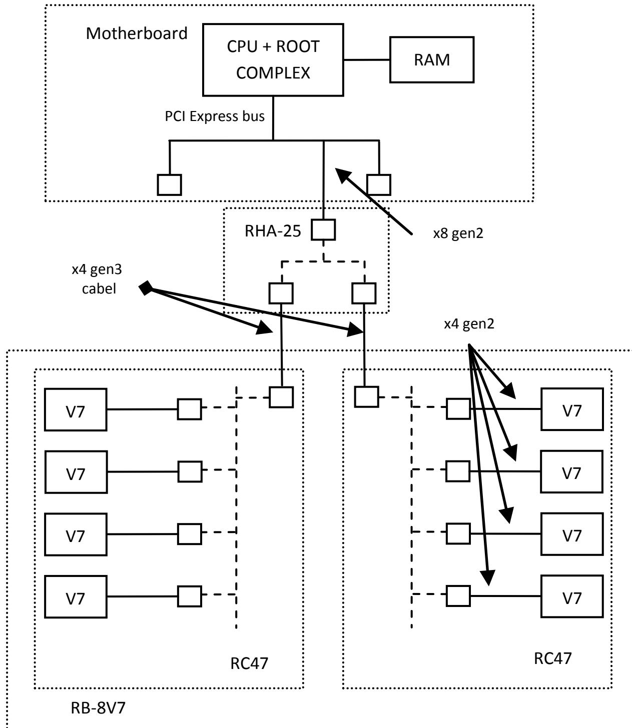


Рис. 1 Аппаратное обеспечение. Блок RB-8V7 подключен через кабельные PCIe соединения и адаптер RHA-25 к хост компьютеру

Внутри ПЛИС была реализована следующая схема (рис. 2). Проект использует PCI Express IP ядро фирмы Xilinx (7). Блок Rosta DMA Engine определяет функционал устройства на шине PCI Express. ПЛИС может выступать как в роли ведущего, так и ведомого устройства. Как ведомое устройство ПЛИС предоставляет центральному процессору доступ по чтению и записи к своим регистрам, а также может принимать большие пакеты данных от других устройств на шине (например, от других ПЛИС) с сохранением принятых данных в очереди EP_RX_FIFO. Как ведущее устройство ПЛИС способна обращаться в оперативную память компьютера в режиме DMA (чтение/запись). При этом при записи в память данные будут считываться из TX_FIFO, а при чтении из памяти – записываться в RX_FIFO. Также устройство способно генерировать транзакции записи по произвольному адресу на шине (например, для передачи данных в другие ПЛИС), в этом случае данные для передачи считываются из очереди EP_TX_FIFO. Организацией приема поступающих пакетов занимается автомат RX_STATE_MACHINE, за передачу пакетов отвечает автомат TX_STATE_MACHINE. Прием и передача пакетов могут идти одновременно.

Блок TX_ARBITER определяет, какой пакет генерировать для передачи следующим: абсолютный приоритет отдается генерации ответов в процессе чтения регистров центральным процессором, остальные пакеты (запросы на чтение/запись оперативной памяти или запросы на запись по произвольному адресу) планируются с равным приоритетом (round-robin).

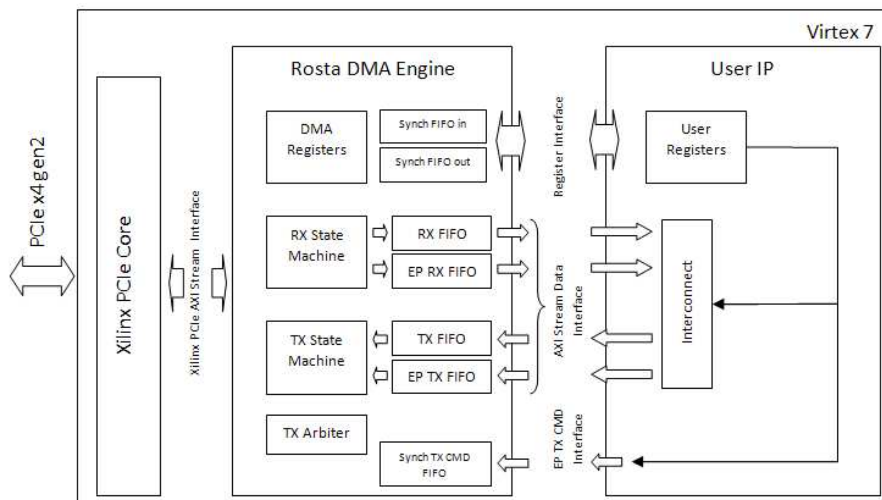


Рис. 2 Блок-схема проекта ПЛИС

Программированием DMA передачи данных между ПЛИС и ОЗУ занимается центральный процессор (путем записи в блок DMA_REGISTERS), а для управления процессом записи данных по произвольному адресу в другую ПЛИС есть внутренний аппаратный интерфейс EP TX CMD. Для доступа к регистрам пользователя в блоке User IP существует интерфейс Register Interface, а для блочной передачи данных между пространством PCI Express и пользовательской схемой есть четыре AXI Stream интерфейса, соединенных с очередями RX_FIFO, TX_FIFO, EP_RX_FIFO, EP_TX_FIFO. Наконец в сторону PCI Express ядра Xilinx блок Rosta DMA Engine поддерживает Xilinx PCIe AXI Stream Interface, имеющий ширину 64 бита. Блок работает на двух тактовых частотах: на частоте работы PCIe ядра Xilinx (250 МГц) – левая часть схемы, и на произвольной пользовательской частоте – правая часть схемы. Развязка по частотам происходит через очереди FIFO. Но во всех описываемых далее экспериментах пользовательская частота была равна частоте, на которой работало ядро PCIe (250 МГц).

Пользовательский блок User IP определяет поведение устройства на уровне приложения. В данной работе использовались несколько разных схем для разных назначений. Во-первых, использовалась схема для проверки корректности передачи данных между ПЛИС и ОЗУ. В этом случае в блоке

Interconnect выход RX_FIFO просто замыкался на вход TX_FIFO. Это позволяло записывать в оперативную память абсолютно те же данные, что и были считаны из нее. Программа на центральном процессоре (в дальнейшем просто хост) записывала данные в ПЛИС, считывала их, сравнивала и убеждалась в правильности сравнения данных.

Во-вторых, использовалась схема для измерения максимальной скорости передачи данных между ПЛИС и ОЗУ в обоих направлениях. Для этого из RX_FIFO данные, постоянно считывались, т.е. очередь всегда была пустая, и не было задержек при приеме данных из-за ее переполнения, а в TX_FIFO шла постоянная запись данных, т.е. не возникало задержек при передаче из-за недостатка данных в очереди.

В-третьих, была разработана схема для проверки корректности прямой передачи данных между ПЛИС. В блоке Interconnect была реализована схема коммутации выходов RX_FIFO и EP_RX_FIFO и входов TX_FIFO и EP_TX_FIFO. В первом случае выход RX_FIFO замыкался на вход TX_FIFO, а выход EP_RX_FIFO – на вход EP_TX_FIFO. Во втором случае выход RX_FIFO замыкался на вход EP_TX_FIFO, а выход EP_RX_FIFO – на вход TX_FIFO. Управлялась эта схема коммутации битом из одного из пользовательских регистров. В блок User Registers были добавлены регистры для управления интерфейсом EP TX CMD. В данном случае сам хост управлял передачей данных между ПЛИС, но вообще интерфейс EP TX CMD разрабатывался таким образом, чтобы сама схема ПЛИС могла инициировать передачу данных.

В четвертых, для измерения максимальной скорости передачи данных между ПЛИС была разработана специальная схема, которая при передаче данных постоянно записывала данные в EP_TX_FIFO, а при приеме постоянно вычитывала данные из EP_RX_FIFO. При этом внутри схемы был реализован аппаратный таймер, значения которого сохранялись и в дальнейшем отправлялись на хост. Интерфейсом EP TX CMD управлял хост через пользовательские регистры.

Наконец, для измерения латентности использовалась схема, передающая данные в другую ПЛИС. Принимающая данные ПЛИС сразу же записывала их обратно в то же устройство. В передающей ПЛИС одновременно с началом передачи запускался аппаратный таймер, который останавливался в тот момент, когда данные начинали поступать в очередь EP_RX_FIFO. Далее значение таймера можно было считать на хост через пользовательские регистры.

На хосте была установлена ОС Linux. Для работы с аппаратурой применялись драйверы и библиотеки собственной разработки.

3. Пропускная способность шины PCI Express

Прежде чем переходить к описанию экспериментов по измерению скорости передачи данных необходимо выяснить ее теоретический предел. Известно, что из-за применения кодирования 8B/10B максимальная теоретическая скорость передачи информации по одной линии PCI Express на частоте 2.5 ГГц (gen1) равна $V_{theory} = 2.0\,Гбит/с$. Для протокола второй генерации Gen 2.0 с частотой 5 ГГц эта скорость в 2 раза выше ($V_{theory} = 4.0\,Гбит/с$). Скорость передачи информации для третьей генерации выше еще в 2 раза и равна $V_{theory} = 8.0\,Гбит/с$ на одну линию (несмотря на то, что частота равна 8 ГГц, в протоколе третьей генерации применяется другой метод кодирования символов 128B/130B снижающий дополнительную нагрузку).

Однако данные передаются с чуть меньшей скоростью из-за того, что передача идет пакетами, включающими в себя дополнительную информацию (стартовые/стоповые биты, заголовок, контрольная сумма и тд). В итоге при передаче одного пакета транзакции записи помимо данных передаются дополнительные 20 байт, относящиеся к этому же пакету. Также по шине PCI Express передаются пакеты, вовсе не содержащие данные, а выполняющие чисто служебные функции. К таким можно отнести подтверждения о приеме пакетов с данными, требования повторить передачу в случае обнаружения несоответствия контрольной суммы, пакеты, обновляющие счетчики буферов свободного места в коммутаторах, и другие. Точно оценить их влияние на скорость передачи данных сложно (это зависит от конкретной реализации), однако можно в среднем примерно оценить их вклад как 3 дополнительных байта на 1 пакет с данными (8). Итого мы принимаем, что на передачу одного пакета с данными в среднем передается 23 дополнительных служебных байтов. Более подробно об этом написано в (8). В дальнейшем, если не оговорено обратное, под длиной пакета мы будет понимать количество данных в пакете.

По шине PCI Express данные могут передаваться пакетами разной длины. Максимальное количество данных при передаче одного пакета определяется параметром MAX_PAYLOAD_SIZE, значение которого равно степени двойки. У каждого устройства есть параметр MAX_PAYLOAD_SIZE_SUPPORTED, определяющий максимальный размер пакета, который может быть передан этим устройством. Конфигурационное ПО (программа BIOS) настраивает параметр MAX_PAYLOAD_SIZE для всех устройств в системе равным наименьшему из значений, поддерживаемых устройствами в системе. Как правило, современные чипсеты поддерживают размер пакетов до 128 байт, однако есть системы и с 256 байтами. В наших экспериментах параметр чипсета MAX_PAYLOAD_SIZE_SUPPORTED был равен 128 байт, и, несмотря на то, что устройства ПЛИС и PCI Express коммутаторы на платах RHA-25 и RC-47 поддерживали большие размеры пакетов (до 512), BIOS

настроила параметр MAX_PAYLOAD_SIZE для всех устройств в системе равным 128 байт.

Чем больше размер передаваемых пакетов, тем ближе пропускная способность приближается к теоретическому пределу передачи информации. Если теоретический предел принять равным 1, то по следующей формуле можно вычислить практический предел передачи данных в зависимость от размера пакета (см. табл. 1):

$$\xi = \frac{x}{x + 23}$$

$$V_{prac} = V_{theory} * \xi \qquad (1)$$

где x – размер пакета.

Таблица 1. Зависимость относительной скорости передачи данных от размера пакета

x, байт	4	8	16	32	64	128	256	512	1024
ξ	0,148	0,258	0,41	0,581	0,735	0,847	0,917	0,957	0,978

В табл. 2 приведено сравнение максимальных теоретических и практических скоростей передачи собственно данных для длины пакета равной 128 и 256 байт.

Таблица 2. Максимальные скорости передачи информации и данных на шине PCI Express для пакетов длиной 128 и 256 байт

Интерфейс	Ширина	V_{theory}, Гбит/с (10^9 $бит$)	V_{theory}, МБайт/с (2^{20} $байт$)	Payload 128	Payload 256
				V_{prac}, Мбайт/с (2^{20} $байт$)	V_{prac}, Мбайт/с (2^{20} $байт$)
PCIe 1.0	x1	2	238	201	218
	x2	4	476	403	436
	x4	8	953	806	873
	x8	16	1907	1612	1748
PCIe 2.0	x1	4	476	403	436
	x2	8	953	806	873
	x4	16	1907	1612	1748
	x8	32	3814	3225	3497
PCIe 3.0	x1	8	953	806	873
	x2	16	1907	1612	1748
	x4	32	3814	3225	3497
	x8	64	7629	6450	6995

4. Передача HOST-ПЛИС

При передаче данных между ПЛИС и системной памятью используется механизм прямого доступа в память. Пользовательское приложение на хосте подготавливает буфер в ОЗУ и делает системный вызов write или read. PCI Express драйвер устройства фиксирует страницы пользовательского буфера в памяти и использует механизм scatter/gather DMA. Список дескрипторов (пара адрес-длина) страниц записывается во внутреннюю память устройства ПЛИС, а затем ПЛИС сама обращается в память по адресам из этого списка. После завершения передачи данных ПЛИС генерирует прерывание, которое и завершает системный вызов. Когда пользовательское приложение намерено записать данные в ПЛИС, устройство обращается в ОЗУ с транзакциями чтения, а при чтении из ПЛИС устройство генерирует транзакции записи.

Далее везде будет подразумеваться, что слова «запись» и «чтение» относятся к ПЛИС, т.е. скорость записи подразумевает скорость процесса записи, начатого ПЛИС.

Транзакции записи, генерируемые на шине PCI Express, всегда однонаправлены. Инициатор транзакции записи формирует пакет, состоящий из заголовка и данных. Инициатор сам определяет размер данных, учитывая только ограничение MAX_PAYLOAD_SIZE. Все, что было сказано про скорость передачи данных в предыдущем параграфе, относится именно к транзакциям записи. Для транзакций записи скорость передачи данных легко оценить теоретически и измерить ее зависимость от размера передаваемого пакета.

С транзакциями чтения все немного сложнее. Инициатор транзакции чтения сначала формирует запрос на чтение – короткий пакет, состоящий только из заголовка. В этом пакете указывается, откуда (с какого адреса) и сколько необходимо считать данных. Максимальное количество данных, которое может быть запрошено за один раз определяется параметром MAX_READ_REQUEST_SIZE и обычно равно 4 КБ. Обычно периферийные устройства обращаются с запросами на чтение в ОЗУ, но могут запросить данные и из другого устройства. Когда устройство (периферийное или контроллер ОЗУ) получает запрос на чтение, сначала оно запрашивает требуемые данных из своей памяти, а затем возвращает их инициатору транзакции по шине PCI Express, генерируя ответные завершающие пакеты (completion request). При этом оно само определяет размер возвращаемых пакетов, опять же учитывая только ограничение MAX_PAYLOAD_SIZE. Инициатор транзакции не может повлиять на размер возвращаемых пакетов. Как правило, контроллер ОЗУ будет возвращать пакеты с длиной данных равной MAX_PAYLOAD_SIZE. Скорость передачи данных для транзакций чтения оценить сложно по нескольким причинам. Во-первых, генерируются пакеты двух типов, двигающиеся в противоположных направлениях – запросы на чтения и завершающие ответы. Во-вторых, на скорость передачи данных будут влиять задержки, возникающие при чтении из ОЗУ. Наконец, неясно, как проследить зависимость скорости передачи данных от количества данных в пакете. Поэтому в данной работе мы просто измеряли интегральное время от посылки первого запроса на чтение до прихода последнего байта данных, по нему вычисляли скорость чтения данных из ОЗУ и не прослеживали ее зависимость от размера пакета.

Шина PCI Express обеспечивает надежную передачу данных на уровне транзакций, т.е., пересылая данные, агенты (оконечные устройства и промежуточные коммутаторы) автоматически вычисляют контрольную сумму пакета, сравнивают ее с закодированной в самом пакете и требуют повторной передачи в случае обнаружения ошибки при передаче данных. Однако это не означает, что приложение на хосте или схема ПЛИС не могут сформировать и передать неправильные данные в результате ошибки программиста. Поэтому

для верификации корректности нашей схемы, включающей в себя приложение, драйвер и схему ПЛИС, были проведены проверки корректности передачи данных. Для этого использовалась первая схема User IP, в которой выход RX_FIFO замыкался на вход TX_FIFO. Размеры очередей FIFO были равны 4 КБ. Были проведены два эксперимента. В первом хост последовательно записывал в ПЛИС 4 КБ данных, затем их считывал и сравнивал. Второй эксперимент использовал тот факт, что тракты приема и передачи данных в ПЛИС могут работать параллельно. Хост сначала программировал обе операции записи и чтения, а затем ПЛИС начинала считывание данных из ОЗУ, и как только они поступали сначала в RX_FIFO, а затем и в TX_FIFO, сразу же начинала их запись обратно в ОЗУ. Это позволило передавать за раз намного больший объем данных по сравнению с размером очередей RX_FIFO и TX_FIFO (в нашем эксперименте 4 МБ). В обоих экспериментах сравнение переданных и принятых данных проходило успешно, что и позволило нам судить о правильности работы нашей схемы.

Эксперименты по измерению скорости передачи данных начались с измерения зависимости скорости записи в ОЗУ одного устройства ПЛИС от размера данных в передаваемых пакетах. Проводились эксперименты со значениями payload равными 8, 16, 32, 64 и 128 байт. В каждом эксперименте передавалось 4 МБ в одну сторону. Узким местом в тракте Virtex7 – ОЗУ было соединение PCI Express x4 Gen 2.0 между ПЛИС и коммутатором PCI Express на плате RC47 (см. рис. 1). Результаты представлены на рис. 3. Верхняя кривая Vtheory представляет собой теоретическую зависимость (формула 1) максимально достижимой скорости передачи данных от длины пакета (для интерфейса PCI Express x4 Gen 2.0). Кривая посередине Vhard представляет собой скорость, измеренную с помощью аппаратного таймера в ПЛИС и учитывающую только непосредственно передачу данных на шине PCI Express (время замерялось от начала передачи первого пакета с данными до конца передачи последнего). Наконец, кривая Vapp представляет собой скорость, измеренную в приложении на хосте (измерялось время выполнения системного вызова read).

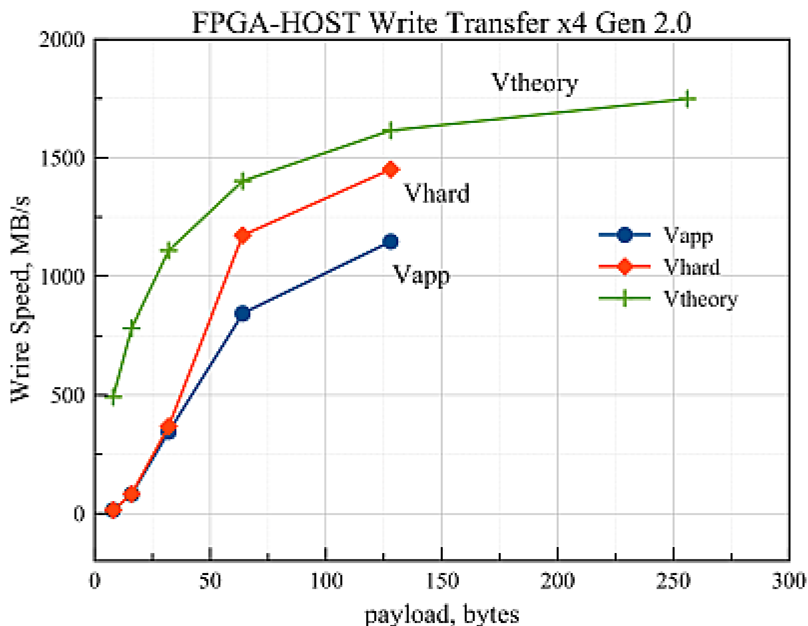


Рис. 3 Зависимость скорости записи в ОЗУ от длины пакета для интерфейса PCI Express x4 Gen 2.0

Из графика видно, что при значениях payload меньше 64 байт скорости Vhard и Vapp намного меньше Vtheory. Это объясняется тем, что в качестве приемника данных в данном случае выступает DDR память ОЗУ, обеспечивающая высокую скорость записи только в пакетном режиме (burst), передавая большое количество данных за одну транзакцию. Для пакета размером 128 байт Vhard = 1451 МБ/с, что составляет 90% от максимального значения 1612 МБ/с. Также видно, что скорость, измеренная в приложении Vapp (1146 МБ/с при payload=128) намного ниже Vhard для значений payload, начиная с 64 байт. Это связано с тем, что при передаче 4 МБ данных формируется порядка 1000 дескрипторов страниц, которые процессор записывает в ПЛИС. Эта начальная задержка (порядка 1 мс) существенно влияет на скорость передачи данных. Полное время выполнения системного вызова read для payload = 128 байт равно примерно 3,7 мс. Если из этого времени отнять начальную задержку в 1 мс, то получиться скорость примерно равная 1450 МБ/с, что совпадает со скоростью Vhard, измеренной аппаратно. В наших планах стоит изменение логики драйвера и схемы Rosta DMA Engine для уменьшения начальной задержки при программировании DMA передачи. Идея усовершенствования заключается в том, чтобы не передавать весь список дескрипторов страниц пользовательского буфера в ПЛИС, а вместо

этого сохранить его в доступной для ПЛИС области в ОЗУ. Тогда ПЛИС сама сможет вычитывать из ОЗУ дескрипторы, по которым уже будут передаваться данные. Процесс вычитывания дескрипторов из памяти и собственно передачу данных можно будет запустить параллельно, тем самым существенно сократив начальную задержку и как следствие, увеличив скорость передачи данных. Пока же мы будем ориентироваться на скорость, измеренную аппаратно. Также в последующих экспериментах не будут исследоваться передачи данных с payload меньше 128 байт.

В следующем эксперименте запускалось одновременно несколько передач данных с участием разных ПЛИС. Приложение на хосте запускало несколько потоков pthreads – по одному потоку на отдельную передачу. В каждом потоке ПЛИС сначала программировалась для чтения из ОЗУ, а затем на запись. Одновременность передачи достигалась использованием барьерной синхронизации в приложении. Размер передаваемого буфера был равен 4 МБ, размер пакетов при записи был равен 128 байт. В эксперименте были задействованы все 8 устройств ПЛИС Virtex7, содержащихся в блоке RB-8V7. Скорости записи данных в ОЗУ в зависимости от количества одновременно работающих ПЛИС представлены на рис. 4.

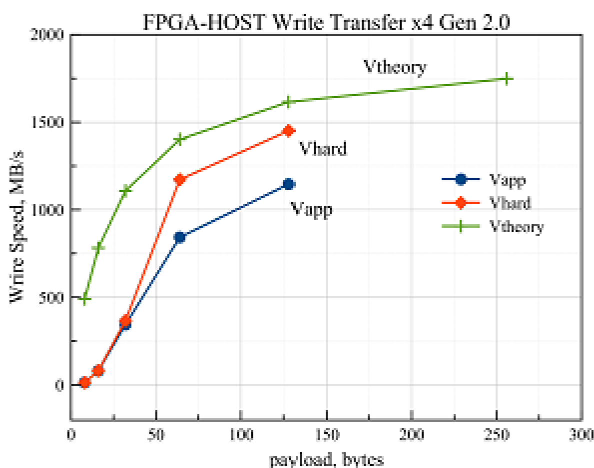


Рис. 4 Зависимость скорости записи в ОЗУ от количества одновременных транзакций

Две нижние кривые (V_average и V_average_app) отображают среднюю скорость записи, измеренную аппаратно и в приложении соответственно. Каждая отдельно взятая ПЛИС ограничена по скорости максимальной скоростью записи данных через интерфейс PCI Express x4 Gen 2.0, в нашем случае 1451 МБ/с. Кривые V_sum и V_sum_app – это суммы скоростей передачи данных отдельных устройств. Прямая Vmax = 3225 МБ/с

представляет собой максимальную скорость передачи данных через узкое место системы, ограничивающее скорость одновременной передачи данных. Таким узким местом является соединение PCI Express x8 Gen 2.0 адаптера RHA-25 с материнской платой компьютера. Для одной и двух одновременных передач скорость записи одинакова (1451 МБ/с), потому что суммарная скорость двух передач меньше Vmax. Начиная с трех передач, скорость записи отдельного устройства падает, однако суммарная скорость одинакова и равна Vmax. То, что на графике суммарная скорость для трех и более устройств превышает Vmax, объясняется «псевдоодновременностью» передачи данных. Как бы хорошо потоки не были параллельно распределены между ядрами центрального процессора, по шине PCI Express команды все равно идут последовательно. Поэтому какие-то устройства начинают передачу раньше, другие позже. Это приводит к тому, что в течение короткого промежутка времени вначале и в конце передачи данных общий канал соединения адаптера и материнской платы используется не всеми участвующими в передачи устройствами. Поэтому для них скорость передачи получается больше, а сумма всех скоростей превышает максимальную. Реально же получается, что общий канал используется на все 100%, а пропускная способность разделяется между устройствами в равных долях.

Аналогично можно рассмотреть случай чтения из ОЗУ (рис. 5).

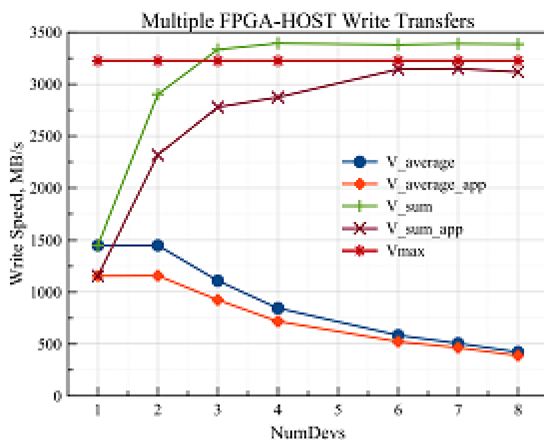


Рис. 5 Зависимость скорости чтения из ОЗУ от количества одновременных транзакций

Тут видно, что для передачи данных с участием от одного до трех устройств скорость чтения для каждого устройства одинакова и равна 1000 МБ/с. Начиная с четырех устройств, скорость ограничивается пропускной способностью канала связи с хостом.

5. Передача ПЛИС-ПЛИС

Запросы на запись и чтение памяти на шине PCI Express направляются в соответствии с адресом, закодированным в заголовках пакетов. Ведущее устройство способно сгенерировать пакет с произвольным адресом. Этот адрес может указывать в оперативную память, а может принадлежать области адресов, выделенных другому периферийному устройству. В последнем случае пакет запроса будет направлен от одного устройства к другому. В нашем случае таким образом передавались запросы на запись между разными ПЛИС. На чтение тесты не проводились.

Каждая ПЛИС получает от BIOS или от операционной системы диапазон адресов, по которым можно обратиться в это устройство. Чтобы запрограммировать передачу данных от ПЛИС A в ПЛИС B, приложение на хосте должно сообщить ПЛИС A (записать в соответствующий регистр в блоке User IP) базовый адрес ПЛИС B. В ПЛИС в блоке Rosta DMA Engine был реализован аппаратный интерфейс EP TX CMD, предназначенный для инициации передачи данных в другую ПЛИС изнутри схемы. Схема записывает данные для передачи в EP_TX_FIFO и передает через интерфейс EP TX CMD базовый адрес другого устройства и длину передачи. Далее передающий автомат TX_STATE_MACHINE в блоке Rosta DMA Engine начинает передачу данных из EP_TX_FIFO по указанному адресу. В принимающей ПЛИС данные записываются в очередь EP_RX_FIFO.

Для начала надо было убедиться в корректности передачи данных между ПЛИС. Для этого был поставлен следующий эксперимент. В передаче была задействована последовательность из всех 8 ПЛИС Virtex7, входящих в состав блока RB-8V7 (рис. 6). В первые 7 ПЛИС хост записывал базовые адреса устройств по следующей схеме. В первое устройство – базовый адрес второго, во втрое – третьего и тд. Первое устройство хост программировал на чтение данных из ОЗУ и на передачу считанных данных во второе устройство. Для этого в User IP блок коммутации Interconnect через регистры User IP настраивался на соединение выхода RX_FIFO с входом EP_TX_FIFO. Устройства 2-7 были настроены на передачу данных в следующие по цепочке. В них блок Interconnect был настроен на соединение EP_RX_FIFO с входом EP_TX_FIFO. Наконец, восьмое устройство было запрограммировано на передачу принятых в EP_RX_FIFO данных через TX_FIFO обратно в ОЗУ. После окончания передачи хост сравнивал данные. В данном эксперименте также измерялась скорость передачи данных в тракте. Она была равна скорости чтения из ОЗУ одним устройством ПЛИС.

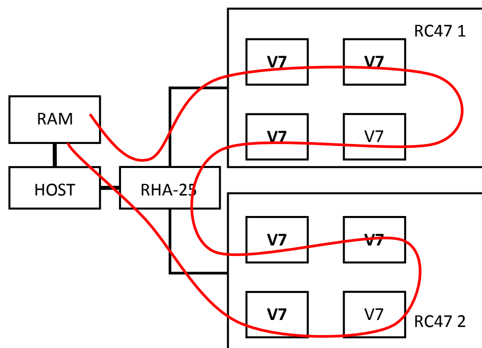


Рис. 6 Схема передачи данных HOST-ПЛИС-HOST

Удостоверившись в корректности передачи данных можно было переходить к измерению скорости. Из-за того, что максимальная длина пакетов, поддерживаемая чипсетом, была равна 128 байтам, параметр MAX_PAYLOAD_SIZE для всех коммутаторов и оконечных устройств в системе был установлен равным 128 байт. Поэтому по умолчанию передача данных между ПЛИС шла пакетами такой же длины. Однако было сделано наблюдение, что промежуточные PCI Express коммутаторы, находящиеся на платах RC-47 и адаптере RHA-25, а также интерфейсы PCI Express внутри ПЛИС поддерживали длину пакета равную 256 байт. При этом сам чипсет не участвовал в передаче данных между ПЛИС. Было сделано предположение, что если настроить MAX_PAYLOAD_SIZE для всех устройств в тракте передачи данных ПЛИС-ПЛИС равным 256, то можно будет запустить передачу пакетов длиной 256 байт, несмотря на то, что чипсет поддерживает только 128.

Для изменения параметра MAX_PAYLOAD_SIZE с помощью Linux команды setpci была проведена запись в регистр PCI Express Device Control для всех устройств и портов коммутаторов на платах RHA-25 и RC-47. Также был изменен блок Rosta DMA Engine так, чтобы можно было генерировать пакета длиной 256 байт. После чего действительно было возможно организовать передачу данных с блинной пакетов 256, и скорость передачи данных возросла.

Измерения скорости записи ПЛИС-ПЛИС были проведены для разного количества одновременных передач. Результаты представлены в таблице 3.

Таблица 3. Зависимость скорости записи ПЛИС-ПЛИС от количества одновременных передач при длинах пакетов 128 и 256 байт.

Количество передач	1	2	3	4	8.1	8.2
Скорость при payload = 128, МБ/с	1603	1549	1530	1520	1520	808
Скорость при payload = 256, МБ/с	1740	1704	1696	1685	1685	870

В первых четырех случаях передачи шли между устройствами на одной плате RC47. Максимальная скорость записи была получена во время одной передачи данных и составила 1603 МБ/с для длины пакета 128 байт и 1740 МБ/с для длины пакета 256 байт. В обоих случаях скорость составляла 99% от максимально возможной для соответствующей длины пакета.

Схемы взаимодействия устройств для случаев 8.1 и 8.2 представлены на рис.7.

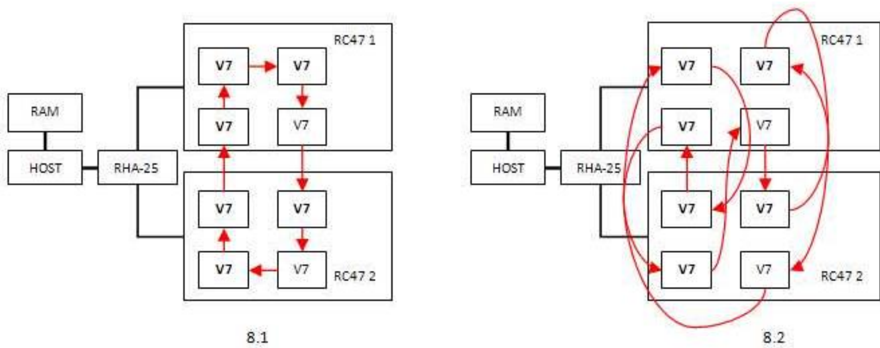


Рис. 7 Взаимодействие ПЛИС в случае 8ми одновременных передач

В случае 8.1 данные передавались последовательно от ПЛИС к ПЛИС сначала по кругу на одной плате, а затем через внешнее кабельное соединение и адаптер RHA-25 передача продолжалась в ПЛИС на другой плате. На второй плате данные также последовательно переписывались из одной ПЛИС в другую, а затем возвращались на первую. В итоге через адаптер RHA-25 в каждую сторону шло по одной передаче данных, и кабельное соединение PCI Express x4 Gen 3.0 никак не ограничивало ее скорость. В случае 8.2 данные передавались от одной ПЛИС к другой, но находящейся на другой плате. В итоге через адаптер RHA-25 шло 8 одновременных передач, по 4 в каждую сторону. Для длины пакета 128 байт скорость общего потока данных в одну сторону был равен 4*1520 = 6080 МБ/с, что превышает максимальную

скорость для канала PCI Express x4 Gen 3.0 равную 3225 МБ/с. Поэтому скорость канала должна была быть поделена поровну между устройствами, а средняя скорость передачи данных в каждой паре должна была бы стать 3225/4 = 806 МБ/с. Что подтверждается измеренным значением в 808 МБ/с. Поэтому можно утверждать, что канал связи плат между собой оказался задействован на 100%. Такие же рассуждения можно сделать и для передачи пакетов длиной 256 байт.

Наконец, был проведен эксперимент по измерению латентности передачи данных. Идея эксперимента была следующей. ПЛИС A посылает данные в ПЛИС B и запускает аппаратный таймер. Часы в разных устройствах сложно синхронизовать, поэтому используются таймер только в ПЛИС A. Как только ПЛИС B получает данные, она их сразу же начинает записывать обратно в ПЛИС A. Таймер останавливается, как только ПЛИС A получает первый байт данных. Значение таймера представляет собой удвоенную латентность передачи данных между блоком User IP одной ПЛИС и User IP другой. Латентность измерялась в двух случаях: при передачи между ПЛИС на одной плате RC-47 (рис. 8 A) и между ПЛИС на разных платах (рис. 8 B).

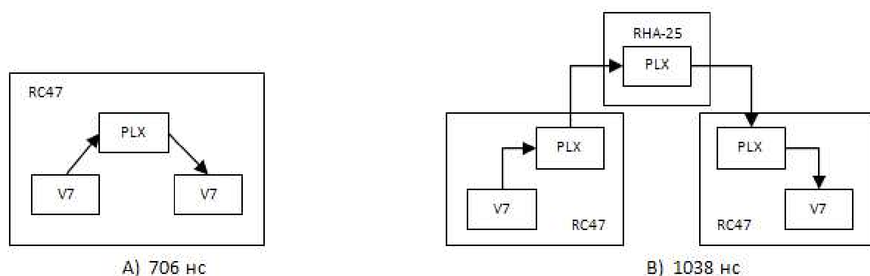


Рис. 8 Схемы передачи данных при измерениях латентности

При передаче данных между устройствами на одной плате RC-47 в тракте передачи был один коммутатор PLX, и задержка была равна 706 нс. При передаче с плату на плату было три промежуточных коммутатора, а задержка была равны 1038 нс. По этим данным можно определить задержку, возникающую в ПЛИС и вносимую коммутатором. Задержка в ПЛИС на приеме и передачи оказывается равной 270 нс, а в коммутаторе 166 нс, что хорошо согласуется с заявленной фирмой PLX Technology латентностью своих коммутаторов равной 150 нс.

6. Заключение

В данной работе была описана передача данных по шине PCI Express с одновременным участием нескольких ПЛИС. При записи в ОЗУ была получена скорость равная 90% от максимальной для соединения PCIe x4 Gen 2.0 при длине пакета 128 байт (1451 МБ/с). При чтении из ОЗУ скорость составила 1000 МБ/с. В случае одновременных передач ПЛИС-HOST скорость передачи данных не уменьшалась пока количество одновременных передач не насыщало узкий канал связи с хостом, в дальнейшем канал использовался на 100%, а его пропускная способность разделялась равномерно между устройствами.

Во время передачи ПЛИС-ПЛИС удалось запустить обмен пакетами длиной 256 байт, хотя чипсет хост компьютера поддерживал только 128. В этом случае удалось получить скорость равную 1740 МБ/с, что составляет 99% от максимальной скорости передачи интерфейса PCI Express x4 Gen 2.0 при длине пакета 256 байт. Также было показано, что возможно запустить несколько одновременных передач ПЛИС-ПЛИС, и пока суммарная скорость передачи не превышает пропускную способность общего канала, скорость отдельных передач не уменьшается, а затем канал используется на 100%, а его пропускная способность делится поровну между устройствами.

Была измерена латентность передачи данных ПЛИС-ПЛИС, которая составила 706 нс для одного промежуточного коммутатора и 1038 нс для трех.

Все это позволяет считать, что подход, основанный на использовании ПЛИС и IP ядра PCI Express интерфейса Xilinx и коммутаторов PLX Technology, может быть эффективно использован для организации обмена данными между большим количеством ПЛИС, подключенным к локальной PCI Express шине компьютера.

Список литературы

[1]. Rosta LTD, 2013. http://www.rosta.ru/.

[2]. nVidiaCorporation. GPUDirect. https://developer.nvidia.com/gpudirect.

[3]. Ra Inta, David J. Bowman, Susan M. Scott, "The “Chimera”: An Off-The-Shelf CPU GPGPU FPGA Hybrid Computing Platform", *International Journal of Reconfigurable Computing*, 2012.

[4]. Bruno da Silva, An Braeken, Erik H. D'Hollander, Abdellah Touhafi, Jan G. Cornelis, Jan Lemeire, "Performance and toolchain of a combined GPU/FPGA desktop", *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2013.

[5]. Ray Bittner, Erik Ruf, Alessandro Forin, "Direct GPU/FPGA Communication Via PCI Express", *Cluster Computing,* 2013.

[6]. Sunita Jain, Guru Prasanna, "Point-to-Point Connectivity Using Integrated Endpoint Block for PCI Express Designs", Xilinx Corporation, XAPP869, 2007.

[7]. 7 Series FPGAs Integrated Block for PCI Express v1.7 Product Guide, Xilinx Corporation, 2012.

[8]. Alex Goldhammer, John Ayer, "Understanding Performance of PCI Express Systems", Xilinx Corporation, WP350, 2008.

Direct data transfer between FPGAs Virtex-7 via PCI Express bus

Yu.A. Rumyantsev <rumyantsev@rosta.ru>
Rosta Ltd, Moscow, Russia

Abstract. This article describes two types of data transfers via PCI Express bus involving several FPGA. The first one is a simultaneous DMA data transfer between the system memory and different FPGA chips. The second one is a simultaneous direct data transfer between different FPGA.

The data transfer speed was measured for both cases with results being about 99% from maximum speed for PCIe x4 Gen 2.0 link for the direct transfer between FPGAs (1603 MB/s for 128 bytes payload and 1740 MB/s for 256 bytes payload). The direct data transfer latency was also measured to be 0,7 us for one intermediate PCIe switch and 1 us for three intermediate switches.

Also the effect of simultaneous transfers on data transfer speed was studied with the result that, as long as the aggregate transfer speed does not overcome the shared link bandwidth, each transfer is performed on its maximum speed; after that the shared link utilization reaches 100% with its bandwidth being distributed equally between individual transfers.

Keywords: FPGA; PCI Express.

References

[9]. Rosta LTD, 2013. http://www.rosta.ru/.

[10]. nVidiaCorporation. GPUDirect. https://developer.nvidia.com/gpudirect.

[11]. Ra Inta, David J. Bowman, Susan M. Scott, "The “Chimera”: An Off-The-Shelf CPU GPGPU FPGA Hybrid Computing Platform", International Journal of Reconfigurable Computing, 2012.

[12]. Bruno da Silva, An Braeken, Erik H. D'Hollander, Abdellah Touhafi, Jan G. Cornelis, Jan Lemeire, "Performance and toolchain of a combined GPU/FPGA desktop", Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays, 2013.

[13]. Ray Bittner, Erik Ruf, Alessandro Forin, "Direct GPU/FPGA Communication Via PCI Express", Cluster Computing, 2013.

[14]. Sunita Jain, Guru Prasanna, "Point-to-Point Connectivity Using Integrated Endpoint Block for PCI Express Designs", Xilinx Corporation, XAPP869, 2007.

[15]. 7 Series FPGAs Integrated Block for PCI Express v1.7 Product Guide, Xilinx Corporation, 2012.

[16]. Alex Goldhammer, John Ayer, "Understanding Performance of PCI Express Systems", Xilinx Corporation, WP350, 2008.

Методы оптимизации Си/Си++ - приложений распространяемых в биткоде LLVM с учетом специфики оборудования[1]

Курмангалеев Ш.Ф.
kursh@ispras.ru

Аннотация. В статье рассматриваются методы оптимизации Си/Си++ приложений, применяемые в системе двухэтапной компиляции, позволяющей распространять такие приложения в промежуточном представлении LLVM [1]. Рассматривается методика замещения кода функций во время исполнения, реализованная в динамическом компиляторе LLVM, позволяющая осуществлять динамическую оптимизацию программ. Описывается метод статического инструментирования с неполным покрытием всех дуг потока управления программы и последующей его коррекцией для сбора профиля, позволяющий получать профиль сравнимый по качеству с классическим подходом, но при этом обеспечивающий существенное снижение накладных расходов на сбор профиля. Помимо этого описывается разработанная и реализованная методика конвертации динамически собираемого профиля, для статически скомпилированного приложения в формат, используемый LLVM для машинно-независимых оптимизаций. Описываются как существующие в LLVM оптимизации, модифицированные для использования профиля, так и предлагаются новые.

Предлагается метод позволяющий использовать команды предвыборки для повышения эффективности кода обработки массивов. Описывается подход к построению специализированного облачного хранилища приложений, позволяющего решать вопросы оптимизации и защиты программ, а также обеспечивающий снижение накладных расходов на компиляцию и оптимизацию приложений в облачной инфраструктуре.

---

[1]Работа выполнена при финансовой поддержке Минобрнауки по государственному контракту от 15.06.2012 г. № 07.524.11.4018 в рамках ФЦП "Исследования и разработки по приоритетным направлениям развития научно технологического комплекса России на 2007-2013 годы"

Ключевые слова: Двухэтапная компиляция, оптимизация, LLVM, облачное хранилище.

1 Введение

В связи с широким распространением мобильных платформ имеющих ограниченные вычислительные ресурсы и жесткие требования к энергопотреблению становится актуальной задача оптимизации программы для конкретного пользователя, а также задача оптимизации под конкретную реализацию архитектуры. Оптимизация программы с учетом профиля конкретного пользователя на его машине:

- Учитывает особенности поведения данного пользователя, что позволяет ускорить выполнение программы именно для тех наборов входных данных, которые важнее для этого пользователя.
- Позволяет применять «дорогие» оптимизации только к часто выполняющимся участкам кода программы с использованием компиляции во время выполнения программы. Это уменьшает затраты на компиляцию программы, которые становятся особенно существенны при проведении оптимизаций на машине пользователя, при сохранении производительности (так как важные участки оптимизируются).

Оптимизация программы для учета архитектуры машины пользователя:

- Учитывает детальные параметры архитектуры (размер кэша, соотношение между частотой памяти и процессора, наличие специальных векторных инструкций). Эти параметры необходимо учитывать при оптимизациях обращений к памяти; при векторизации; при встраивании функций и развертке циклов. Полученная производительность будет заведомо не хуже, чем при выполнении машинно-зависимых оптимизаций на стороне разработчика, где известна только часть параметров целевой машины.
- Позволяют сократить для разработчика затраты на распространение и поддержку программы. Разработчику достаточно поддерживать одну версию программы, при сборке которой применялись лишь машинно-независимые оптимизации.

Для языков общего назначения (Си/Си++) в настоящий момент не существует решения обеих задач (специализации под пользователя и под его целевую машину) в виде общей среды, удобной для использования, как на стороне разработчика, так и на стороне пользователя.

Предлагаемый метод распространения программ написанных на языках Си/Си++, в промежуточном представлении позволяет указанные задачи

переносимости программ в пределах одного семейства процессоров с учетом специфических особенностей каждого конкретного процессора, проводить адаптивную компиляцию, учитывая поведение пользователя и характер входных данных. Собирая информацию о профиле программы, поступающую от пользователей, можно применить к промежуточному представлению машинно-независимые оптимизации для повышения быстродействия программы для наиболее часто встречающихся вариантов использования. Помимо этого распространение программы в промежуточном представлении позволяет применять средства статического анализа программ для поиска уязвимостей, и производить запутывание программ для защиты от обратного проектирования. Все указанные операции могут происходить как на машине пользователя, но это может привести к дополнительным накладным расходам, что является важным фактором в случае работы программы на мобильных устройствах. Однако этого можно избежать, переместив второй этап компиляции, анализ и запутывание программ на сервер приложений [2].

Требования, предъявляемые к программному обеспечению для мобильных платформ, такие как производительность и энергопотребление справедливы и для гетерогенных кластеров. Таким образом, создание облачной инфраструктуры, которая будет позволять осуществлять развертывание программного обеспечения с учетом конкретного аппаратного обеспечения каждого узла. А также производить оптимизацию программного обеспечения между запусками заданий. В качестве примера такого программного обеспечения можно рассматривать пакет OpenFoam (открытая интегрируемая платформа для численного моделирования задач механики сплошных сред). Поскольку пакет написан на языке Си++ с использованием виртуальных функций, то имеет смысл оптимизировать вызовы виртуальных функций, для чего и была предложена оптимизация спекулятивной девиртуализации[2].

В настоящей статье в разделе 2 описывается система двухэтапной компиляции программ на основе LLVM и изменения, внесенные в динамический компилятор LLVM. Описывается метод статического инструментирования обеспечивающий снижение накладных расходов на сбор профиля программы, рассматривается подход позволяющий конвертировать динамически собираемый профиля в формат LLVM. Раздел 3 описывает реализованные оптимизации: открытую вставку функций, вынос участков кода в отдельные функции, спекулятивная девиртуализация и использование команд предвыборки при обработке массивов в цикле. В разделе 4 описывается функциональность сервера приложений и снижение расходов на компиляцию с помощью динамического выбора уровня оптимизаций. Раздел 5 завершает статью.

---

[2] Здесь и далее под девиртуализацией понимается генерация прямого вызова функции без обращения к таблице виртуальных функций.

2 Двухэтапная компиляция

Кратко опишем предложенный поход: На первом этапе приложение компилируется на машине разработчика специальным набором инструментов основанных на LLVM, на этом этапе выполняются машинно-независимые оптимизации и оптимизации времени связывания (LTO - link time optimizations). Затем происходит определение зависимостей между программными компонентами и происходит генерация программного пакета содержащего файлы с промежуточным представлением LLVM и информацию о схеме инсталляции. На втором этапе программа оптимизируется и инсталлируется на машине пользователя. Во время оптимизации возможен учет поведения пользователя и особенностей аппаратуры, на которой будет выполняться программа. Поддерживается два варианта инсталляции: 1) генерация и инсталляция бинарной программы; 2) инсталляция программы для динамической компиляции. В случае динамической компиляции программы поддерживается система легковесного сбора профиля на основе Oprofile [3],[4]. В обоих случаях специальная программа «демон» позволяет осуществлять оптимизацию программы во время простоя системы с учетом профиля пользователя.

В систему двухэтапной компиляции, рассматриваемую в работах [2] и [3], были внесены изменения позволяющие повысить производительность и совместимость системы. Была реализована поддержка нового "gold" линкера [5], что позволило отказаться от двух последовательных запусков компилятора – одного для получения бинарного кода, и второго для получения эквивалентного биткода. Новый линкер позволяет производить компиляцию программы, генерируя объектные файлы содержащие биткод. Причем генерация бинарного кода производится только на этапе генерации финального модуля программы. Помимо этого поддерживается прозрачное связывание между объектными файлами, содержащими бинарный код и промежуточное представление LLVM. Поскольку применяемый ранее компоновщик имел ряд ограничений, по сравнению с компоновщиком из пакета Binutils [5][6], например отсутствие поддержки скриптов линкера, поддержка стандартного компоновщика позволяет повысить совместимость и расширить спектр программ, сборка которых происходит автоматически с помощью предлагаемых инструментов. Важной особенностью предлагаемого подхода, является поддержка сборки программ использующих стандартные системы сборки, основанные на Autotools[7].

В систему сборки пользовательских программ были внесены изменения, позволившие сократить как время компиляции программы, так и время генерации инсталляционного пакета, а также снять ранее присутствующие ограничения на расположение каталогов сборки и инсталляции программы. Время генерации инсталляционного пакета уменьшилось из-за того, что вместо затратной по времени процедуры сопоставления контрольных сумм исходных файлов и файлов в после развертывания программы, для получения

списка файлов включаемых в пакет используется утилита installwatch (пакета Checkinstall [8]), перехватывающая системные вызовы при копировании файлов.

Помимо этого была реализована поддержка автоматического преобразования RPM-пакетов, содержащих исходный код в пакеты, содержащие биткод LLVM и скрипты для развертывания программы. Формат RPM используется в распространенных дистрибутивах на базе Linux, таких как Red Hat, OpenSUSE, Fedora и в открытой мобильной платформе Tizen.

2.1 Изменения в динамическом компиляторе LLVM

Для обеспечения непрерывного учета поступающего профиля программы, от динамического компилятора требуется поддержка механизма замещения на стеке. Замещение на стеке – механизм, позволяющий изменять код часто исполняемых функций во время исполнения программы. Применяется во многих динамических компиляторах (HotSpot, Jikes RVM, CACAO и др.), но пока не реализован в динамическом компиляторе LLVM.

Динамическая компиляция в LLVM организована следующим образом: трансляция в целевой код происходит непосредственно в памяти машины по функциям. Если функция исполняется дольше или большее количество раз, чем остальные, то, вероятнее всего, если заново оптимизировать ее более агрессивными оптимизациями, то можно получить прирост производительности.

На базе системы двухэтапной компиляции была реализована поддержка замены на стеке. В качестве профилировщика используется модифицированный Oprofile, сбор и обработка поступающей от профилировщика информации происходит в реальном времени. Компиляция и обработка информации от профилировщика осуществляется в разных потоках: в одном осуществляется динамическая компиляция и запуск исполняемого кода, а второй осуществляет сбор информации от профилировщика и, в зависимости от полученных данных, принимая решение о том, необходимо ли компилировать функцию заново.

Решение о повторной компиляции принимается в зависимости от «температуры» функции: «горячие» функции должны будут быть повторно скомпилированы с более глубокими оптимизациями, «теплые» останутся без изменений, а код «холодных» функций можно удалить из памяти[9]. Вызов функции осуществляется через специальный переходник, который производит одно из следующих действий: если функция еще скомпилирована, компилирует ее, если для функции доступна оптимизированная версия, вызывается она, даже если первоначальная функция все еще находится в памяти. Для освобождения памяти занимаемой различными вариантами функций в начало и конец каждой из функций вставляется атомарная операция инкремента/декремента, что позволяет оценить количество потоков

использующих данную функцию, после обнуления количества активных пользователей, функцию можно безопасно удалить из памяти. Кроме того, для горячих функций имеющих небольшой размер, происходит удаление таких счетчиков, для сокращения накладных расходов.

Температурные характеристики выделяются методом скалярной кластеризации. Необходимо построить три кластера: «горячий», «теплый» и «холодный». Известно, что «горячему» будет принадлежать элемент с максимальной числовой характеристикой, «холодному» - с минимальной.

2.2 Статический профиль с неполным покрытием всех дуг потока управления программы

Было проведено исследование влияния статического профиля с неполным покрытием всех дуг потока управления программы и последующей его коррекцией, что позволило сократить накладные расходы, обуславливаемые инструментированием.

В LLVM применяется два подхода к инструментации. Первый метод вставляет счетчики на каждое ребро в программе (опция - insert-edge-profiling). Он собирает полный профиль, но замедление оказывается порядка 100%. Второй метод основан на законе Кирхгофа - общее вес входящих в блок ребер, равен общему весу исходящих, инструментируются только вершины входящие в максимальное остовное дерево графа потока управления (опция -insert-optimal-edge-profiling). Замедление инструментированной программы порядка 50%.

Предложенный метод состоит в том, чтобы инструментировать ребра входящие в максимальное остовное дерево с некоторой вероятностью и восстановлением на этапе обработки собранного профиля, для снижения накладных расходов при минимальном снижении качества профиля.

Во время тестирования на легковесной СУБД Sqlite было выявлено, что инструментация с помощью алгоритма «optimal-edge-profiling» замедляет программу примерно на 46%, а предложенный подход с 50% вероятностью вставки счетчика на 9%. При этом при использовании полученных профилей для оптимизации программы показывает, что производительность обоих версий различается ~1%. Из чего можно сделать вывод, что качество собираемого профиля примерно совпадает.

2.3 Конвертирование динамического профиля в формат LLVM

Также для обеспечения возможности сбора профиля для статически скомпилированной программы с помощью динамический профилировщиков был разработан и реализован алгоритм сопоставления базовых блоков в

промежуточном представлении программы с бинарным образом исполняемого файла, что позволило конвертировать такой профиль в формат LLVM.

Для этого, во время генерации кода, смещения базовых блоков относительно начала функции их размер сохраняются в специальный файл с расширением «.pmap». Каждый такой файл состоит из заголовка и блоков, идущих друг за другом. Заголовок состоит из полей:

слово «AMAP» | размер файлы
имя модуля
имя компилятора
версия компилятора
строка аргументов компилятору.

Каждый блок состоит из полей:

размер имени функции
имя функции
размер функции в базовых блоках
номер блока | смещение | размер
номер блока | смещение | размер
….
номер блока | смещение | размер.

Поскольку профиль, собираемый Oprofile, привязан к виртуальным адресам, для его корректной загрузки, после завершения генерации кода и компоновки модулей при помощи утилиты objdump извлекаются виртуальные адреса функций. Имя виртуальные адреса функций, и смещения базовых блоков относительно начала функции мы можем конвертировать профиль в формат LLVM.

3 Оптимизации учитывающие информацию о профиле программы

3.1 Открытая вставка функций

Оптимизация открытой вставки функции – оптимизирующее преобразование компилятора, вставляющее код функции на место его вызова в тело вызывающей функции.

Для реализации оптимизации открытой вставки функций была рассмотрена существующая реализация – оптимизация компилятора GNU GCC[10]. Данная оптимизация использует данные профилирования для решения вопроса о

вставке функции: относительная частота вызова функций задается параметром frequency в пределах от 0 до 1, количество вызовов задается параметром calls, а потенциальный рост общего количества инструкций задается параметром growth. Таким образом, решение о том, вставить ли малую функцию вычисляется по формуле:

growth < 0 → growth

growth >=0 → calls/growth или growth/frequency – в зависимости от того, включен глобальный или локальный профиль соответственно.
Для вычисления веса функции применяется следующая эвристика:

FunctionWeight= NumofInstructions*InstrucctionPenalty-NumArgguments*AllocaPenalty-NumofConstInstruction*ConstantPenalty, где

InsrucitonPenalty=2 – штраф за каждую инструкцию в функции

AllocaPenalty=2 – штраф за локальную переменную

ConstantPenalty=2 – штраф за константу

Вес функции с учетом информации из профиля вычисляется следующим образом:

NewWeight= (NumCallFromProfileInfo)/FunctionWeight

После присвоения весов, функции сортируются по возрастанию веса и встраиваются до тех пор, пока суммарный вес не превысит некоторого значения (по умолчанию 1000).
На тестах SQLite, Expedite, Cray и Coremark дала прирост скорости в ~2%.

3.2 Визуализация графа потока управления программы

Среди проходов LLVM имеется набор для визуализации графов потока управления функций, которые сохраняют каждый граф в формате Graphviz[11] в отдельный файл, либо выводят их на экран. Для удобства отладки оптимизаций использующих информацию о профиле программы была реализована возможность выбора нужной функции, а также аннотирование базовых блоков и ребер выводимого графа весами из профиля.

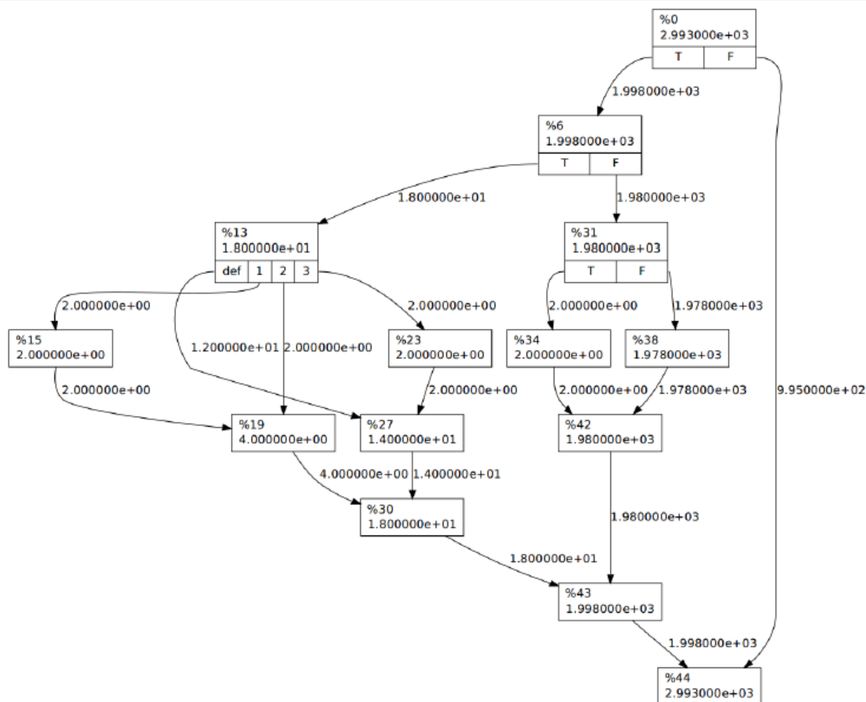


Рис. 1 Пример аннотированного графа функции

3.3 Вынос "холодных" участков кода в отдельные функции

Для оптимизации предлагается рассматривать функции, которые исполняются наибольшее число раз (”горячие”). Если функцию условно можно разделить на 2 части - большой редко исполняемый участок кода, относительно малый "горячий" участок. Из таких функций предлагается выносить ”холодную” часть функции в отдельную новую функцию, уменьшая при этом размер рассматриваемой функции и расстояния в памяти между часто исполняемыми участками кода[12].

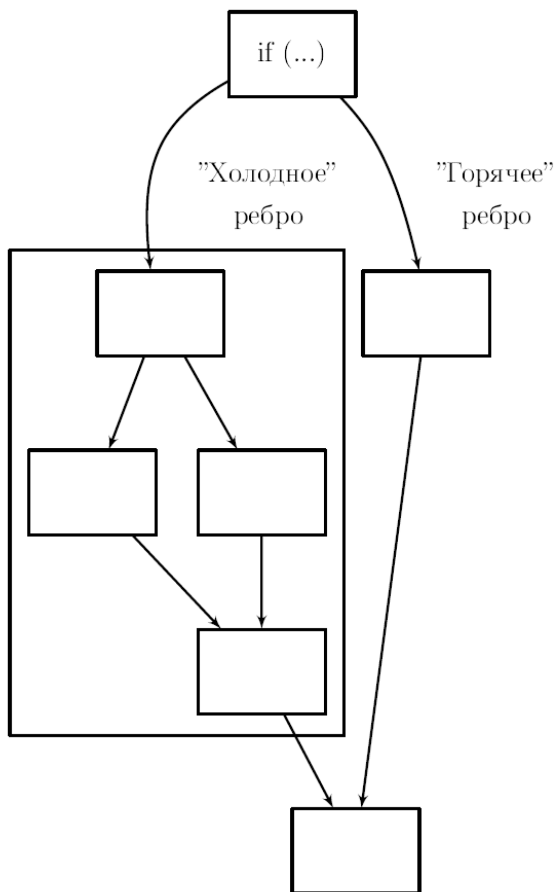
Для выделения ”горячих” функций используется одномерной кластеризация по весу из профиля с помощью алгоритма k-средних[13] при $k = 3$. Мы разделяем функции на 3 класса:

- ”горячие”
- ”средние”
- ”холодные”

Начальными центрами масс каждого класса выбраны максимальный, средний и минимальный веса функций соответственно.

Для выделения ”холодных” ребер внутри ”горячих” функций рассматриваются условные переходы - считаем ребро холодным, если оно имеет вес в 100 и более раз меньший другого ребра данного условного перехода. После выделения "холодного" ребра, мы должны выделить максимальный набор базовых блоков, в которые может попасть поток управления только при прохождении по выделенному на предыдущем шаге ребру. Выбираются все

Рис. 2 Рассматриваемый граф с выделенным для вынесения подграфом



блоки, над которыми доминирует блок, в который и входит выделенное ребро [12][14].

Рассмотрим некоторую часть графа потока управления "горячей" функции (см. рисунок 2).

Стоит заметить, что требуется корректно обрабатывать следующие случаи:

- множественные выходы из выносимого подграфа
- внешние - функции, содержащие переменные из выносимых блоков

Для случая с множественными выходами, мы создаем специальный блок, который будет являться единственной точкой выхода из функции. В созданном блоке автоматически выбирается нужный код возврата. Проблема с -функциями решается их разделением на две части, одна остается во внешнем коде, другая вставляется в последний блок выносимого кода, обрабатывая соответствующие переходы.

На тестах SQLite, Expedite, Cray и Coremark дала средний прирост скорости в 0,8%. При использовании ее вместе с оптимизацией встраивания получен средний прирост в ~3%. Размер исполняемого файла увеличивается на 1- 7%, в зависимости от приложения.

3.4 Спекулятивная девиртуализация

Для объектно-ориентированных языков программирования решение о вставке виртуальных функций является проблемой, для решения которой недостаточно знать количество ее вызовов. В программах, написанных на языке Си++, могут быть два типа виртуальных вызовов функций: классические вызовы по указателю на функцию и вызовы виртуальных методов классов. Когда компилятор встречает такие вызовы, он не может определить, какая функция будет вызываться. Чтобы понять, какая функция будет вызвана, необходимо произвести дополнительный анализ [15]. Этот анализ включает в себя: сравнение сигнатур функций, анализ иерархии наследования классов и анализ типов существующих в точке вызова. Сравнение сигнатур отсекает «неподходящие» по возвращаемому значению и параметрам функции. Анализ иерархии наследования – выявляет классы, для которых существует реализация виртуального метода и отсекает классы, находящиеся выше по иерархии, чем тип указателя. Анализ существующих в точке вызова объектов рассматривает, объекты каких классов были созданы и еще не уничтожены в момент вызова функции.

Помимо этого была добавлена возможность инструментирования вызовов виртуальных функций сохранением информации о количестве вызовов конкретной виртуальной функции. Таким образом, используя профиль, мы можем определить наиболее вероятного кандидата на девиртуализацию.

Реализованный алгоритм сочетает в себе вышеописанные методики. После проведения девиртуализации и принятия решения о вставке функции, если оказывается, что кандидат на вставку всего один, – вставляется он. Если кандидатов несколько - производится спекулятивная девиртуализация: по данным профилирования мы можем сказать, какая реализация виртуальной функции исполнялась наиболее часто, и вставляем инструкцию “if”, в теле которой производится вставка «горячей» функции, а в ветке “else” произведется вызов альтернативной, «холодной» версии функции.

Поскольку биткод LLVM не содержит высокоуровневой информации, анализ иерархии должен производиться в компиляторе переднего плана, с сохранением результатов в метаинформации биткода. Таким образом, понадобилось дополнительно реализовать экспорт метаинформации в компиляторе переднего плана Clang[16]. Во время тестирования был отмечен прирост производительности в 3% при увеличении размера кода всего на 1%.

Помимо этого, алгоритм успешно проходит синтетические тесты для тестирования девиртуализации предложенные сообществом GCC[17]. В настоящее время ведутся работы направленные на увеличение точности статического анализа

3.5 Использование команд предвыборки при обработке массивов в цикле

Оптимизация вставляет команды предвыборки для оптимизации использования кэша процессора во время последовательной загрузки данных из массивов в циклах.

Процессоры архитектуры ARM серии Cortex-A9 имеют встроенный автоматический механизм предвыборки данных, который загружает данные в кэш, учитывая промахи кэша[18], массив загружается в кэш после нескольких итераций и промахов кэша. Такое поведение не оптимально и может быть исправлено с помощью команды предвыборки “PLD”, которая указывает процессору, что вскоре будут использованы данные, на которые указывает команда, так что их желательно загрузить в кэш, если их там еще нет.

Данная оптимизация анализирует циклы с индукционной переменной в канонической форме – индукционная переменная имеет тип целого числа, инициализируется нулем и увеличивается на единицу каждую итерацию. В цикле в такой форме выделяются все загрузки из массивов, использующие индукционную переменную.

Команды предвыборки должны быть выполнены заранее, чтобы данные находились в кэше в тот момент, когда они будут использоваться. Слишком позднее выполнение команды предвыборки приведет к тому, что данные не будут загружены в кэш к моменту их использования. Оптимизация производится над промежуточным представлением LLVM, поэтому, чтобы эффективно генерировать команды предвыборки, необходимо оценивать

количество машинных команд в цикле. Мы используем приближенную оценку – считаем, что вызов функции преобразуется в столько машинных команд, сколько у функции аргументов плюс 1, некоторые команды промежуточного представления не преобразуются в машинные команды (фи-функции, команды “GetElementPtr”, некоторые команды преобразования типов), остальные команды преобразуются в одну машинную инструкцию.

После оценки размера цикла, вычисляется, на сколько итераций вперед следует совершать предвыборку данных. Эта величина равна отношению задержки загрузки данных в кэш после выполнения команды предвыборки к количеству команд в цикле. На процессоре ARM Cortex-A9 требуется выполнение около 200 команд после команды предвыборки, чтобы данные были загружены в кэш.

Также производится попытка определить количество итераций цикла. Если количество итераций не является константой и вычисляется во время выполнения программы, то оно определяется из собранного профиля программы, если он доступен.

Если же и профиль программы недоступен, то количество итераций цикла оценивается на основе вероятностной оценки ветвления программы. Подход заключается в оценке количества переходов по каждому ребру на основе эвристических признаков и статистических данных [19]. Данный метод оценивает количество раз выполнения каждого базового блока. Для того, чтобы оценка была целочисленной, количество раз выполнения первого базового блока функции считается равным 1024. Далее в каждой точке ветвления программы производится попытка определить сколько раз произойдет переход по каждому ребру.

Используемые эвристические признаки:

– В цикле переход в заголовок происходит чаще, чем выход из цикла (вероятность перехода в заголовок цикла – 97%).

– Переход в базовый блок с командой возврата происходит редко (его вероятность 25%).

– Сравнение двух указателей или указателя с NULL чаще всего дает отрицательный результат (вероятность этого 62.5%).

– Сравнение переменной с нулем чаще всего дает отрицательный результат (вероятность этого 62.5%).

Эвристики применяются последовательно до первого подходящего. Если ни один из признаков не применим к точке ветвления, то все переходы считаются равновероятными.

После применения эвристик каждому ребру графа потока управления соответствует вероятность перехода по этому ребру. Используя эту

информацию можно получить ожидаемую частоту выполнения каждого базового блока. По всем управляющим конструкциям языка, кроме цикла, частота распространяется тривиальным образом – пропорционально вероятностям исходящих из базового блока ребер. Для циклов вычисляется вероятность, учитывающая переходы по обратным ребрам. Используя данную вероятность вычисляется частота заголовка цикла. Поделив частоту заголовка цикла на частоту первого базового блока функции можно получить статистическую оценку количества итераций цикла.

Предвыборка данных улучшает производительность генерируемого кода, когда команды обращения к памяти чередуются с вычислительными командами ЦПУ. Если в оптимизируемом цикле нет достаточного количества вычислительных команд, то предвыборка не даст значительного выигрыша в производительности. В разработанной оптимизации используется оценка отношения количества вычислительных команд к командам работы с памятью. Для выигрыша в производительности это отношение должно быть больше установленного значения. Для процессора ARM Cortex-A9 значение данной величины равно 3.

Для того чтобы команды предвыборки не выполнялись слишком часто и не указывали на участки памяти, которые уже загружены в кэш, используется развертывание циклов. Цикл развертывается столько раз, чтобы загружаемые данные за один проход развернутого цикла полностью заполняли одну строку кэша. Например, если размер строки кэша 32 байта (как на процессоре ARM Cortex-A9), а размер загружаемых каждую итерацию данных равен 4 байта, то цикл стоит развернуть 8 раз (32 / 4), и вставить команду предвыборки лишь в первую итерацию. Тестирование на наборе тестов SPEC CPU 2000 показало, что прирост производительности составляет ~0.9%. На тестах SQLite, Expedite, Cray и Coremark дала прирост производительности от 0,5 до 5 %, средний прирост составляет ~2.5%

4 Сервер приложений

Предлагаемый метод двухэтапной компиляции позволяет проводить оптимизацию программы с учетом собранного профиля, как при динамической компиляции, так и во время простоя системы. Но для мобильных устройств зачастую оптимизация программ на устройстве является затруднительной. Для снижения нагрузки на устройство предлагается использовать специальный сервер приложений [2], при таком подходе приложения, скомпилированные в промежуточное представление LLVM, будут храниться в специальном облачном хранилище, там же будет происходить генерация бинарного кода и оптимизация программы с учетом информации о ее профиле. Поскольку для каждого приложения будет поступать профиль от нескольких пользователей, то усреднив полученный набор профилей и проведя машинно-независимую оптимизацию, мы получим промежуточное представление, более отвечающее реальным вариантам

использования. Используя информацию усредненного профиля, мы можем сократить расходы при компиляции приложений для новых пользователей, а также повысить производительность динамической компиляции для пользователей, использующих ее на своих устройствах.

Помимо решения задач связанных с производительностью, применение сервера приложений позволит проводить запутывание приложений и осуществлять статическую проверку их кода инструментами типа SVACE[20], что поможет обеспечить защиту от обратного проектирования, и повысить устойчивость приложений к использованию их уязвимостей. Применяя различные варианты запутывающих преобразований [21],[22] мы можем получить множество различных вариантов кода одного приложения, что затруднит создание универсальной программы эксплуатирующей уязвимость.

4.1 Динамический выбор уровня оптимизаций

Механизм динамического переключения между уровнями оптимизации предназначен для решения вопроса экономии времени компиляции на функциях, которые исполняются редко при одновременном улучшении кода часто используемых функций. Как показано в работе [23], посредством только изменения опций можно достичь значительных результатов даже для статических компиляторов.

Идея оптимизации состоит в том, чтобы не применять оптимизации для «холодных» функций, поскольку их оптимизация не оказывает существенного влияния на производительность программы. Но поскольку таких функций достаточно много, время затрачиваемое на их анализ и оптимизацию может составлять большую часть времени расходуемого на компиляцию и оптимизацию программы[24][25][26]. Такая методика применима, как для динамической компиляции, где важна быстрая компиляция, а дополнительная оптимизация может производиться во время работы программы, так и для первичной компиляции больших программных комплексов для сбора профиля, и последующей быстрой генерации оптимизированной версии программы[27][28][29].

Было реализовано 3 варианта сочетаний наборов оптимизаций:

- минимальный O0 (не оптимизировать вовсе) для «холодных» на O2(стандартный набор) для «горячих»
- средний O1(минимальный набор) для «холодных» и O3(агрессивные оптимизации) для «горячих»
- максимальный - O2 для «холодных», O3 для «горячих»

При тестировании на SQLite было выявлено, что при минимальном уровне экономится до 90% времени компиляции, при аналогичной производительности. Для среднего экономия составляет 2-5% при

производительности, лучшей, чем при обычной компиляции с O3 на 1-3%, и максимальный дает экономию ~5% при ускорении на 3-4% в сравнении с O3.

5 Заключение

В данной статье были рассмотрены изменения, внесенные в систему двухэтапной компиляции позволяющие повысить производительность и совместимость системы. Описывается подход позволяющий снизить накладные расходы на сбор профиля программы с помощью инструментации. Для обеспечения учета профиля во время работы динамического компилятора был реализован механизм замещения на стеке. В качестве вспомогательного инструмента для отладки оптимизаций была реализована возможность аннотации графа потока управления весами из профиля программы, при выводе его графическом виде. Были предложены оптимизации учитывающие профиль программы. На основе приложенных методов и разработанных технологий возможно создать облачное хранилище позволяющее обеспечить как переносимость программ, в пределах одной архитектуры, так и учет специфики конкретной аппаратуры на которой производится развертывание программы. С другой стороны предлагаемый метод распространения программы в промежуточном представлении позволяет обеспечить высокую степень надежности и безопасности приложений, распространяемых через облачное хранилище, поскольку делает возможным применение специализированных инструментов статического анализа и запутывания кода.

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.
[2]. А. И. Аветисян. Двухэтапная компиляция для оптимизации и развертывания программ на языках общего назначения. – Труды ИСП РАН – 2012. - №12.
[3]. А.И. Аветисян, К. Ю. Долгорукова; Ш. Ф. Курмангалеев, Динамическое профилирование программы для системы LLVM, 201171-82
[4]. OProfile official website. http://oprofile.sourceforge.net.
[5]. Ian Lance Taylor (2008). "A New ELF Linker". GCC Developers' Summit 2008. pp. 129–136. Retrieved 2013-03-06.
[6]. GNU Binutils, http://www.gnu.org/software/binutils
[7]. Introduction to the Autotools, http://www.dwheeler.com/autotools/
[8]. CheckInstall, http://asic-linux.com.mx/~izto/checkinstall/
[9]. Pratschner, S. An overview of the .net compact framework garbage collector.— 2004. http://blogs.msdn.com/b/stevenpr/archive/2004/07/26/197254.aspx.
[10]. Soman, S. Efficient and General On-Stack Replacement for Aggressive Program Specialization / Sunil Soman, Chandra Krintz // International Conference on Programming Languages and Compilers. — 2006. — . — http://www.cs.ucsb.edu/~{ }ckrintz/papers/osr.pdf.
[11]. Graphviz Documentation, http://www.graphviz.org/Documentation.php

[12]. Peng Zhao “Code and Data Outlining”, 2005.
[13]. Мандель И. Д. Кластерный анализ. — М.: Финансы и статистика, 1988.
[14]. Jun-Pyo Lee, Jae-Jin Kim, Soo-Mook Moon, Suhyun Kim ”Aggressive Function Splitting for Partial Inlining”, Seoul, South Korea, 2011.
[15]. David F. Bacon and Peter F. Sweeny, Fast Static Analysis of C++ Virtual Functuion Call, 1996
[16]. Clang: a C language family frontend for LLVM, http://clang.llvm.org/
[17]. GCC Free software foundation, http://gcc.gnu.org
[18]. Справочное руководство по процессорной архитектуре ARM., http://infocenter.arm.com
[19]. Youfeng Wu, James R. Larus. Static Branch Frequency and Program Profile Analysis. 27th International Symposium on Microarchitecture, 1994
[20]. Арутюн Аветисян, Андрей Белеванцев, Алексей Бородин, Владимир Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН том 21, 2011, стр. 23-38.
[21]. Ш.Ф. Курмангалеев, В.П. Корчагин, Р.А. Матевосян. Описание подхода к разработке обфусцирующего компилятора. Труды Института системного программирования РАН, том 23, 2012 г. Стр. 67-76.
[22]. Ш.Ф. Курмангалеев, В.П. Корчагин, В.В. Савченко, С.С. Саргсян. Построение обфусцирующего компилятора на основе инфраструктуры LLVM. Труды Института системного программирования РАН, том 23, 2012 г. Стр. 77-92.
[23]. Р. Жуйков, Д. Плотников, М. Варданян. Автоматическая настройка оптимизационных преобразований компилятора GCC для платформы ARM. Труды Института системного программирования РАН, том 22, 2012 г. Стр. 49-66.
[24]. K. Pettis, R. C. Hansen. Profile guided code positioning . SIGPLAN Not. 1990 June.Vol. 25, no. 6, pp. 16-27. http://dx.doi.org/10.1145/93548.93550.
[25]. P. P. Chang, S. A. Mahlke, W.-m. W. Hwu. Using profile information to assist classic code optimizations. Center for Reliable and High-Performance Computing. Urbana-Champaign: University of Illinois, 1991.
[26]. Da Silva, A. F. Our Experiences with Optimizations in Sun’s Java Just-In-Time Compilers / Anderson F. Da Silva, Vitor S. Costa // Journal of Universal Computer Science. — 2006. — Vol. 12.
[27]. M. Arnold, Stephen J. Fink, D. Grove, M. Hind, Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. IBM T. J. Watson Research Center, Hawthorne, USA, 2004.
[28]. Stephen Fink, David Grove, Michael Hind. Dynamic Compilation and Adaptive Optimization in Virtual Machines. IBM, June 2004.
[29]. Holzle, U. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming: Ph. D. thesis / Stanford University. —1994. — http://research.sun.com/self/papers/hoelzle-thesis.ps.gz.

Machine-specific optimization methods for C/C++ applications that are distributed in the LLVM intermediate representation format

Kurmangaleev Sh.F.
kursh@ispras.ru
ISP RAS, Moscow, Russia

Annotation. This paper analyzes approaches for optimizing C/C++ applications used in two-stage compilation system, allowing distributing such applications in the LLVM (low level virtual machine) intermediate representation. The on-stack replacement technique implemented in the LLVM just-in-time compiler is described. The paper presents a static instrumentation technique with incomplete control flow edge covering and its correction to the full control flow profile. The proposed approach allows the derived profile information to be comparable in quality to the classical approach while significantly reducing the profiling overhead.

Also, the technique for converting dynamically collected profile to LLVM profile format for statically compiled applications is presented. This paper evaluates both existing optimizations modified for using the collected profile and the newly developed ones. Implemented profile based inlining, outlining, speculative devirtualization. Also implemented annotation of control flow graph by profile weights to make profile based optimizations easy for debugging. Proposed the technique which allows using the prefetch instructions to improve the effectiveness of array processing codes. This paper describes the approach for building a specific cloud application storage that allows solving program optimization and protection problems. Finally, we present the approach for reducing compilation and optimization overhead in the cloud infrastructure.

Keywords: Two-stage compilation, optimization, LLVM, cloud storage

References

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

[2]. Arutyun Avetisyan. Dvukhehtapnaya kompilyatsiya dlya optimizatsii i razvertyvaniya programm na yazykakh obshhego naznacheniya. [Two-stage compilation for optimizing and deploying programs in general purpose languages]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 11-18 (in Russian).

[3]. A.I. Avetisyan, K.U. Dolgorukova; Sh.F. Kurmangaleev. Dinamicheskoe profilirovanie programmy dlya sistemy LLVM [Dynamic profile collection for LLVM]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 21, pp. 71-82 (in Russian)

[4]. OProfile official website. http://oprofile.sourceforge.net.

[5]. Ian Lance Taylor (2008). "A New ELF Linker". GCC Developers' Summit 2008. pp. 129–136.

[6]. GNU Binutils, http://www.gnu.org/software/binutils
[7]. Introduction to the Autotools, http://www.dwheeler.com/autotools/
[8]. CheckInstall, http://asic-linux.com.mx/~izto/checkinstall/
[9]. Pratschner, S. An overview of the .net compact framework garbage collector.— 2004. http://blogs.msdn.com/b/stevenpr/archive/2004/07/26/197254.aspx.
[10]. Soman, S. Efficient and General On-Stack Replacement for Aggressive Program Specialization / Sunil Soman, Chandra Krintz // International Conference on Programming Languages and Compilers. — 2006. — . — http://www.cs.ucsb.edu/~{}ckrintz/papers/osr.pdf.
[11]. Graphviz Documentation, http://www.graphviz.org/Documentation.php
[12]. Peng Zhao "Code and Data Outlining", 2005.
[13]. Mandel' I. D. Klasternyj analiz [Cluster Analysis]. — M.: Finansy i statistika, 1988. (in Russian)
[14]. Jun-Pyo Lee, Jae-Jin Kim, Soo-Mook Moon, Suhyun Kim "Aggressive Function Splitting for Partial Inlining", Seoul, South Korea, 2011.
[15]. David F. Bacon and Peter F. Sweeny, Fast Static Analysis of C++ Virtual Functuion Call, 1996
[16]. Clang: a C language family frontend for LLVM, http://clang.llvm.org/
[17]. GCC Free software foundation, http://gcc.gnu.org
[18]. ARM architecture., http://infocenter.arm.com
[19]. Youfeng Wu, James R. Larus. Static Branch Frequency and Program Profile Analysis. 27th International Symposium on Microarchitecture, 1994
[20]. Arutyun Avetisyan, Andrey Belevantsev, Alexey Borodin, Vladimir Nesov. Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode programm [Using static analysis for finding security vulnerabilities and critical errors in source code]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 21, pp. 23-38. (in Russian)
[21]. Kurmangaleev S.F. Korchagin V.P. Matevosyan H.A. Opisanie podkhoda k razrabotke obfustsiruyushhego kompilyatora. [Description of the approach to development of the obfuscating compiler]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol.23, pp. 67-76 (in Russian)
[22]. Kurmangaleev S.F. Korchagin V.P. Savchenko V.V. Sargsyan S.S. Postroenie obfustsiruyushhego kompilyatora na osnove infrastruktury LLVM. [Building obfuscation compiler based on LLVM infrastructure]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 23, pp. 77-92. (in Russian)
[23]. Roman Zhuykov, Dmitry Plotnikov, Mamikon Vardanyan. Avtomaticheskaya nastrojka optimizatsionnykh preobrazovanij kompilyatora GCC dlya platformy ARM. [Automatic tuning of GCC optimizations for ARM platform]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 49-66, (in Russian)
[24]. K. Pettis, R. C. Hansen. Profile guided code positioning . SIGPLAN Not. 1990 June.Vol. 25, no. 6, pp. 16-27. http://dx.doi.org/10.1145/93548.93550.
[25]. P. P. Chang, S. A. Mahlke, W.-m. W. Hwu. Using profile information to assist classic code optimizations. Center for Reliable and High-Performance Computing. Urbana-Champaign: University of Illinois, 1991.
[26]. Da Silva, A. F. Our Experiences with Optimizations in Sun's Java Just-In-Time Compilers / Anderson F. Da Silva, Vitor S. Costa // Journal of Universal Computer Science. — 2006. — Vol. 12.

[27]. M. Arnold, Stephen J. Fink, D. Grove, M. Hind, Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. IBM T. J. Watson Research Center, Hawthorne, USA, 2004.
[28]. Stephen Fink, David Grove, Michael Hind. Dynamic Compilation and Adaptive Optimization in Virtual Machines. IBM, June 2004.
[29]. Holzle, U. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming: Ph. D. thesis / Stanford University. —1994. — http://research.sun.com/self/papers/hoelzle-thesis.ps.gz.

О методах деобфускации программ

Ш.Ф. Курмангалеев, К.Ю. Долгорукова,
В.В. Савченко, А.Р. Нурмухаметов, Р.А. Матевосян, В.П. Корчагин
kursh@ispras.ru, unerkannt@ispras.ru
sinmipt@ispras.ru, oleshka@ispras.ru, hripsime@ispras.ru, korchagin@ispras.ru

Аннотация. Целью работы является разработка программного обеспечения, проводящего деобфусцирующие преобразования. Основная область применения – это анализ запутанного кода вредоносного программного обеспечения. Потребность в подобного рода продуктах возникла в связи с ростом популярности методик запутывания кода для сокрытия алгоритмов работы. Основным инструментом аналитика является дизассемблер, осуществляющий преобразование бинарного кода в читаемый человеком текст, но не проводящий его верификацию и упрощение. Ранее для «чистки» запутанного кода хватало удаления бесполезного кода по шаблонам, но применяемые методики запутывания усложняются, и для распутывания требуются средства, использующие более прогрессивные методы анализа и упрощения кода. В связи со схожестью проблем, стоящих перед оптимизирующим компилятором и деобфускатором, было опробовано использование компиляторной инфраструктуры LLVM в качестве ядра деобфускатора. Основным различием является то, что оптимизатор компилятора работает в условиях полного знания о программе, в то время как деобфускатор обладает неполной информацией, извлеченной непосредственно из участка анализируемого кода. По результатам испытаний, в связи с высокой зашумленностью выходного кода, было принято решение разработать свою инфраструктуру, которая позволяет добиться более чистого выходного кода. Тем не менее, применение LLVM или аналогичной разработки остается одним из перспективных направлений при разработке деобфусцирующего программного обеспечения.

Ключевые слова: анализ бинарного кода, деобфускация, обфускация, LLVM

1. Введение

Запутывание (обфускация) кода широко применяется при защите программного обеспечения от обратного проектирования, причем как

легального, так и вредоносного. Поскольку запутывание вызывает существенное замедление программы, очень часто запутывается только небольшая ее часть, содержащая важный алгоритм. Распутывание (деобфускация) применяется в области минимизации бинарного кода программ, что является важным для встраиваемых систем, так и для анализа вредоносного кода.

Современные вирусы наделены определенной способностью к маскировке выполняемых действий. Вирусы пытаются избежать обнаружения и воспрепятствовать анализу своего кода за счет применения обфусцирующих преобразований, включая изменение графа потока управления, эквивалентные замены команд, перестановки команд, изменение назначения адресов. Для создания алгоритма лечения вируса, требуется сначала детально разобрать алгоритм его работы. Без использования автоматических деобфускаторов скорость анализа весьма мала. С другой стороны, полностью автоматизировать этот процесс очень сложно, т.к. вредоносный код может иметь динамическое шифрование, различные варианты самомодификации. Как правило, аналитик вынужден работать с небольшим «окном» кода, на котором произведено запутывающее преобразование. Важной особенностью является то, что, как правило, одновременно применяется небольшое количество преобразований во избежание неожиданного и неправильного поведения кода. Создание эффективных средств лечения неизбежно требует изучения методов обфускации программ и разработки, специальных деобфусцирующих алгоритмов.

В разделе 2 описываются основные подходы к анализу программ. В разделе 3 описывается реализованный вариант деобфускатора, в разделе 4 описываются полученные экспериментальные результаты.

2. Алгоритмы распутывания

В данном разделе мы рассмотрим методы, которые применяются при анализе программ. Цель таких методов – выявление зависимостей между компонентами программы, что даёт возможность применить определённые оптимизирующие преобразования.

Методы анализа программ могут быть разделены на 4 группы [1]:

- Синтаксические**.** К этой группе относятся методы, основанные только на результатах лексического, синтаксического и статического семантического анализа программы.
- Статические [1][2]**.** К этой группе относятся методы анализа потоков управления и данных и методы, основанные на результатах анализа потоков управления и данных. Статические методы анализа работают

с программой, не используя информацию о работе программы на конкретных начальных данных.

- Динамические**.** Динамические методы анализа программ используют информацию, полученную в результате "наблюдения" за работой программы на конкретных входных данных.
- Статистические**.** Статистические методы используют информацию, собранную в результате значительного количества запусков программы на большом количестве наборов входных данных.

Важным этапом при проведении анализа кода является его нормализация. Нормализация кода – это процесс преобразования участка кода в каноническую форму, более пригодную для сравнения. Большинство из используемых преобразований обфускации ведет к увеличению размера кода. Иными словами, различные мутации фрагмента программы, могут рассматриваться как неоптимизированные версии своего прототипа, поскольку они содержат некоторые вычисления, присутствие которых имеет единственную цель – затруднение анализа. Нормализация кода направлена на устранение всех изменений, внесенных в ходе процесса обфускации.

Методы оптимизации, применяемые компиляторами, могут быть использованы для уменьшения количества «мусорного» кода. Обычно оптимизацию выполняет компилятор для уменьшения времени выполнения или для уменьшения размера кода или используемых данных.

Рассмотрим технику, предложенную в работе [5]: метапредставление инструкций. Все инструкции процессора можно разделить на три категории:

- условные и безусловные переходы;
- вызовы функций и возврат из функции;
- все остальные инструкции, оказывающие влияние на регистры, память и флаги управления.

Все инструкции третьей группы можно называть присваиваниями. Инструкции сравнения могут быть представлены в виде присваиваний, т.к. они обычно выполняют арифметическую операцию над своими операндами, а результат заносят в регистр управления (например, eflags для архитектуры IA-32). Для облегчения манипуляции объектным кодом будем использовать высокоуровневое представление, отражающее семантику машинных инструкций. В результате получаем так называемые «тройки» и «четверки», анализ и упрощение которых хорошо изучен в теории компиляторов. Стоит отметить, что даже простая инструкция декремента скрывает сложную семантику: аргумент уменьшается на единицу, и в зависимости от результата устанавливаются флаги.

Пример представлен в табл. 1.

Табл.1. Метапредставление инструкций

Машинная инструкция	Метапредставление
pop eax	r10 = [r11] r11 = r11 + 4
lea edi,[ebp]	r06 = r12
dec ebx	tmp = r08 r08 = r08 - 1 NF = r08@[31:31] ZF = [r08 = 0?1:0] CF = (˜(tmp@[31:31])

Поскольку большинство выражений содержит арифметические или логические операторы, иногда они могут быть упрощены в соответствии с обычными алгебраическими правилами [5]. Когда упрощение невозможно, переменные и константы могут быть переупорядочены, чтобы обеспечить возможность дальнейшего упрощения после этапа продвижения констант. В табл. 2 приведены примеры правил, которые могут быть использованы для выполнения упрощения и переупорядочивания (C обозначает константу, а t – переменную).

Табл.2. Примеры алгебраических правил

Оригинальное выражение	**Упрощенное выражение**
c1 + c2	Рассчитанная сумма констант
t1 − c1	−c1 + t1
t1 + c1	c1 + t1
0 + t1	t1
t1 + (t2 + t3)	(t1 + t2) + t3
(t1 + t2) * c1	(c1 * t1) + (c1 * t2)

3. Архитектура деобфускатора

Основной целью работы программы является получение листинга ассемблерного кода, по виду близкого к оригинальному (до обфускации) и пригодному для визуального анализа человеком. В процессе деобфускации происходят различные оптимизирующие преобразования, позволяющие добиться поставленной цели. Задача по своему характеру достаточна близка к задачам, встающим перед компиляторами на этапе оптимизации кода [3], и большинство подходов, выработанных при конструировании компиляторов, могут с успехом применяться в задачах деобфускации. В качестве вспомогательных библиотек для реализации используются boost [7][8] и graphviz [9].

Задачу деобфускации можно свести к построению оптимизирующего компилятора для языка ассемблера. Основное отличие от трансляторов ассемблера – наличие этапа оптимизации, характерного для высокоуровневых языков программирования. Архитектура системы будет иметь вид, представленный на рис. 1.

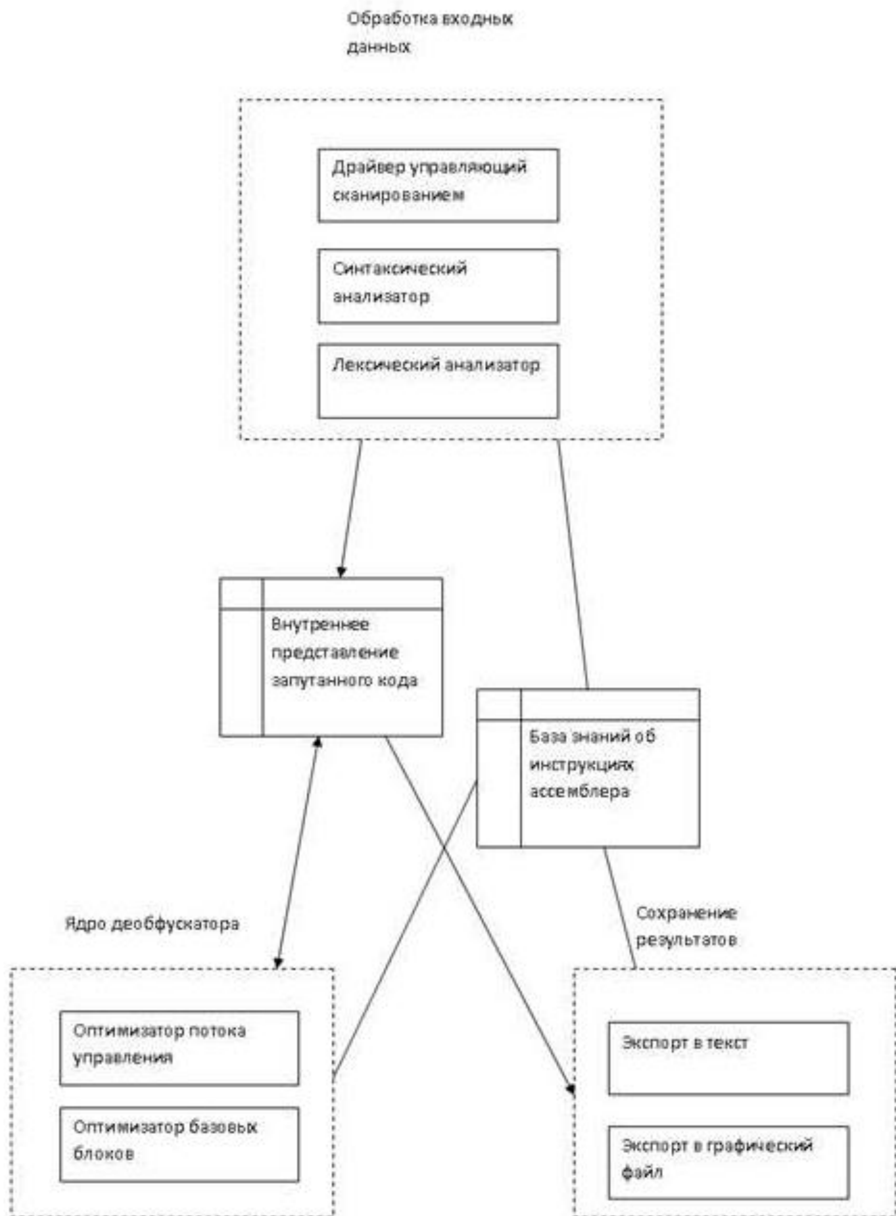


Рис.1. Архитектура системы

3.1 Используемые пакеты ПО

Graphviz — разработанный специалистами лаборатории AT&T пакет утилит для автоматической визуализации графов, заданных в виде описания на языке «dot».

Для построения лексического анализатора используется программа Flex, генерирующая лексический анализатор на основе описаний токенов, заданных с помощью регулярных выражений. В качестве генератора синтаксического анализатора используется Bison [6].

Low Level Virtual Machine (LLVM) [4] — инфраструктура для разработки компиляторов, содержащая:

- компилятор языка C++ в промежуточный байт-код, разработанный с целью обеспечения возможности оптимизации программы на всех этапах использования;
- набор RISC-подобных инструкций виртуального процессора, из которых строится байт-код;
- мощный оптимизатор байт-кода;
- окружение, предназначенное для исполнения байт-кода на различных платформах.

Первоначально в качестве ядра оптимизатора кода была использована LLVM. Но в связи с высоким уровнем «шума» в выходном коде, обусловленным потерями информации об именах регистров, было принято решение спроектировать свое ядро, проводящее оптимизации, специфические для обфусцированного кода.

3.2 Описание применяемых алгоритмов

3.2.1 Деобфускатор на базе LLVM

Для иллюстрации жизнеспособности гипотезы о применении техник оптимизации, используемых в компиляторах, был разработан транслятор некоторых машинных инструкций в язык промежуточного представления компиляторной инфраструктуры LLVM в форме SSA (форма с единственным присваиванием). Этот язык [4] компилируется в байт-код, который подвергается оптимизации с помощью набора инструментов, предоставляемого LLVM. В качестве теста использовался обфусцированный код выражений, приведенный в листинге 1.

```
EAX = EAX + 22;  ADD EAX, 22;
ECX = EDX + 1; MOV ECX, EDX; INC ECX;
```

Листинг 1. Код, использованный для обфускации

В качестве запутывающих преобразований использовались:

- внесение мертвого кода;
- внесение недостижимого кода, и, как следствие, изменение потока управления;
- внесение избыточного кода (избыточные вычисления).

После обфускации код примет вид, показанный в листинге 2.

```
ADD EAX,10
ADD EAX,10
INC EAX
MOV EAX,EAX
MOV ECX,EDX
MOV EDX,ECX
MOV EDX,ECX
ADD EAX,10
SUB EAX,15
INC EAX
INC EAX
NOP
NOP
NOP
NOP
INC EAX
INC EAX
INC EAX
JMP DEAD_CODE
NOP
NOP
NOP
NOP
MOV EAX,90909090
NOP
NOP
NOP
NOP
DEAD_CODE:
INC EAX
INC EAX
```

```
INC EAX
INC EAX
INC ECX
DEC EAX
DEC EAX
DEC EAX
```

Листинг 2. Обфусцированный код

Видно, что сложность восприятия кода заметно повысилась. После трансляции в форму SSA код примет вид, приведенный в листинге 3.

```
@eax = global i32 0;
@ebx = global i32 0;
@ecx = global i32 0;
@edx = global i32 0;
@esi = global i32 0;
@edi = global i32 0;
@eip = global i32 0;
@ebp = global i32 0;
@esp = global i32 0;
define void @main() nounwind {
 entry:
%0 = load i32* @eax
%1 = add i32 %0,10
store i32 %1, i32* @eax
%2 = load i32* @eax
%3= add i32 %2,10
store i32 %3, i32* @eax
%4 = load i32* @eax
%5 = add i32 %4,1
store i32  %5, i32* @eax
….
```

```
%38 = load i32* @eax
%39 = sub i32 %38,1
store i32  %39, i32* @eax
%40 = load i32* @eax
%41 = sub i32 %40,1
store i32  %41, i32* @eax
ret void }
```

Листинг 3. Код на промежуточном языке в форме SSA

После преобразования в форму SSA этот код передается на вход оптимизатора LLVM. В связи с особенностями архитектуры происходит потеря информации о регистрах, и единственный способ сохранить информацию – это использование имен регистров в качестве имен глобальных переменных, не подлежащих исключению. После проведения оптимизации получим код, продемонстрированный в листинге 4.

```
mov     EAX, DWORD PTR [_edx]
inc     EAX
mov     ECX, DWORD PTR [_eax]
mov     DWORD PTR [_ecx], EAX
add     ECX, 22
mov     DWORD PTR [_eax], ECX
```

Листинг 4. Код после проведения оптимизации с помощью LLVM

Код заметно упрощен, хотя и не соответствует в точности исходной форме, т.к. нет возможности повлиять на выбор регистров оптимизатором.

Поскольку для архитектуры Intel x86 существует возможность обращения к различным частям регистров, имеется регистр флагов, содержание которого может изменяться в результате исполнения команд, что приводит к дополнительному зашумлению листинга. Рассматривалась возможность внесения исправлений в архитектуру LLVM, но в связи с большим объемом работ и ограниченными сроками было принято решение отказаться от использования LLVM в качестве оптимизатора. Несмотря на указанные выше недостатки, использование LLVM в качестве ядра деобфускатора остается перспективным направлением.

3.2.2 Описание ядра деобфускатора

Для тестирования стойкости методов запутывания кода часто применяют алгоритмы оптимизации. Нами были реализованы следующие алгоритмы

оптимизации, позволяющие упрощать код, запутанный с использованием часто встречающихся методов:

- статическое построение и анализ потока управления;
- объединение базовых блоков;
- удаление недостижимого кода;
- продвижение констант;
- свертывание констант;
- удаление «мертвого» кода;
- алгебраическое упрощение, как часть алгоритма свертывания констант;
- статический слайсинг.

Дадим некоторые определения и опишем работу перечисленных алгоритмов.

Граф потока. Программа состоит из одного и более базовых блоков, являющихся узлами графа. Ребрами графа являются дуги переходов. Дуги переходов могут быть трех видов:

- нормальное движение по потоку управления Flow;
- переход по условию Cond;
- безусловный переход Uncond.

Статическое построение и анализ потока управления. Работа этого алгоритма начинается еще на стадии синтаксического анализа входного текста. Каждая распознанная инструкция передается в функцию, анализирующую переданную ей инструкцию и в зависимости от результата анализа либо добавляющую код в конец базового блока, либо создающую новый блок с достраиванием ребра типа Flow. После завершения этапа синтаксического анализадля всех блоков, на которые возможен переход, т.е. имеющих «метку», просматриваются все блоки графа и достраиваются ребра соответствующих переходов.

Объединение базовых блоков. Два следующих друг за другом базовых блока можно объединить, если они соединены ребром типа Flow или Uncond либо эквивалентом, состоящим из Flow и Cond, которые исходят из одной вершины. Обход вершин осуществляется обходом графа с помощью поиска в глубину. Обход проводится только из одной стартовой вершины. Для обхода графа используется модифицированная версия нерекурсивного алгоритма поиска в глубину с учетом приоритетов по типам ребер, где наибольший приоритет имеют ребра типа Flow и Uncond, а наименьший – Cond, и допускается удаление ребер из графа во время работы.

Удаление недостижимого код. Алгоритм основан на поиске в глубину. После завершения поиска из стартовой вершины все посещенные вершины имеют цвет, отличный от белого, и являются достижимыми из начала кода;

Вершины, цвет которых остался белым, могут быть удалены из графа, так как они не могут быть достигнуты из стартовой точки. Алгоритм состоит из двух частей – поиска в глубину и прохода по всем вершинам и удаления не достижимых.

Продвижение констант. Для всех инструкций в базовом блоке происходит сравнение с шаблоном mov reg, number. При совпадении значение регистра считается известным, и все его вхождения в качестве второго операнда заменяются на значение.

Свертывание констант. Для всех инструкций в базовом блоке происходит сравнение с шаблоном mov reg, number. При совпадении значение регистра считается известным. Далее продолжается просмотр инструкций, изменяющих значение регистра, и, если это возможно, происходит вычисление его нового значения. После вычисления нового значения происходит замена значения, инициализирующего регистр, и удаление проэмулированной инструкции. Поиск продолжается с инструкции, следующей за удаленной.

Удаление «мертвого» кода. Производится удаление бесполезных присваиваний для всех инструкций в базовом блоке. Выполняется сравнение с шаблоном mov reg, number. При совпадении поиск продолжается со следующей инструкции. Если встречается инструкция, перезаписывающая значение регистра без использования его предыдущего значения, то первоначально найденное присваивание можно удалить.

Алгебраическое упрощение, как часть алгоритма свертывания констант. Эта оптимизация заключается в попытке эмулировать выполнение арифметических инструкций, если известно значение операндов.

Статический слайсинг. Оптимизация удаляет конструкции вида push reg/pop reg, если значение регистра не используется для вычислений новых значений. Также удаляются инструкции, присваивающие новое значение этому регистру.

4. Тестирование

В качестве ключей оптимизации можно использовать следующие:

- «-JMPCLUE» – объединение базовых блоков;
- «-PROP» – распространение констант;
- «-FOLD» – свертывание констант;
- «-UNREACH» – удаление недостижимого кода;
- «-DEADELIM» – удаление мертвого кода
- «-PUSHPOP» – удаление конструкций вида push reg/pop reg и вычислений, результаты которых не используются

Запутанный код	Распутанный код

label_1:
push eax
mov eax,0ffaah
xor ecx,eax
mov ebx,15h
pop eax
mov ebx,eax
jmp loc_1
dec ecx
loc_1:
inc ebx
jmp loc_2
loc_2:
jmp loc_3
loc_3:
mov eax,12ffh
jmp loc_4
loc_4:
jnz label_2
label_2:
jz label_77
label_77:
jnz label_3
xor eax,eax
jmp label_4
jnz label_5
label_5:
inc ecx
label_3:
jmp label_1
label_4:
jns label_2
label_99:
jz label_99
	label_1:
	xor ecx,eax
	mov ebx,eax
	inc ebx
	mov eax,12ffh
	label_2:
	jnz label_1
	xor eax,eax
	jns label_2
	label_99:
	jz label_99

Табл.3. Результаты работы деобфускатора

Результаты работы деобфускатора и граф потока управления для запутанного и распутанного кода приведены в табл. 3 и на рис. 2 соответственно. В таблицах наглядно видно результаты работы алгоритмов, связанных с

анализом потока управления. Убраны все недостижимые из начальной точки вершины, проведено объединение базовых блоков, выполнен анализ условий переходов и убраны транзитные переходы. Кроме того, к базовым блокам были применены такие оптимизации, как слайсинг, свертывание и распространение констант.

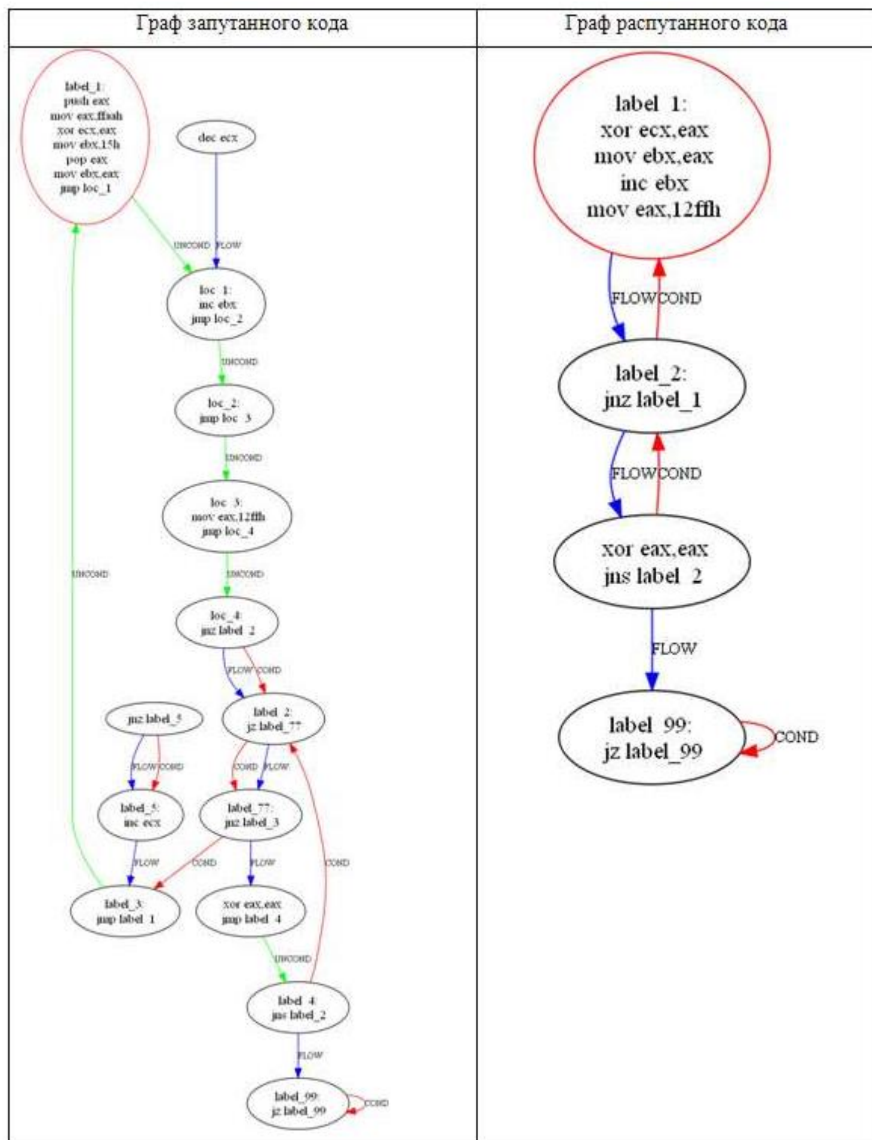


Рис. 2. Граф потока управления до и после оптимизации

В табл.4 приведены результаты последовательной оптимизации базового блока.

Табл. 4 - Результаты последовательных оптимизаций

Запутанный код	Слайсинг	Удаление мертвого кода	Распространение и свертывание констант

mov eax,10h
mov ecx,150h
mov ecx,0ffffh
mov ebx,90h
xor ebx,21h
add eax,11h
xor ecx,0ffh
mov ebx,eax
push eax
sub eax,20h
xor eax,777h
add eax,0codeh
pop eax
sub eax,5
mov ecx,eax
	mov eax,10h
	mov ecx,150h
	mov ecx,ffffh
	mov ebx,90h
	xor ebx,21h
	add eax,11h
	xor ecx,ffh
	mov ebx,eax
	sub eax,5
	mov ecx,eax
		mov eax,10h
		mov ecx,ffffh
		mov ebx,90h
		xor ebx,21h
		add eax,11h
		xor ecx,ffh
		mov ebx,eax
		sub eax,5
		mov ecx,eax
			mov eax,1ch
			mov ecx,1ch
			mov ebx,21h

Видно, что применение методик деобфускации дает существенное улучшение читаемости кода. Для лучшего эффекта распространение и свертывание констант применяются в одном проходе как взаимодополняющие оптимизации.

5. Заключение

В рамках данной работы разработан прототип деобфускатора программного кода, представляющий собой программное решение для оптимизации кода ассемблера процессоров intel x86.

Были реализованы следующие методики распутывающих преобразований:

- статическое построение и анализ потока управления;
- объединение базовых блоков;
- удаление недостижимого кода;
- продвижение констант;
- свертывание констант;
- удаление «мертвого» кода;
- алгебраическое упрощение, как часть алгоритма свертывания констант.

В результате это обеспечило качественную деобфускацию программного кода. Кроме того, созданная архитектура программного обеспечения позволяет легко добавлять новые алгоритмы деобфускации, т.к. предоставляет удобный доступ к данным и функции для работы с ними. Рассмотрено перспективное направление использования компиляторной инфраструктуры LLVM в качестве ядра деобфускатора, определены основные сложности такого подхода.

Список литературы

[1]. Анализ запутывающих преобразований программ. Чернов А. В., Труды Института Системного программирования РАН [электронный ресурс] http://www.citforum.ru/security/articles/analysis/, свободный.-Загл с экрана.

[2]. Reverse Compilation Techniques By Cristina Cifuentes [электронный ресурс] http://www.itee.uq.edu.au/~cristina/dcc/decompilation_thesis.ps.gz, свободный.-Загл с экрана.

[3]. Ахо А., Лам М., Сети Р., Ульман Д.Компиляторы. Принципы, технологии, инструментарий, 2-изд.: Пер с англ.- М.: ООО «И.Д. Вильямс», 2008. – 1184с. : ил.

[4]. LLVM Language Reference Manual [электронный ресурс] http://LLVM.org/docs/LangRef.html, свободный.-Загл с экрана.

[5]. Using Code Normalization for Fighting Self-Mutating Malware Danilo Bruschi, Lorenzo Martignoni, Mattia Monga [электронный ресурс] http://idea.sec.dico.unimi.it/~lorenzo/rt0806.pdf, свободный.-Загл с экрана.

[6]. John R Flex & bison. 1st edition, 304p. Levine Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. ISBN: 978-0-596-15597-1

[7]. Библиотека BOOST C++ [электронный ресурс] http://www.solarix.ru/for_developers/cpp/boost/boost-library.shtml, свободный.-Загл с экрана.

[8]. Сик, Ли, Ламсдэйн, C++ Boost Graph Library. Библиотека программиста / Пер. с английского Сузи Р. – СПб.: Питер,2006.- 304с. ил. ISBN 5-469-00352-3.

[9]. Using Graphviz as a library [электронный ресурс] http://www.graphviz.org/pdf/libguide.pdf, свободный.-Загл с экрана.

Software deobfuscation methods: analysis and implementation

Sh.F. Kurmangaleev, K.Y. Dolgorukova,
V.V. Savchenko, A.R. Nurmukhametov, H. A Matevosyan, V.P. Korchagin
kursh@ispras.ru, unerkannt@ispras.ru
sinmipt@ispras.ru, oleshka@ispras.ru, hripsime@ispras.ru, korchagin@ispras.ru
ISP RAS, Moscow, Russia

Annotation. This paper describes the work on development of the deobfuscation software.

The main target of the developed software is the analysis of the obfuscated malware code. The need of this analysis comes from the obfuscation techniques being widely used for protecting implementations. The regular disassembly tool mostly used by an analyst transforms a binary code in a human-readable form but doesn't simplify the result or verify its correctness. Earlier for this task it was enough to apply pattern-matching cleanup of the inserted useless garbage code, but nowadays obfuscation techniques are getting more complicated thus requiring more complex methods of code analysis and simplification.

As deobfuscation methods require analysis and transformation algorithms similar to those of an optimizing compiler, we have evaluated using LLVM compiler infrastructure as a basis for deobfuscation software. The difference from the compiler is that the deobfuscation algorithms do not have the full information about the program being analyzed, but rather a small part of it. The evaluation results show that using LLVM directly does not remove all the artifacts from the obfuscated code, so to provide the cleaner output it is desirable to develop an independent tool. Nevertheless, using LLVM or similar compiler infrastructure is the feasible approach for developing deobfuscation software.

Keywords: binary code analysis, obfuscation, deobfuscation, LLVM

References

[1]. A.V. Chernov. Analiz zaputyvayushhikh preobrazovanij programm. [Analysis obfuscating program transformations] Trudy ISP RAN [The Proceedings of ISP RAS], 2002, vol.3, pp. 7-38 (in Russian).
[2]. Reverse Compilation Techniques By Cristina Cifuentes http://www.itee.uq.edu.au/~cristina/dcc/decompilation_thesis.ps.gz
[3]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN:0321486811
[4]. LLVM Language Reference Manual http://LLVM.org/docs/LangRef.html
[5]. Using Code Normalization for Fighting Self-Mutating Malware Danilo Bruschi, Lorenzo Martignoni, Mattia Monga http://idea.sec.dico.unimi.it/~lorenzo/rt0806.pdf
[6]. John R Flex & bison. 1st edition, 304p. Levine Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. ISBN: 978-0-596-15597-1
[7]. BOOST C++ http://www.solarix.ru/for_developers/cpp/boost/boost-library.shtmlc

[8]. Jeremy G. Siek; Lie-Quan Lee; Andrew Lumsdaine. The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©2002 ISBN:0-201-72914-8
[9]. Using Graphviz as a library http://www.graphviz.org/pdf/libguide.pdf

Вывод типов для языка Python

Бронштейн И. Е.
ibronstein@ispras.ru

Аннотация. Тема статьи — вывод типов для программного кода на языке Python. Сначала производится обзор описанных в научной литературе алгоритмов вывода типов для языков с параметрическим полиморфизмом. Затем даётся описание нового алгоритма, являющегося модификацией одного из предыдущих: алгоритма декартова произведения. Показывается, как модуль вывода типов, использующий новый алгоритм, анализирует различные конструкции языка Python. Представляются результаты работы над прототипом.

Ключевые слова: python; вывод типов; динамическая типизация данных; статический анализ; обнаружение дефектов.

1. Введение

В современном индустриальном программировании остро стоит проблема надёжности разрабатываемого программного обеспечения. Поэтому существует множество программных средств для обнаружения тех или иных дефектов в анализируемом коде. Анализ, осуществляемый по исходному коду без запуска соответствующей программы, называется *статическим анализом* программного обеспечения. Исследования на конкретных программных продуктах показывают, что сложность исправления ошибки, найденной при помощи статического анализа, ниже, чем если бы эта ошибка была найдена на более поздних этапах разработки [1].

Статические анализаторы существуют в основном для статически типизированных языков программирования (например, Си, Си++ и Java). Однако в настоящее время всё большую популярность завоёвывают языки с динамической типизацией данных. Согласно авторитетному рейтингу от компании TIOBE Software [2], в десятку самых популярных языков программирования входят Objective-C (смешанная типизация) и динамически типизированные PHP, Python, Perl и Ruby (JavaScript — 11-й). Благодаря тому, что динамически

типизированные языки гибки и способствуют быстрой разработке, их всё чаще предпочитают для создания даже весьма крупных программных систем. В то же время понятно, что отсутствие у компилятора информации о типах выражений может приводить к ошибкам времени исполнения, которые для языков со статической типизацией могут быть обнаружены уже на этапе компиляции. Так, для следующего примера кода на динамически типизированном языке Python отсутствие в классе A метода not_present_method будет обнаружено лишь при попытке вызвать метод с таким именем:

```
class A:

    pass

a = A()

a.not_present_method()
```

Чтобы понять, какие виды дефектов — ошибок и уязвимостей — распространены в программном коде на языке Python, мы проанализировали более 150 отчётов, хранящихся в системах отслеживания ошибок (bug trackers) проектов с открытым исходным кодом. В качестве таких проектов были выбраны крупные системы с сотнями тысяч строк кода: Django, Gramps и Twisted.

Подавляющее большинство отчётов об ошибках в таких системах пришлось исключить из рассмотрения, поскольку речь в них идёт о логических ошибках, а не о дефектах, которые возможно найти при помощи статического анализа. Общими логическими ошибками являются, например, несоответствия между документацией определённого программного модуля и его реальной функциональностью. Примерами частных логических ошибок могут служить неправильные регулярные выражения для проверки пользовательских данных или нуждающийся в корректировке текстовый вывод программы.

Всё остальное, т. е. найденные на реальных проектах дефекты, можно разделить на две основные категории:

- «простые» дефекты — те дефекты, которые можно обнаружить без вывода типов;
- «сложные» дефекты — те дефекты, найти которые можно, лишь используя информацию, полученную при выводе типов.

В табл. 1 приведены данные об общем количестве проанализированных отчётов об ошибках и числе найденных, соответственно, «простых» и «сложных» дефектов на выбранных проектах.

Табл. 1. Данные о найденных дефектах

Название проекта	**Все отчёты об ошибках**	**«Простые» дефекты**	**«Сложные» дефекты**
Django	**135**	3	7
Gramps	**3**	0	1
Twisted	**16**	0	2

Как видно из таблицы, для языка Python большинство ошибок, которые теоретически могут быть обнаружены с помощью статического анализа, составляют именно ошибки несоответствия типов. Статический анализатор сможет находить такие ошибки, только если в нём имеется реализованный модуль вывода типов. Однако статических анализаторов для динамически типизированных языков, где присутствовал бы нетривиальный вывод типов, нам найти не удалось. Между тем, анализ одного из отчётов об ошибке, найденного в системе отслеживания ошибок проекта Gramps [3], показал: модуль вывода типов должен поддерживать большинство основных конструкций языка Python, чтобы на основе такого вывода типов можно было обнаружить дефект, о котором идёт речь в отчёте: дефект несоответствия типов. Должны поддерживаться как минимум следующие элементы языка Python: стандартные структуры данных; основные управляющие конструкции; импорт модулей; классы/объекты и их методы; некоторые встроенные функции для стандартных типов; некоторые бинарные операции; атрибуты объектов; функции, используемые как параметры других функций; исполнение программного кода из файлов; анонимные (лямбда-) функции.

Целью данной работы является реализация вывода типов для программного кода на языке Python. Под выводом типов подразумевается модуль, в результате работы которого каждому выражению в программном коде на языке Python будет поставлено в соответствие множество возможных типов для этого выражения. При этом, чтобы продемонстрировать работоспособность полученной реализации, планируется использовать вышеупомянутый пример несоответствия типов из проекта Gramps.

2. Обзор существующих решений

В [4] даётся обзор алгоритмов вывода типов для программного кода, в котором присутствует *параметрический полиморфизм*. Под параметрическим полиморфизмом понимается способность функции или метода работать со значениями различных типов (например, функция len в Python может принимать на вход различные коллекции и возвращать их размер). В работе показано развитие алгоритмов вывода

типов (на примере динамически типизированного языка программирования Self) от самого простого алгоритма (т. н. «базового» алгоритма — basic algorithm) до алгоритма декартова произведения (АДП). Мы рассмотрим достаточно подробно только «базовый алгоритм» и АДП.

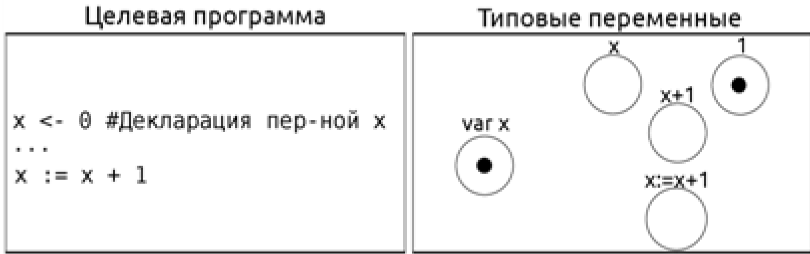


Рис. 1. Фрагмент анализируемой программы на языке Self (слева) и созданные для него типовые переменные (справа). Типовая переменная для декларации x помечена как var x, чтобы отличать её от выражения x (операнда сложения).

«Базовый алгоритм», или алгоритм Палсберга — Шварцбаха был впервые представлен в [5] как решение некоторой системы ограничений, накладываемых на типы выражений в целевой программе. В [4] алгоритм рассматривается в несколько другой, операционной форме: как последовательность шагов, в ходе выполнения которых осуществляется анализ одновременно потока выполнения и потока данных программы. Такой подход позволяет провести прямое соответствие между исполнением программы и её анализом (выводом типов), что упрощает описание алгоритма. Поэтому мы также будем придерживаться операционного подхода. Согласно нему «базовый алгоритм» состоит из трёх шагов:

1. **Создание типовых переменных**. Первый шаг заключается в том, что всем переменным и выражениям в целевой программе ставятся в соответствие *типовые переменные* (см. рис. 1), т. е. переменные, значениями которых являются множества типов. Сначала типовые переменные хранят в себе пустые множества типов, а на следующих двух шагах в них добавляются типы, соответствующие переменным и выражениям, с которыми переменные связаны. Типы из типовых переменных не удаляются.

2. **Инициализация типовых переменных**. На втором шаге в типовые переменные добавляются определённые типы — переменные инициализируются. Цель — зафиксировать первоначальное состояние целевой программы, «засевая» типовые переменные, относящиеся к выражениям или переменным, где некоторые объекты можно найти с самого начала. Так, если в коде программы объявлена переменная, в соответствующую типовую переменную добавляется тип инициализатора. Например, для декларации x <- 0 переменная var x получает значение {*int*}.

Аналогично для строкового литерала ’abc’ в его типовую переменную добавляется тип str. На рис. 2 тип int, добавленный в типовые переменные var x и 1, показан в виде чёрных кружков.

3. **Установка ограничений и пропагация типов**. На последнем шаге строится ориентированный граф, вершинами которого являются типовые переменные. Вершины добавляются поочерёдно и отражают ограничения на типы. Ограничение фактически представляет собой аналог потока данных, только не во время исполнения программы, а на этапе её анализа (вывода типов). Так, если в программе выполняется присваивание x := y, то поток данных направлен из y в x. На интуитивном уровне это означает, что любой тип, который может принимать выражение y, может также иметь и переменная x. Поэтому, когда модуль вывода типов доходит до этого присваивания, он добавляет ребро от типовой переменной для y к типовой переменной для x, отражая тот факт, что type(y) ⊆ type(x). Это иллюстрирует рис. 2.

Рис. 2. Ситуация до и после добавления ребра, соответствующего ограничению для x := y, а также пропагация типов.

Всякий раз, когда в граф добавляется ребро, типы пропагируются (распространяются) вдоль него. На рис. 2 показано, как типы из узла y распространяются по ребру и копируются в узел x. По мере того, как в графе появляются новые рёбра, соответствующие ограничениям, типы, которые изначально были лишь в инициализированных типовых переменных, пропагируются всё дальше и дальше. Пропагация устроена таким образом, что ограничения на типы, такие, как type(y) ⊆ type(x), сохраняются после каждого её этапа: если в type(y) добавляются новые типы, они сразу же пропагируются в type(x), и таким образом отношение «является подмножеством» продолжает выполняться.

Добавление новых ограничений создаёт необходимость в дополнительной пропагации. Верно и обратное: например, для вызова некоторого метода в результате пропагации может быть вычислен новый тип объекта. Значит, возможно, будет необходимо анализировать тела методов в этом объекте, а при анализе тел методов могут быть созданы новые рёбра в графе (ограничения) и т. д. Поэтому на третьем шаге циклически создаются новые ограничения и пропагируются типы до тех пор, пока эти действия вносят какие-либо изменения в граф типовых переменных.

Мы рассмотрели только случай присваиваний, но ограничения на типы должны генерироваться и для других языковых конструкций, например:

- для выражений вида var, где происходит чтение значения некоторой переменной, генерируется ребро от типовой переменной, соответствующей var, к типовой переменной, соответствующей выражению;
- для вызовов функций, благодаря использованию специальных *шаблонов для функций*, генерируются рёбра от аргументов вызова к соответствующим параметрам функции, а от операторов return в вызываемых функциях генерируются рёбра к вызовам.

Последняя фраза нуждается в пояснениях. Шаблоном для функции F называется подграф графа типовых переменных, состоящий из:

- вершин (типовых переменных), соответствующих выражениям, локальным переменным и параметрам F;
- рёбер (ограничений), инцидентных этим вершинам.

Пусть имеется функция max:

```
def max(x, y):
    if x > y:
        return x
    else:
        return y
```

Шаблон для этой функции изображён на рис. 3. На нём также показано, какие рёбра к шаблону создаются при вызове max(3, 4). Для типовой переменной «вызов функции» её значение вычисляется путём пропагации типов аргументов вызова по шаблону для вызываемой функции. Если вызываться может несколько функций, итоговый тип — это объединение типов возвращаемых значений для этих функций. Следовательно, для типовой переменной max(3, 4) будет вычислен тип $\{int\}$.

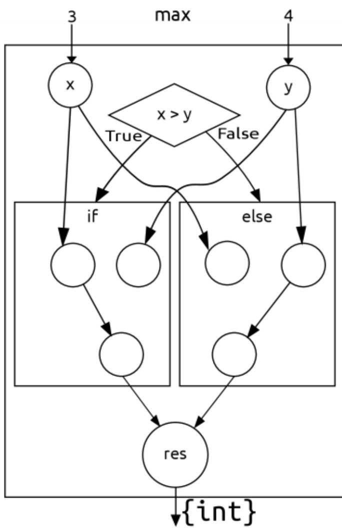


Рис. 3. Шаблон для функции max: сама функция и аргументы её вызова. Тип вызова (тип возвращаемых значений для вызываемых функций) выводится путём пропагации типов аргументов вызова по шаблону.

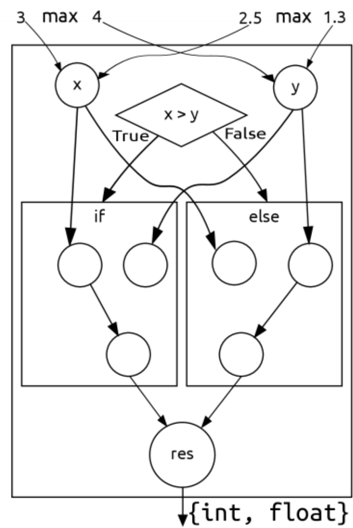


Рис. 4. «Базовый алгоритм» выводит неточный тип {int, float} для двух вызовов max, хотя один из вызовов считает максимум двух целых чисел, а второй — двух вещественных

Проблема «базового алгоритма» в том, как он работает для полиморфных функций, например, для max. На рис. 4 показано, что произойдёт, если в программе присутствует два вызова max: один с целочисленными, а второй с вещественными аргументами. Для обоих вызовов вычисляется тип $\{int, float\}$. Это неточно, так как очевидно, что вызов max от целых чисел не может вернуть вещественное число и наоборот. Такая неточность вызвана тем, что разные типы — int и float — сливаются на входе в шаблон для функции max. Такое слияние вызывает потерю информации: «склеенные» типы сложно разделить обратно, и в результате получаются неточные типы для всех вызовов одной и той же функции.

Из-за того, что «базовый алгоритм» выводит неточные типы, если в коде присутствует параметрический полиморфизм, Палсберг и Шварцбах представили несколько идей по улучшению алгоритма. Например, было предложено создавать отдельные шаблоны для каждого вызова определённой функции, например, max. Однако, это показало себя крайне неэффективным с точки зрения производительности.

Вместо внесения незначительных улучшений в «базовый алгоритм» в [4] авторы представляют новый алгоритм вывода типов: алгоритм декартова произведения (АДП), который не сливает типы при вызовах функции с параметрами разных типов и, в то же время, не страдает от низкой эффективности.

При анализе вызовов функций АДП работает с *мономорфными*, а не с *полиморфными* типами аргументов вызова. Под мономорфным типом понимается конкретный тип, с которым связано выражение по ходу выполнения программы. Примеры мономорфных типов: NoneType, int, класс A. Полиморфный тип для переменной или выражения определяется как совокупность всевозможных мономорфных типов для неё (него). Например, для следующего кода переменная x будет иметь полиморфный тип, равный $\{NoneType, int, A\}$:

```
x = None
...
x = 1
...
x = A()
```

Ключевую идею АДП можно проследить, если вернуться к аналогии между исполнением программы и её анализом. Во время исполнения программы все вызовы функций на самом деле осуществляются «мономорфно». Поясним, что имеется в виду. Представим себе функцию square, которая в программе принимает на вход аргумент a, имеющий полиморфный тип $\{int, float\}$.

```
def square(x):
    return x * x
a = 1
...
a = 3.14
...
b = square(a)
```

Очевидно, что во время каждого конкретного вызова функции square аргумент вызова может иметь либо тип int, либо тип float, но не оба сразу. Это наблюдение можно сформулировать следующим образом:

полиморфных вызовов функций не существует, о полиморфизме можно говорить лишь для выражений «вызов функции» в коде программы.

В АДП, в отличие от предшествующих алгоритмов, создаются лишь *мономорфные шаблоны*. Например, пусть имеется вызов max(x, y). Обозначим полиморфный тип аргумента x как X, тип аргумента y как Y. Предположим, мы каким-либо образом вычислили, что $X = \{x_1, x_2, \dots, x_s\}$, а $Y = \{y_1, y_2, \dots, y_t\}$. Чтобы вычислить тип для вызова max, АДП вычисляет декартово произведение типов аргументов вызова. В данном случае это множество пар: $X \times Y = \{(x_1, y_1), \dots, (x_1, y_t), \dots, (x_i, y_j), \dots, (x_s, y_1), \dots, (x_s, y_t)\}$. В общем случае это множество кортежей по n элементов в каждом, где n — число аргументов вызова.

Затем АДП для каждого $(x_i, y_j) \in X \times Y$ пропагирует типы аргументов по отдельному мономорфному шаблону — шаблону для этого кортежа. Если такой шаблон уже создавался ранее, он повторно используется (из него берутся возвращаемые типы), в противном случае создаётся новый шаблон, который в дальнейшем — для других вызовов max с аргументами (x_i, y_j) — будет доступен для повторного использования. Значение типовой переменной, связанной с вызовом функции, вычисляется как объединение множеств типов, возвращаемых в каждом используемом шаблоне. Ситуация проиллюстрирована на рис. 5.

В действительности при анализе некоторого вызова функции мы можем не знать точного полиморфного типа для каждого аргумента. Рассмотрим следующий код:

```
z = max(x, y)

...

y = 'abc'
```

В приведённом выше примере вызов функции max анализируется до того, как в множество возможных типов для y добавляется тип str. Это не является проблемой для АДП: декартово произведение вычисляется для **известных на текущий момент** типов аргументов вызова. Как только становятся известными новые возможные типы для аргументов, процедура вычисления повторяется заново. Декартово произведение — монотонная функция, поэтому при расширении типов аргументов «новое» декартово произведение включает в себя «старое» (см. рис. 6).

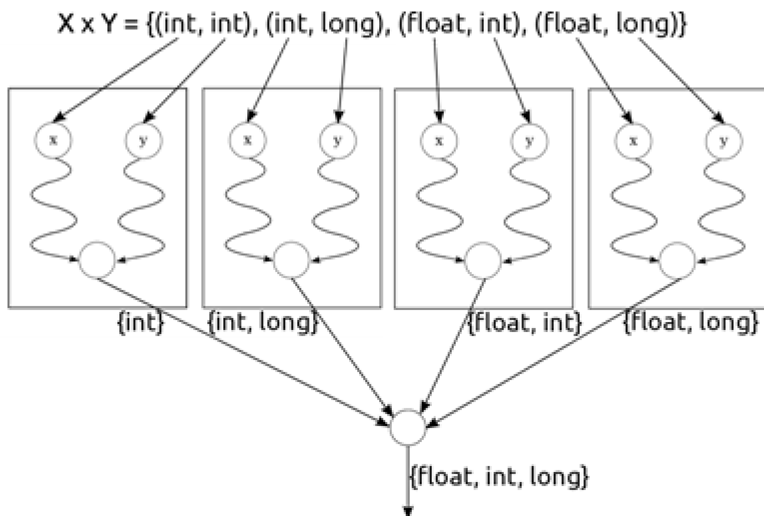


Рис. 5. АДП для вызова max(x, y), где X = type(x) ={int, float}, Y = type(y) = {int, long}.

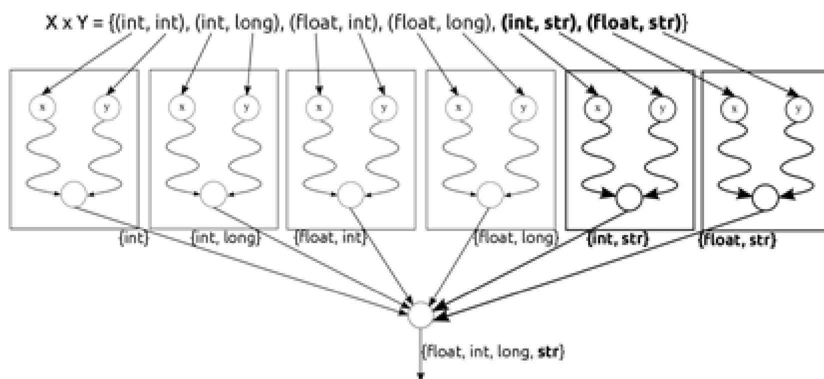


Рис. 6. АДП для вызова max(x, y), где X = type(x)= {int, float}, Y = type(y) = {int, long, str}. Здесь показано, что происходит при расширении типа аргумента вызова. Жирным шрифтом выделены «новые» структуры, создаваемые во время повторного вычисления декартова произведения.

Следовательно, мономорфные шаблоны не создаются напрасно. Объединение множеств также является монотонной функцией, поэтому тип вызова функции может только увеличиваться и т. д. Это гарантирует монотонность значения типовых переменных, т. е. типовые переменные всегда содержат правильные (но не обязательно все) возможные типы для определённого выражения.

В заключительной части описания АДП в [4] приводится оценка этого алгоритма:

- алгоритм является *точным* в том смысле, что он может анализировать цепочки вызовов полиморфных функций без потери точности;
- алгоритм является *эффективным*, потому что позволяет избежать избыточного анализа, так как вызов функции с аргументами мономорфных типов будет анализироваться ровно один раз.

АДП разрабатывался специально для языка программирования Self, который существенно отличается от языка Python, и непосредственное применение АДП к коду на Python не представляется возможным. В [6] приводится описание модифицированного АДП, который возможно использовать для анализа Python-программ. В частности, в новом алгоритме даются попытки решения проблем, с которыми АДП (для языка Self) не справляется либо справляется плохо. Вот эти проблемы:

- Алгоритм АДП хорошо справляется с полиморфическим полиморфизмом (см. выше), но плохо работает для *полиморфизма по данным*. Полиморфизм по данным — возможность хранения объектов разных типов в одной и той же «ячейке» данных. Под «ячейкой» понимается, например, элемент стандартной структуры «список» в языке Python. В списке [1, 3.14, 'abc'] разные «ячейки» хранят объекты, имеющие, соответственно, тип int, float и str.
- В АДП не описано, как обрабатывать внешний по отношению к анализируемой программе код. Например, метод append для CPython — стандартной реализации Python — написан на языке Си, и нужно иметь возможность анализировать такой метод с точки зрения вывода типов.
- АДП (в том виде, в котором он описан в [4]) зацикливается при наличии рекурсивных функций.

Несмотря на свои преимущества, алгоритм из [6] до настоящего момента реализован не был. Он служит основой для алгоритма вывода типов, реализованного в текущей работе. Однако, алгоритмы всё же существенно различаются, так как в исходный алгоритм было внесено значительное число изменений с целью решения практической задачи, сформулированной в начале работы: нахождения несоответствия типов в примере из проекта Gramps.

3. Описание реализации

Модуль вывода типов в настоящее время представляет собой отдельную программу Tirpan (**T**ype **i**nfe**r**ence-based **p**ython **an**alyzer), написанную на языке Python. Tirpan запускается на некотором Python-файле и по завершении своей работы сопоставляет каждое выражение в программном коде (исходного файла, а также транзитивно импортируемых из него файлов) с множеством возможных типов для

этого выражения. Мы исходим из естественного для статического анализа предположения, что программный текст всех файлов, нуждающихся в анализе, доступен. В остальном алгоритм вывода типов не накладывает никаких ограничений на анализируемые файлы. В этом отличие текущего алгоритма от исходного алгоритма. Исходный алгоритм, описанный в [6], предполагалось использовать для трансляции анализируемого файла на языке Python в код на Си, что требовало абсолютной точности вывода типов. Последнее требование, в свою очередь, не допускало наличия в коде динамических языковых конструкций вроде eval, exec и т. д. Для статического анализа стопроцентная точность вывода типов не требуется, поэтому подобные ограничения и отсутствуют.

При рассмотрении нашего алгоритма вывода типов мы будем придерживаться операционного подхода так же, как это делают авторы в [4, 6]. Это означает, что мы будем описывать алгоритм путём перечисления шагов, которые выполняются при анализе тех или иных элементов языка Python, а также структур данных, создаваемых во время выполнения этих шагов. Описание того, как происходит анализ различных конструкций Python, будет производиться от простых конструкций (например, стандартных структур, таких, как списки) к более сложным (например, классам и объектам). Кроме того, мы осветим процесс анализа внешнего по отношению к программе на языке Python кода.

3.1 Чувствительность к потоку выполнения

Текущий алгоритм, как и его предшественник, является нечувствительным к потоку выполнения (flow-insensitive). Это означает, что он добавляет возможные типы к переменным, опираясь только на исходный код анализируемой программы. Чувствительные к потоку выполнения (flow-sensitive) алгоритмы, напротив, добавляют возможные типы к конкретным *местам доступа к переменным*. Нечувствительные к потоку выполнения алгоритмы являются гораздо более простыми, чем чувствительные. В то же время первые теоретически являются менее точными, чем вторые. Действительно, рассмотрим следующий код:

```
def foo():
    if ...:
        x = 1
        return (x, square(x))
    else:
```

```
        x = 3.14
        return (x, square(x))
```

Переменной x в одной ветке условного оператора присваивается значение типа int, а в другой — типа float. Соответственно, функция foo вернёт либо пару значений типа int (из ветки if), либо пару значений типа float (из ветки else). Однако, текущий алгоритм, являясь нечувствительным к потоку выполнения, «склеит» два присваивания разных значений переменной x. В результате получится, что x имеет полиморфный тип {*int*, *float*}, а foo возвращает полиморфный тип, включающий в себя и {*tuple*(*int*, *float*)}, и {*tuple*(*float*, *int*)}, что неправильно.

Чувствительный к потоку выполнения алгоритм смог бы разделить два присваивания переменной x. Вообще говоря, текущий алгоритм можно было бы сделать чувствительным, если бы мы решили предварительно преобразовывать анализируемый исходный код в т. н. SSA-представление (static single assignment form) [7]. В этом представлении переменные переименовываются таким образом, чтобы чтение некоторой переменной соответствовало записи в неё. В таком случае в выше приведённом примере переменная x в одной ветке была бы переименована в x1, а в другой — в x2, и коллизии бы не произошло. Однако и этот, и некоторые другие методы всё равно помогают лишь в простых случаях, когда одной переменной в одной и той же области видимости присваиваются разные значения. В более сложных случаях не спасают и они. Рассмотрим следующий пример кода на языке Python:

```
def foo(x):
    if isinstance(x, int):
        return x
    else:
        return None
y = foo(1)
```

Чувствительный к потоку выполнения алгоритм определил бы, что тип выражения foo(1) (и, следовательно, переменной y) равен {*int*}. Поскольку нечувствительный к потоку выполнения алгоритм не учитывает условия в if, он выведет для y неточный тип, равный {*int*, *NoneType*}.

В то же время следует отметить, что присваивать одной переменной в одной области видимости значения разных типов — плохая практика. Таким образом, получается, что сильно усложнять алгоритм пришлось

бы ради лучшей поддержки кода не очень высокого качества. Поэтому было решено отказаться от такого усложнения и оставить алгоритм нечувствительным к потоку выполнения программы.

3.2 Общая схема работы программы

Общая схема работы программы изображена на рис. 7. Программе в виде аргументов командной строки передаётся путь к файлу, который необходимо проанализировать, а также, возможно, некоторые флаги. В модуле Tirpan происходит обработка этих флагов. Если всё прошло успешно, в модуль Importer передаётся команда «импорт главного модуля (модуля __main__)» с указанием адреса исходного файла. Importer считывает этот файл и передаёт его содержимое в модуль Parser, где по исходному коду строится дерево абстрактного синтаксиса (abstract syntax tree, AST). Для этого используются стандартные средства языка Python (модуль ast, включающий в себя функцию ast.parse) [8]. Сгенерированный AST пересылается обратно в Importer. Внутри себя Importer содержит *таблицу модулей*, каждая запись в которой содержит сведения об отдельном модуле: информацию о пути, по которому был найден импортированный файл, ссылку на AST для этого файла и некоторые другие признаки. Таблица модулей заполняется первой записью для главного модуля, после чего AST передаётся в модуль TIVisitor (type inference visitor). TIVisitor фактически представляет собой обходчик синтаксического дерева, в котором на определённые узлы дерева «навешивается» дополнительная информация о возможных типах для этих узлов. Понятно, что в процессе обхода дерева могут встретиться import-директивы. В таком случае модулю Importer пересылается команда «импорт модуля x», и (если модуль не был импортирован ранее) вновь выполняется процедура генерации синтаксического дерева, создания записи в таблице модулей и обхода полученного дерева.

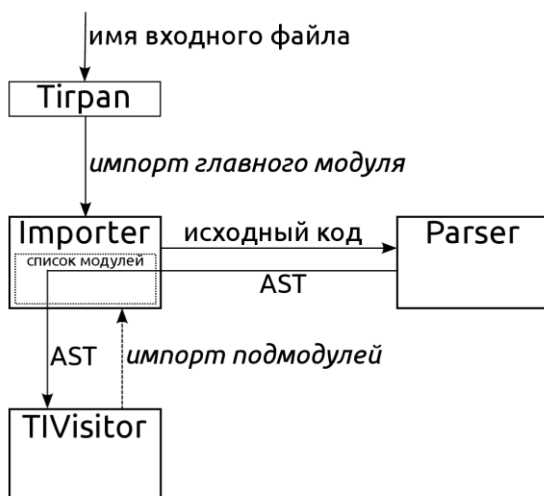


Рис. 7. Общая схема работы программы.

3.3 Структуры данных

В программе Tirpan используются две основные иерархии классов:

- классы, унаследованные от TypeNode (см. рис. 8), используются для представления типов, которые пропагируются по графу типовых переменных. Некоторые классы из этой иерархии имеют атрибуты, необходимые для правильного представления типа. Так, например, у классов TypeList и TypeTuple есть атрибут elems, хранящий множество типов элементов списка или, соответственно, кортежа. У классов TypeInt, TypeStr и TypeUnicode есть атрибут value, который хранит конкретное значение (например, 1, ’abc’ или u’abc’), если оно может быть вычислено, или None в противном случае. Это было сделано из-за того, что для точного анализа программы часто недостаточно просто знать, что определённая переменная хранит в себе некие строковые или целочисленные значения, а требуется точно вычислять это значение. В текущей реализации было решено отказаться от хранения значений для других встроенных типов. Это было сделано с целью улучшения производительности и экономии памяти.
- классы, унаследованные от TypeGraphNode, используются для представления типовых переменных, из которых конструируется граф вывода типов. При обходе синтаксического дерева для различных языковых конструкций создаются объекты различных классов этой иерархии, а «навешивание» информации о типах на AST заключается в том, что к узлам дерева динамически

добавляется атрибут link, ссылающийся на соответствующую типовую переменную. У всех классов TypeGraphNode-иерархии есть атрибут nodeType, хранящий в себе множество типов, т. е. экземпляров классов TypeNode-иерархии. Также некоторые классы могут иметь специфичные для них атрибуты: например, у VarTypeGraphNode есть поле name (имя переменной), а у AttributeTypeGraphNode — поле objects (множество всевозможных объектов для выражения вида x.y) и т. д.

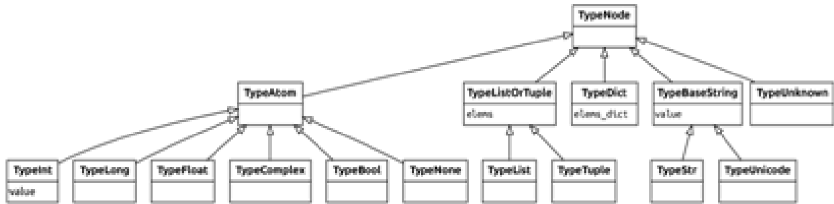


Рис. 8. Иерархия классов, унаследованных от TypeNode.

Вне двух основных иерархий находится класс Scope, используемый для представления областей видимости имён. В Scope хранится словарь, где ключами являются строки (имена переменных), а значениями — соответствующие переменные (объекты класса VarTypeGraphNode). Кроме того, у класса Scope есть атрибут parent, хранящий в себе ссылку на более верхнюю (родительскую) область видимости. У переменных также есть поле parent — оно указывает на область видимости, в которой находится данная переменная.

3.4 Константы и стандартные структуры языка Python

Самый простой случай при выводе типов — анализ констант (целочисленных, вещественных и т. д.) и строковых литералов. Каждый узел AST, соответствующий константе (литералу), связываются со своей типовой переменной — объектом класса ConstTypeGraphNode. Значением атрибута nodeType у этой типовой переменной становится множество из одного элемента: экземпляра класса из TypeNode-иерархии, соответствующего типу константы. Например, для константы 3.14 таким элементом является объект класса TypeFloat, а для строкового литера ’abc’ — объект класса TypeStr со значением ’abc’ в поле value (как уже говорилось, для целых чисел и строк при анализе учитывается не только тип, но и значение).

Теперь рассмотрим, как происходит анализ стандартных структур языка Python: списков, кортежей и словарей. Если при обходе синтаксического дерева встретился узел «список» (ast.List), то:

1. Создаётся типовая переменная (назовём её list_var) — объект класса ListTypeGraphNode.
2. Обходятся поддеревья, соответствующие элементам списка.
3. От каждой типовой переменной (назовём её elem_var), соответствующей i-му элементу списка, проставляется связь вида Elem к переменной list_var.

При добавлении некоторого ребра (связи) между типовыми переменными автоматически происходит пропагация типов вдоль этого ребра. То, как в результате пропагации изменится значение типовой переменной на конце ребра, зависит от вида связи и, возможно, от того, какому классу принадлежит эта типовая переменная. Если значение переменной меняется, то изменение распространяется по рёбрам, исходящим из этой переменной, и т. д.

Связь вида Elem от elem_var к list_var означает, что в списки, входящие в множество значений list_var, необходимо добавить типы, пришедшие из elem_var. Ситуация проиллюстрирована на рис. 9. Для представления типов «списки» используются объекты класса TypeList.

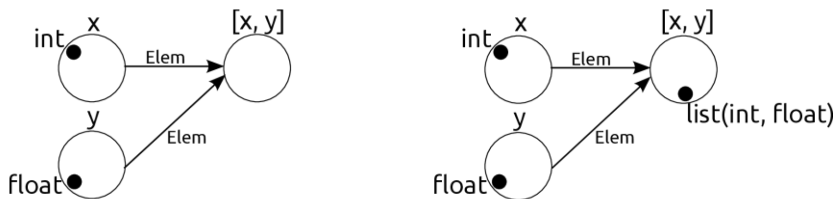


Рис. 9. Связи вида Elem, созданные при анализе списка [x, y]. Слева показаны типовые переменные до пропагации типов по соответствующим рёбрам, справа — после. Если в type(x) добавится новый тип, например, long, тип на другом конце связи превратится из {list(int, float)} в {list(int, long, float)}.

Аналогичным образом при анализе кортежей создаются и обрабатываются типовые переменные класса TupleTypeGraphNode. В отличие от ситуации со списками, связь Elem от elem_var к tuple_var задаётся с аргументом — индексом элемента в кортеже. Это необходимо для правильного вычисления возможных типов tuple_var. А при анализе словарей используются типовые переменные класса DictTypeGraphNode. К таким переменным могут проставляться связи двух видов: вида Key (от ключей словаря) и вида Val (от значений словаря).

3.5 Имена

Если при обходе дерева встретился узел «имя» (ast.Name), производится поиск типовой переменной класса VarTypeGraphNode с соответствующим именем (поле node.name в узле). Если такая переменная найдена, к ней проставляется ссылка от узла. В противном случае переменная с нужным именем создаётся в текущей области

видимости и опять же «навешивается» на анализируемый узел. Из этой простой логики есть свои исключения: например, если в теле некой функции в левой части присваивания есть переменная x, определять, глобальная или локальная переменная имеется в виду, необходимо по наличию директивы global x в том же теле функции. Такая семантика учитывается при анализе ast.Name.

3.6 Присваивания

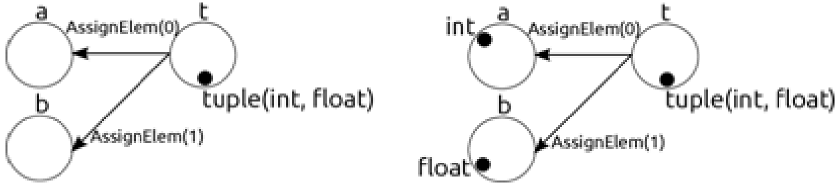


Рис. 10. Связи вида AssignElem, создаваемые при анализе присваивания списку переменных, например: a, b = t. Слева показаны типовые переменные до пропагации типов по соответствующим рёбрам, справа — после.

При анализе обычных присваиваний вида x = y сначала обходятся левая и правая часть присваивания. Затем от типовой переменной, соответствующей правой части, проставляется связь вида Assign к типовой переменной, соответствующей левой части. Связи вида Assign соответствуют обычным связям из «базового» алгоритма: все типы, которые были выведены для типовой переменной y, автоматически пропагируются в типовую переменную x.

Помимо обычных присваиваний, в Python присутствует конструкция вида x_1, …, x_n = y (присваивание списку переменных). При анализе таких присваиваний в первую очередь обходятся правая часть присваивания и все переменные в левой части. Затем от типовой переменной, соответствующей правой части, проставляются связи вида AssignElem к типовым переменным, соответствующим элементам списка присваивания. Для правильного вывода типов при добавлении связи к ней приписывается целочисленный аргумент: индекс переменной в списке. Ситуация проиллюстрирована на рис. 10. Связь вида AssignElem также используется при обработке конструкции in, в которой тип итератора определяется как множество элементов, содержащихся в типах коллекции. Так, для следующего примера будет вычислено, что тип коллекции равен $\{list(int, float, str)\}$, а тип итератора, соответственно, равен $\{int, float, str\}$:

```
collection = [1, 3.14, 'abc']
for iterator in collection:
    print iterator
```

3.7 Функции

В программе на языке Python объявление именованной функции, например foo, равносильно объявлению безымянной функции и связыванию её с переменной foo (в дальнейшем с foo может быть связано значение любого другого типа). По аналогии с выполнением программы при анализе объявления функции создаются два узла из TypeGraphNode-иерархии: UsualFuncDefTypeGraphNode и UsualVarTypeGraphNode (с таким же именем, как у функции), причём от первого объекта ко второму проставляется связь вида Assign. В объекте класса UsualFuncDefTypeGraphNode хранятся ссылки на параметры функции (в виде области видимости, в которую были добавлены соответствующие переменные) и на AST для тела функции. Этот AST при анализе определения функции не обходится — его обход откладывается до момента вызова, что также соответствует семантике языка Python. Кроме того, в объекте UsualFuncDefTypeGraphNode создаётся изначально пустое множество шаблонов для функции. Определения безымянных (лямбда-) функций обрабатываются аналогично, только в этом случае элемента UsualVarTypeGraphNode не создаётся.

При обработке узла «вызов функции» в первую очередь обходится поддерево «имя функции», затем поддеревья, соответствующие аргументам вызова, и в конце концов создаётся объект класса FuncCallTypeGraphNode, который «навешивается» на узел. К этому объекту проставляется два вида связей:

- от типовых переменных, соответствующих аргументам вызова, — связи вида Arg;
- от типовой переменной, соответствующей имени вызываемой функции, — связь вида Func.

Далее обработка вызова происходит в соответствии с АДП. Сначала вычисляется декартово произведение типов аргументов вызова и возможных значений типовой переменной, соответствующей имени функции. Затем для каждого элемента декартова произведения — обозначим элемент как $(f, a_1, \dots, a_n)$ — проверяется соответствие числа параметров f и аргументов вызова (и в параметрах, и в аргументах могут встречаться *args- и **kwargs-конструкции), т. е. возможен ли вообще вызов f с аргументами $(a_1, \dots, a_n)$. Если вызов возможен, производится проверка, не существует ли у функции шаблона для данных аргументов. Если шаблон существует, то возможные типы возвращаемого значения функции будут браться из него. Если же шаблона не существует, то он создаётся в виде структуры, хранящей следующие атрибуты:

- args — типы аргументов, для которых был создан шаблон;
- ast — AST для шаблона;

- result — множество возвращаемых из функции типов (инициализируется пустым множеством).

Каждый раз при создании нового шаблона из соответствующего объекта UsualFuncDefTypeGraphNode берётся сохранённый AST для функции и генерируется его точная копия, которая записывается в атрибут ast в шаблоне. Затем делается точная копия параметров функции, и в скопированные типовые переменные записываются мономорфные типы аргументов вызова. После этого производится обход скопированного синтаксического дерева для функции. Работа именно с различными копиями одного и того же дерева нужна для того, чтобы не смешивать типы параметров, локальных переменных и, в конечном счёте, возвращаемых значений функций для разных шаблонов. Если при обходе скопированного дерева встречаются узлы «оператор return», типы возвращаемых в операторе значений добавляются в множество, которое хранится в атрибуте result в шаблоне. После обработки шаблона типы из result добавляются в поле nodeType у объекта FuncCallTypeGraphNode, т. е. в значение соответствующей вызову типовой переменной.

Благодаря тому, что шаблон для функции создаётся перед обходом её тела, зацикливания при анализе рекурсивных функций не возникает. Действительно, рассмотрим рекурсивную реализацию функции factorial:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

При обходе выражения factorial(n - 1) алгоритм обнаруживает, что выражение n - 1 имеет тип int и что соответствующий шаблон для factorial уже был создан. Повторного обхода функции, который привёл бы к зацикливанию, не происходит.

Наличие связей вида Arg и Func гарантирует, что при расширении типов аргументов вызова или вызываемых функций происходит повторное вычисление декартова произведения и, возможно, расширение типа выражения «вызов функции».

3.8 Классы и объекты

Семантика определения класса в Python в чём-то схожа с семантикой определения функции. Определение класса A предписывает создать безымянный класс и связать его с переменной A. Однако, в отличие от

ситуации с функцией, тело класса начинает выполняться (интерпретироваться) сразу же после связывания. Поведение алгоритма вывода типов моделирует эту семантику. При анализе узла «определение класса» создаются объекты UsualClassDefTypeGraphNode и UsualVarTypeGraphNode (с именем класса), и, также как для функций, от первого ко второму объекту проставляется связь вида Assign. Объект класса UsualClassDefTypeGraphNode хранит в себе следующую информацию:

- данные о базовых классах для текущего класса (в настоящий момент не учитывается, что базовые классы могут быть изменены динамически);
- область видимости (изначально пустая), в которую будут добавляться все переменные, являющиеся членами текущего класса;
- список экземпляров класса (изначально пустой).

После того, как были созданы типовые переменные, текущей областью видимости становится область видимости класса и производится обход его тела.

Создание экземпляра определённого класса A в языке Python выглядит как вызов функции A с некоторым (возможно, пустым) списком аргументов, которые передаются в конструктор класса A. Такая семантика учитывается при анализе вызовов функций. Если оказывается, что в некотором элементе декартова произведения f (см. 3.7) — это определённый класс A, то выполняются специальные действия, несколько отличающиеся от обычной обработки вызова функции:

1. Производится поиск конструктора A.__init__. Если конструктора в классе A нет, считается, что происходит вызов конструктора по умолчанию, принимающего пустой список аргументов.
2. Проверяется, соответствуют ли параметры конструктора аргументам вызова, т. е. возможен ли вызов конструктора с такими аргументами.
3. Если проверка соответствия параметров и аргументов прошла успешно, ищется шаблон, соответствующий аргументам вызова. Если такой шаблон найден, то тип возвращаемого значения берётся из него.
4. В противном случае создаётся новый объект ClassInstanceTypeGraphNode, хранящий в себе информацию о созданном в результате вызова экземпляре класса (в текущей реализации и ClassDefTypeGraphNode, и ClassInstanceTypeGraphNode, хотя и являются частью TypeGraphNode-иерархии, используются также в качестве типа, как если бы они был частью TypeNode-иерархии). Созданный объект добавляется в список экземпляров класса A, хранящийся в

соответствующем объекте ClassDefTypeGraphNode. Проставляется и ссылка от поля cls в ClassInstanceTypeGraphNode к соответствующему ClassDefTypeGraphNode.

Дальнейшие действия практически совпадают с обработкой обычного вызова функции: производится копирование AST и параметров функции, связывание параметров с мономорфными типами аргументов и обход AST. Единственным отличие от обычного вызова здесь является то, что параметр self в конструкторе связывается с только что созданным объектом ClassInstanceTypeGraphNode.

3.9 Атрибуты и операции взятия элемента коллекции

Поскольку работа с экземплярами класса осуществляется при помощи операции чтения или записи атрибутов (это верно как для полей, так и для методов в объектах), необходима поддержка соответствующей конструкции при выводе типов.

Если при обходе дерева встретился узел «доступ к атрибуту», то в первую очередь обходится поддерево value, представляющее объект, у которого берётся атрибут, а затем создаётся типовая переменная — объект класса AttributeTypeGraphNode (назовём её var_attr). В эту типовую переменную записывается информация об имени атрибута, необходимая для правильного определения типа выражения. От типовой переменной, «навешенной» на поддерево value (назовём переменную var_value), проставляется связь вида AttrObject к var_attr. В обратную сторону (от var_attr к var_value) также проставляется связь, но вида AssignObject. Двухсторонние связи отражают тот факт, что при расширении типа объектов необходимо пересчитывать тип атрибутов, и наоборот.

К связям AttrObject/AssignObject могут прибавляться связи, созданные во время анализа присваиваний. Так, для самого простого случая записи в атрибут (выражения x.y = z) к var_attr будет идти связь вида Assign от типовой переменной для z. При пропагации новых типов по связи Assign в типовой переменной var_attr происходит процедура пересчёта значения переменной: сначала в функции set_attributes записывается информация о том, что атрибут self.attr во всех возможных объектах (это множество хранится в self.objects) может принимать новые типы — элементы множества self.values, затем в функции get_attributes вычисляется новое значение типовой переменной (поле self.nodeType):

```
def process(self):
    set_attributes(self.objects, self.attr,
self.values)
```

```
    self.nodeType = 
get_attributes(self.objects, self.attr)
```

В функции get_attributes учитывается, что некоторые объекты могут иметь метод __getattr__, переопределяющий стандартную операцию чтения атрибута (например, в методе может быть реализовано чтение одноимённого атрибута из другого объекта).

Почти аналогично производится вывод типов для узлов «взятие элемента» (ast.Subscript), например, для выражения x[0]. Для таких узлов создаётся типовая переменная класса SubscriptTypeGraphNode, хранящее в себе вместо имени атрибута информацию о типе (и, возможно, значении) выражения «индекс». Как и в случае с атрибутами, здесь генерируются связи вида AttrObject и вида AssignObject. Метод process в SubscriptTypeGraphNode похож на аналогичный метод в AttributeTypeGraphNode, только в первом вызываются функции set_subscripts и get_subscripts.

3.10 Внешний код

Для поддержки написанного не на языке Python кода в TypeGraphNode-иерархию были добавлены новые классы: ExternVarTypeGraphNode и ExternFuncDefTypeGraphNode. Как следует из названия, они нужны для моделирования внешних переменных и, соответственно, внешних функций. Расскажем подробнее об каждой из групп.

Примером внешней (по отношению к коду на Python) переменной является path из модуля sys. Перед началом выполнения Python-программы sys.path связывается со списком строк, каждая из которых представляет собой некоторый путь в файловой системе. Список частично формируется из содержимого переменной окружения PYTHONPATH, но содержит в себе и элементы, зависящие от текущей инсталляции интерпретатора Python. Для моделирования подобных внешних переменных перед непосредственным запуском вывода типов (импортом модуля __main__) происходит создание ряда объектов ExternVarTypeGraphNode, в том числе и для sys.path. Для каждого объекта указана определённая функция. После создания некоторого объекта класса ExternVarTypeGraphNode вызывается соответствующая ему функция, и результат её работы записывается в поле nodeType у объекта. Например, для sys.path функция инициализации выглядит следующим образом:

```
def quasi_sys_path():
    res = TypeList()
```

```
    tmp = []
    ...
    for elem in sys.path[1:]:
        tmp.append(get_new_string(elem))
    res.elems = tmp
    return res
```

В качестве примера внешней функции можно привести уже упоминавшуюся append, принимающую на вход объект-список и элемент, который в этот список необходимо добавить. Для обработки append при выводе типов ей, как и в случае с sys.path, ставится в соответствие некоторая функция, только на этот раз функция принимает на вход типы параметров для append. На выходе по ним строится тот тип, который имело бы возвращаемое значение append, а также, возможно, моделируется внесение изменений в типы входных параметров (в случае с append тип первого параметра меняется в ходе выполнения функции). Таким образом работа функции append моделируется с точки зрения типов на входе в функцию и на выходе из неё:

```
def quasi_append(scope):
    type1 = getParamType(scope, 1)
    type2 = getParamType(scope, 2)
    if not isinstance(type1, TypeList):
        return set([type_none])
    type1.add_elem(type2)
    return set([type_none])
```

Каждой поддерживаемой внешней функции ставится в соответствие типовая переменная класса ExternFuncDefTypeGraphNode. У объектов этого класса нет атрибута ast, зато в атрибуте quasi хранится соответствующая моделирующая функция. Если при обработке вызова функции встретился объект ExternFuncDefTypeGraphNode, вместо обхода сохранённого AST происходит вызов соответствующей моделирующей функции.

4. Результаты

Как уже говорилось, для демонстрации работоспособности Tirpan используется пример несоответствия типов из проекта Gramps. В этом примере переменная css_filename в результате работы программы может принимать значение [], передача которого в os.path.basename вызывает ошибку:

```
def set_css_filename(self, css_filename):
    if os.path.basename(css_filename):
        self.css_filename = css_filename
    else:
        self.css_filename = ''
```

Тестирование Tirpan показало, что тип list() (пустой список) правильно выводится в качестве одного из возможных типов для css_filename. Однако, информация о типах, «навешенных» на узлы синтаксического дерева, является внутренней и пользователю непосредственно не видна. Поэтому было решено реализовать прототип обнаружителя дефектов (чекера), который выводил бы сообщение о дефекте «передача аргумента, не являющегося строкой, в функцию os.path.basename» в случае наличия такого дефекта в коде.

В текущей реализации чекер работает не после этапа вывода типов, а непосредственно во время него. Он представляет собой callback (функцию обратного вызова), который вызывается каждый раз при обходе нового узла AST. Чекер проверяет, не встретилась ли функция os.path.basename и, если встретилась, каковы возможные типы её (первого) аргумента. Если не все типы являются строками (т. е. объектами классов TypeStr и TypeUnicode), чекер рапортует сообщение о дефекте несоответствия типов.

Для того, чтобы подобное сообщение было более информативно, в модуль вывода типов было добавлено сохранение трассировки стека функций (traceback). Этой цели в Tirpan служит объект класса Backtrace, хранящий в себе стек *фреймов* — элементов **Ошибка!**, входящих в декартово произведение и анализируемых при обработке вызовов функций. Перед обходом AST для некоторого шаблона функции соответствующий фрейм добавляется в стек, а после обхода — удаляется из стека. Сообщение о найденном дефекте, содержащее трассировку стека, выводится по окончании исполнения программы и выглядит следующим образом:

```
FUNC.ARG.WRONG: 'os.path.basename' expects
<basestring object>, not []. Backtrace:
    <HtmlDoc object>.set_css_filename([])
    <TextReportDialog object>.make_document()
    <TextReportDialog object>.on_ok_clicked(None)
    <Dialog object>.run()
    report(<DbState object>, <DisplayState
object>, ?,
           <class 'FamilySheet'>, <class
'FamilySheetOptions'>,
           'Family Sheet', 'FamilySheet',
           <int value>, <bool value>)
    run_plugin(<PluginData object>, <DbState
object>, <DisplayState object>)
    <lambda>(None)
    main()
    startgtkloop([], <ArgParser object>)
    run()
```

Однако, предположим, что код функции set_css_filename был исправлен, чтобы избежать несоответствия типов, например, следующим образом (очевидно, что возможны и другие варианты):

```
def set_css_filename(self, css_filename):
    if css_filename and
os.path.basename(css_filename):
        self.css_filename = css_filename
    else:
        self.css_filename = ''
```

Здесь в os.path.basename уже не может передаваться пустой список, так как он вычисляется в False. Однако, поскольку алгоритм вывода типов, лежащий в основе Tirpan, нечувствителен к потоку выполнения, изменение условия будет проигнорировано и дефект FUNC.ARG.WRONG будет по-прежнему рапортоваться. Чтобы подобного ложного срабатывания (false positive) не происходило, в алгоритм вывода типов было внесено небольшое исправление, позволяющее фильтровать типы для переменных, используемых в условиях if.

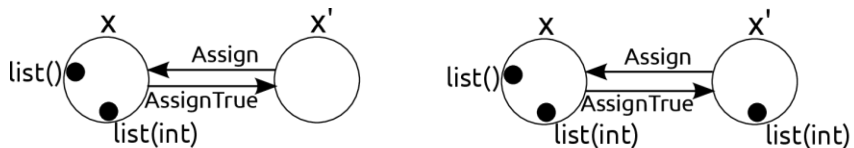


Рис. 11. Связь вида AssignTrue, создаваемая при анализе условия if x:. Слева показаны типовые переменные до пропагации типов по соответствующим рёбрам, справа — после

Исправление такое: если в условии if встречается конъюнкция (возможно, вырожденная), то её операнды обходятся слева направо, и для каждого имени (узла ast.Name) выполняется следующая последовательность действий. Пусть мы встретили имя x. В этом случае создаются новая область видимости и новая же переменная с именем x, которая добавляется в созданную область видимости (чтобы не путать новую переменную со старой, будем называть новую x’). Затем между ними проставляются следующие связи: от x’ к x обычная связь вида Assign, а от x к x’ специальная связь вида AssignTrue, по которой могут распространяться лишь типы, вычисляемые в True. Если внутри ветки if с x будет связано значение нового типа, этот тип будет добавлен в x’ и автоматически распространится в x. Связи между типовыми переменными проиллюстрированы на рис. 11.

Описанная модификация алгоритма позволяет избежать ложного срабатывания как для приведённого выше примера, так и, например, для варианта, когда старый условный оператор (if os.path.basename(css_filename):) добавляется в тело нового условного оператора (if css_filename:).

Тем не менее на проекте Gramps было обнаружено 4 ложных срабатывания реализованного чекера. Исследование показало, что ложные срабатывания вызваны следующими причинами:

- не всегда корректной работой со значениями словарей, из-за чего в множество возможных типов для аргумента os.path.basename попадают неправильные элементы;

- недостаточной чувствительностью к потоку выполнения: тот факт, что «плохие» типы отсеиваются с помощью условий в операторе if, часто не учитывается при выводе типов.

Второй пункт требует пояснений: хотя в алгоритм и была внесена модификация, учитывающая использование имён в условиях if, в более сложных случаях фильтрация типов не работает. Например, если вместо имени в условии стоит атрибут некоторого объекта или вызов функции. Вот один из примеров реального кода (стандартный модуль webbrowser), для которого в текущей реализации заданные в операторе if условия игнорируются:

```
if isinstance(name, basestring):
    self.name = name
else:
    self.name = name[0]
self.basename = os.path.basename(self.name)
```

Перечислим другие известные проблемы и ограничения в текущей реализации Tirpan:

- существуют проблемы с производительностью (время работы — несколько десятков минут) и объёмом потребляемой оперативной памяти (около 3 ГБ) на проекте Gramps;
- для того, чтобы избежать зацикливания алгоритма при наличии циклических связей (создаваемых, например, при анализе кода x = [x]) число возможных типов для любого выражения ограничено сверху константой, что может ухудшать точность анализа;
- с целью улучшения производительности и экономии памяти введены ограничения на количество возможных шаблонов для одной функции; кроме того, не для всех файлов вычисляются значения (а не только типы) целочисленных и строковых констант, что также может ухудшать точность.

5. Заключение

На основе модифицированного алгоритма декартова произведения был реализован работоспособный модуль вывода типов, который в состоянии обнаруживать несоответствие типов на реальных Python-проектах. Информация, полученная на этапе вывода типов, может быть использована не только для поиска дефектов, но и для рефакторинга кода, навигации по нему или вычисления определённых метрик.

Представляются возможными несколько направлений дальнейшей работы:

- устранение проблем производительности и вытекающих из них ограничений;
- улучшение вывода типов для уменьшения числа ложных сообщений о дефектах;
- генерация для каждого обнаруженного дефекта его трассы — последовательности шагов, из которой можно понять, как неправильный тип попал в функцию, содержащую несоответствие типов (что должно помочь пользователю статического анализатора определить, является дефект истинным или ложным).

Исходя из того, что трассировка стека, которая в текущей реализации выводится вместо трассы дефекта, является малоинформативной, последнее направление деятельности видится весьма актуальным. Вероятно, при выводе типов наряду с обычными связями следует генерировать «обратные» им, чтобы впоследствии проследить, откуда «плохие» типы пришли в место, где располагается дефект. В таком случае генерация трассы будет представлять собой поиск определённого пути в графе типовых переменных.

6. Список литературы

[1]. В. Савицкий, Д. Сидоров. Инкрементальный анализ исходного кода на языках C/C++. Труды ИСП РАН, том 22, сс. 119—129, 2012.

[2]. TIOBE Programming Community Index for April 2013. http://tinyurl.com/cgjbmjc

[3]. Gramps Bug Report 005023. http://www.gramps-project.org/bugs/view.php?id=5023

[4]. O. Agesen. The Cartesian Product Algorithm. ECOOP’95 Proceedings of the 9th European Conference on Object-Oriented Programming (1995).

[5]. J. Palsberg, M. Schwartzbach. Object-Oriented Type Inference. In OOPSLA’91 Object-Oriented Programming Systems, Languages and Applications, pp. 146—161, Phoenix, Arizona, Oct. 1991.

[6]. M. Salib. Starkiller: a static type inferencer and compiler for Python. The Master of Engineering degree thesis. Massachusetts Institute of Technology, 2004.

[7]. B. Alpern, M. Wegman, K. Zadeck. Detecting equality of values in programs. In Conference Record of the 15th ACM Symposium on Principles of Programming Languages (Jan. 1988), ACM, New York, pp. 1—11.

[8]. Abstract Syntax Trees: ast module. http://docs.python.org/2/library/ast.html

Type inference for Python programming language

Bronshteyn I. E. (ISP RAS, Moscow, Russia)
ibronstein@ispras.ru

Abstract. The article presents type inference for programs written in Python programming language. At first, type inference algorithms for parametric polymorphism that were described in the scientific literature are reviewed. The algorithms evolved from "basic" algorithm (also known as Palsberg — Schwartzbach algorithm) to Cartesian product algorithm (CPA). It was shown that CPA is both precise and efficient, however it has to be modified to be used on Python code. Afterwards, we present a new algorithm (a modification of CPA) for Python. It is shown how type inference module using the new algorithm analyses various Python language structures: constants (literals), basic Python collections, variable names, assignments, function and class definitions, attribute and index expressions. It is also shown how the algorithm deals with external (non-Python) functions using special annotations that specify output types depending on input types of the function. Afterwards, the results of work on the prototype (module that implements described type inference algorithm) are presented. The paper is concluded by an overview of possible future work directions such as generating a defect trace, i.e. description that specifies how expression got its incorrect types.

Keywords: python; type inference; dynamic typing; static analysis; defects detection.

References

[1]. V. Savitskij, D. Sidorov. Inkremental'nyj analiz iskhodnogo koda na yazykakh C/C++ [Incremental source code analysis for C/C++ languages]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 119—129 (in Russian).
[2]. TIOBE Programming Community Index for April 2013. http://tinyurl.com/cgjbmjc
[3]. Gramps Bug Report 005023. http://www.gramps-project.org/bugs/view.php?id=5023
[4]. O. Agesen. The Cartesian Product Algorithm. ECOOP'95 Proceedings of the 9th European Conference on Object-Oriented Programming (1995).
[5]. J. Palsberg, M. Schwartzbach. Object-Oriented Type Inference. In OOPSLA'91 Object-Oriented Programming Systems, Languages and Applications, pp. 146—161, Phoenix, Arizona, Oct. 1991.
[6]. M. Salib. Starkiller: a static type inferencer and compiler for Python. The Master of Engineering degree thesis. Massachusetts Institute of Technology, 2004.
[7]. B. Alpern, M. Wegman, K. Zadeck. Detecting equality of values in programs. In Conference Record of the 15th ACM Symposium on Principles of Programming Languages (Jan. 1988), ACM, New York, pp. 1—11.
[8]. Abstract Syntax Trees: ast module. http://docs.python.org/2/library/ast.html

Автоматический поиск ошибок синхронизации в приложениях на платформе Android

В.П. Иванников, С.П. Вартанов, М.К. Ермаков [1]
ivan@ispras.ru, svartanov@ispras.ru, mermakov@ispras.ru

Аннотация. В данной статье рассматривается задача автоматического поиска ошибок синхронизации при проведении динамического анализа приложений в рамках платформы Android. Приводится теоретический обзор существующих методов проведения подобного анализа, затем рассматриваются основные особенности приложений платформы Android с точки зрения применения данных методов. Приводится обзор существующих инструментов для анализа компонентов Android-приложений как в нативном кода на языке Си, так и реализованных на языке Java. Также в статье описан механизм обнаружения ошибок синхронизации, используемый инструментом динамического анализа байт-кода Coffee Machine и его основные аспекты: инструментирование, генерация трассы отношений предшествования и использование инструмента ThreadSanitizer Offline для обнаружения состояния гонок по сгенерированной трассе для платформы Android.

Ключевые слова: динамический анализ, поиск ошибок синхронизации, Android

1. Введение

Платформа Android, вышедшая на рынок операционных систем для мобильных устройств в 2008 году, прочно занимает одно из лидирующих мест в настоящее время. Разработкой приложений для платформы занимается большое число разработчиков, как в составе крупных компаний в сфере информационных технологий, так и выполняющих работу в небольших командах или в индивидуальном порядке. Как и в общем случае при разработке программного

---

1 Работа проводится при финансовой поддержке Министерства образования и науки Российской Федерации

обеспечения, контроль качества (в том числе устранение дефектов) является одной из приоритетных задач, что в свою очередь обуславливает востребованность средств и инструментов, позволяющих производить подобные действия автоматически.

Для платформы Android приоритетность проведения анализа разрабатываемого программного обеспечения в контексте производительности и стабильности является большей в сравнении с соответствующим аспектом в разработке пользовательских приложений, рассчитанных на использование на персональных компьютерах. Данное различие возникает за счёт ограниченности ресурсов на мобильных устройствах и более жёсткой системы контроля распределения доступных ресурсов, реализованной на платформе Android.

1.1. Приложения платформы Android

Приложения для платформы Android разрабатываются на языке программирования Java или на других языках, транслируемых на этапе компиляции в байткод Java. Полученный байткод затем транслируется в другой формат для виртуальной машины Dalvik, производящей непосредственное выполнение приложения в операционной системе Android. Виртуальная машина Dalvik реализована на языке Си и предоставляет возможности исполнения нативного кода в рамках Java-приложения. Таким образом, проведение статического или динамического анализа приложений с целью обнаружения дефектов или выявления сегментов, влияющих на производительность, должно покрывать как приложения, рассматриваемые в рамках виртуальной машины Dalvik, так и компоненты, реализованные в нативном коде. Несмотря на общие рекомендации по минимальному использованию нативного кода, сформулированные создателями платформы Android, большое количество приложений реализованы таким образом, что фрагменты, отвечающие за вычисления с плотной нагрузкой, вынесены за пределы исходного кода на языке Java.

1.2. Многопоточное исполнение

Операционная система Android практически полностью базируется на использовании графических пользовательских интерфейсов. Реализация интерфейсов на платформе Android выполняется в рамках виртуальной машины Dalvik и, соответственно, использует механизмы и структуры языка Java.

Реализация графического пользовательского интерфейса требует использования многопоточной схемы, в которой для активного приложения существует как минимум одна нить выполнения, осуществляющая регистрацию событий, соответствующих действия пользователя, и как минимум одна нить, осуществляющая выполнение необходимых ответных действий. Приложения, работающие в

операционной системе Android, помимо этого общего ограничения подчиняются стандартным протоколам, определяющих цикл жизни процессов и возможные состояния — необходимо обеспечивать единый подход к использованию мобильного устройства и производить корректное распределение ресурсов.

Указанные особенности и базовые положения платформы Android обуславливают востребованность средств, производящих автоматический анализ Java-составляющей приложений с целью выявления ошибок синхронизации и удовлетворения стандартным положениям реализации графического пользовательского интерфейса.

Начиная с версии Android 3.0 [1] разработчиками была добавлена совместимость с многоядерными архитектурами. Изменения затронули составляющие Android (виртуальную машину Dalvik, основную библиотеку bionic и т. д.), так и предоставили возможность пользоваться параллельными вычисления в нативном коде. В свою очередь, это ставит вопрос об эффективности применения автоматических средств для обнаружения возможных ошибок синхронизации в нативном коде.

1.3. Структура статьи

В настоящей статье рассматривается подход к проведению динамического анализа приложений для платформы Android, позволяющий автоматически выявлять ошибки синхронизации. В разделе 2 приводятся основные алгоритмы, используемые в современных инструментах, их достоинства и недостатки. В разделе 3 рассматриваются особенности проведения анализа нативного кода для приложений Android и приводится обзор существующих инструментов. В разделе 4 приводится описание подхода к проведению анализа для Java-составляющей приложений Android и данные о реализации подхода и полученных результатах. В разделе 5 подводятся итоги проведённого исследования.

2. Динамический поиск ошибок синхронизации

На данный момент инструменты динамического анализа, позволяющие автоматически обнаруживать ошибки синхронизации, реализуют подходы, основанные на одном из двух общих методов: рассмотрении набора блокировок (*lockset*) или на основе отношений предшествования (*happens-before relations*). Существуют также гибридные подходы, сочетающие в себе преимущества обоих алгоритмов и позволяющие уменьшить влияние недостатков алгоритмов на эффективность анализа. Все три типа алгоритмов основаны на выделении событий синхронизации, производимых на основе стандартных средств (семафоров, сигналов и т. д.) и выделении событий доступа к разделяемым между несколькими нитями исполняемой программы. Непосредственная оценка событий доступа и событий синхронизации в

контексте положения о том, что выполнение инструкций отдельными нитями может идти в произвольном порядке (друг относительно друга), позволяет обнаружить состояния гонки и взаимной блокировки.

2.1. Набор блокировок

Метод набора блокировок основан на рассмотрении каждой отдельной нити выполнения как последовательности событий, где каждому событию соответствует некоторое состояние. Данное состояние включает в себя набор объектов, над которым производятся действия, набор блокировок данной нити и набор параметров производимых действий (в частности, тип доступ к объектам – по записи или по чтению). Соответственно, возможные события включают в себя работу с объектами, выставление блокировки для остальных нитей, т. е. получение эксклюзивного доступа для работы в критической секции, и освобождение блокировок (выход из критической секции).

Непосредственный анализ приложения на предмет ошибок синхронизации производится путём рассмотрения подобных последовательностей для всех нитей процесса. Предполагается, что работа с разделяемыми ресурсами является корректной только тогда, когда выполняется следующее условие — доступ к некоторому объекту или набору объектов, одновременно выполняемый двумя или более параллельными нитями, где хотя бы одна из нитей осуществляет изменение содержимого объектов, должен происходить, когда все нити имеют хотя бы одну общую блокировку. С точки зрения средств синхронизации данное условие можно выразить более просто — для каждой критической секции должно иметься хотя бы одно средство синхронизации, которое осуществляет защиту данной секции и используется каждой нитью, претендующей на доступ к данной секции.

Выявление ошибок синхронизации при помощи данного метода возможно как во время работы, так и после завершения по сохранённой трассе событий отдельных нитей. Обновление наборов блокировок происходит напрямую при получении событий входа и выхода из критических секций.

Основным недостатком рассмотренного метод поиска ошибок синхронизации на основе наборов блокировок является использование строго критерия непосредственно ошибочной ситуации. Некоторые механизмы взаимодействия (прямая передача между нитями с помощью каналов, переработка объектов) нарушают заданный критерий, однако являются корректными с точки зрения параллельной работы программы. Соответственно анализ приложений, использующих данные механизмы, приведёт к ложным срабатываниям. Тем не менее данный подход является довольно простым в разработке.

2.2. Отношения предшествования

Концепция отношений предшествования была введена Лэсли Лэмпортом для исследования взаимодействия компонентов распределённых систем [2]. Два события *i* и *j* называются связанными отношением предшествования, если:

1. *i* и *j* произошли в рамках одной нити, причем набор инструкций, вызывающий событие *i,* был выполнен до набора инструкций, вызывающего событие *j*;
2. *i* и *j* произошли в нитях **A** и **B** соответственно, однако *i* является событием отправки сообщения от нити **A** к нити **B**, а *j* является событием получения данного сообщения.

Важной особенностью отношения предшествования является транзитивность — если событие *i* предшествует событию *j*, а событие *j* предшествует событию *k*, то событие *i* также предшествует событию *k*. Именно эта особенность позволяет рассматривать полное объединение событий, произошедших в нитях программы при выполнении и вынести суждение о наличии или отсутствии ошибочных ситуаций.

В рамках данного алгоритма критерием ошибки синхронизации для двух событий доступа к разделяемым объектам (хотя бы одно из которых производить изменение содержимого объектов) является невозможность упорядочить эти события по отношению предшествования. В самом деле, если нельзя утверждать ни то, что одно событие предшествует второму, ни то, что второе событие предшествует первому, то потенциально возможны оба варианта отношения времени возникновения этих событий, то есть имеется ситуация гонки.

Эффективная реализация данного метода является более технически сложной задачей, чем реализация метода наборов блокировок, однако рассмотрение отношений предшествования для событий доступа к разделяемым объектам потенциально устраняет возможность получения ложных срабатываний. Реальная степень устранения ложных срабатываний ограничена полнотой обработки возможных средств прямой коммуникации между параллельными нитями. Как и метод на основе наборов блокировок, подход, рассматривающий отношения предшествования, может применяться во время выполнения программы или по окончании выполнения на основе собранной трассы событий.

Важной особенностью данного метода является более узкое покрытие набора ошибочных ситуаций по сравнению с методом наборов блокировок. Ошибка синхронизации, обнаруженная при рассмотрении отношений предшествования, будет однозначно обнаружена при отслеживании состояний нитей и наборов блокировок, однако обратное утверждение не является верным [3]. При рассмотрении поиска дефектов с точки зрения ложных срабатываний соотношение двух методов меняется на противоположное — рассмотрение отношений предшествования можно производить при фильтрации сообщений об

обнаруженных ошибках синхронизации, найденных на основе исследования наборов блокировок.

2.3. Гибридный метод

Рассмотренные положения однозначно оправдывают эффективность использования гибридного метода, который предоставляет меньшее число ложных срабатываний и покрывает больший объем реальных ошибок синхронизации. Большая часть работ последних лет (например, [3, 7]) посвящена выработке наиболее эффективного применения двухсторонней проверке с помощью гибридного метода.

3. Анализ нативного кода

3.1. Особенности анализа нативного кода

Анализ нативного кода приложений Android затрудняется несколькими особенностями:

- Обращения к нативным методам производятся напрямую из виртуальной машины Dalvik, что делает необходимым анализ самой виртуальной машины для получения полной картины происходящего.
- Процесс запуска приложения (инициализация виртуальной машины Dalvik, загрузка необходимых компонентов) проводится специальным образом, что затрудняет проведение анализа выполнения.

Первая особенность значительно снижает эффективность применения статического анализа кода, в то время как вторая особенность создаёт значительные затруднения как раз для методов динамического анализа.

Общая схема запуска приложения выглядит следующим образом (см. рис. 1):

1. При запуске операционной системы создаётся процесс Zygote, инициализирующий виртуальную машину Dalvik, загружая необходимые системные классы и т. д.
2. Каждый последующий экземпляр Dalvik запускается с помощью стандартного механизма *fork-exec* из процесса Zygote. Непосредственно запросы на запуск копий виртуальной машины процесс Zygote получает от соответствующих сервисов в системе прослушивая специльный канал передачи данных.
3. Для нового экземпляра загружаются непосредственно классы запускаемого приложения, однако часть данных наследуется от процесса Zygote, породившего данный экземпляр.

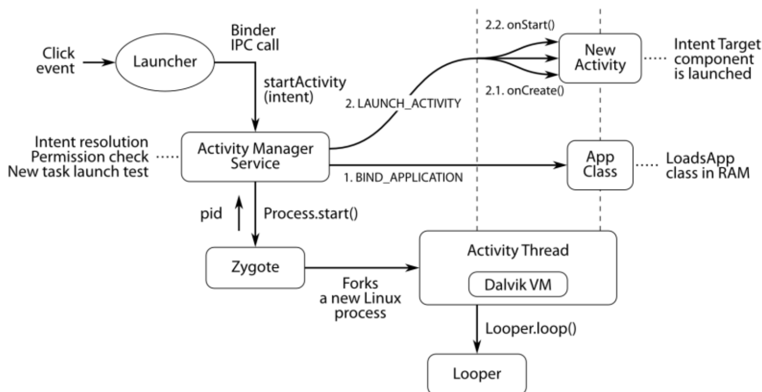


Рис. 1: Запуск Android-приложения

Непосредственная сложность для динамического анализа заключается в том, что должен или осуществляться перехват запуска процесса, или же проводится полный анализ выполнения и процесса Zygote.

В версии Android 3 разработчиками был добавлен специальный механизм обёртки вызовов *exec*, производящихся из процесса Zygote [4], что в значительной степени упрощает проведение динамического анализа, предоставляя пользователю возможность автоматически запускать необходимое средство анализа вместо нового экземпляра виртуальной машины Dalvik (соответственно новый экземпляр создаётся под контролем средства анализа).

Таким образом, проведение динамического анализа представляется более эффективным подходом для автоматического выявления ошибок сихронизации.

3.1. Средства динамического анализа

Среди современных средств динамического анализа с целью обнаружения ошибок синхронизации можно выделить несколько распространённых инструментов:

- Intel Thread Checker [5],
- Sun Thread Analyzer [6],
- helgrind, DRD, ThreadSanitizer [7] (в составе Valgrind).

Intel Thread Checker реализует метод на основе отношений предшествования и доступен (на пробной и коммерческой основе) на платформах Windows и Linux, однако не предоставляется для архитектуры ARM (на которой работает большинство мобильных

устройств с операционной системой Android). Sun Thread Checker производит анализ на основе гибридного метода, однако также не является доступным для архитектур семейства ARM.

Инструменты helgrind, DRD и ThreadSanitizer реализованы в рамках среды разработки средств динамического анализа Valgrind [8], распространяемого свободно. Поддержка архитектуры ARM и платформы Android была добавлена разработчиками в версиях 3.6.0 и 3.7.0 соответственно [9]. Инструменты helgrind и DRD основаны на методе отношений предшествования, но имеют ряд различий на более техническом уровне. Инструмент ThreadSanitizer был разработан независимо от основного комплекта Valgrind и основан на гибридном методе [7], позволяющем демонстрировать более высокую эффективность анализа по сравнению с helgrind и DRD. Реализация инструмента также работает быстрее остальных инструментов и обладает рядом дополнительных особенностей:

1. Помимо общего механизма подавления ошибок (*suppression*), используемом в Valgrind для фильтрации сообщений о ложных срабатываниях, в ThreadSanitizer имеется система фильтрации компонентов, загруженных в оперативную память и используемых при выполнение программы (по библиотекам и по отдельным функциям). Поддержка платформы Android в составе комплекса Valgrind не является полностью завершённой и, в частности, для инструментов helgrind и DRD не накоплено достачное число сигнатур ошибок, обеспечивающих подавление сообщений о ложных срабатываниях.
2. ThreadSanitizer предоставляет поддержку аннотаций исходного нативного кода, которые указывают на отсутствие реального дефекта в «подозрительных» фрагментах. Использование данных аннотаций позволяет снизить число ложных срабатываний в более сложных ситуациях.

Использование ThreadSanitizer является возможным при помощи указанного средства обёртки вызовов *exec*. Проведение анализа в данном случае охватывает большой объем нативного кода (фактически, рассматриваются ситуации, происходящие внутри виртуальной машины Dalvik), однако является единственным способом обработки вызовов нативного кода и выявления потенциальных ошибок синхронизации.

На основе проведенного обзора проведение динамического анализа применение инструмента ThreadSanitizer является наиболее эффективным подходом к автоматическому обнаружению возможных проблем зацикливания.

4. Анализ Java-приложений

4.1. ThreadSanitizer

Как было сказано ранее, ThreadSanitizer — проект, посвящённый динамическому поиску ошибок синхронизации.

С точки зрения алгоритма инструмента, важны следующие типы операций:

чтение и запись, т. е. операции доступа к памяти,

операции блокировок,

сигналы и ожидания сигналов,

операции работы с потоками.

Будем называть такие операции «синхронизационными».

Для алгоритма важно понятие сегментов потока. В терминах ThreadSanitizer, сегмент — непрерывная последовательность инструкций потока, в которой не встречаются указанные выше операции.

В основе алгоритма лежит механизм определения отношения предшествования между различными сегментами и синхронизационными операциями различных потоков. Отношение предшествования означает, что набор операций, стоящий в левой его части не может встречаться в трассе выполнения программы после набора операций, стоящего в правой части.

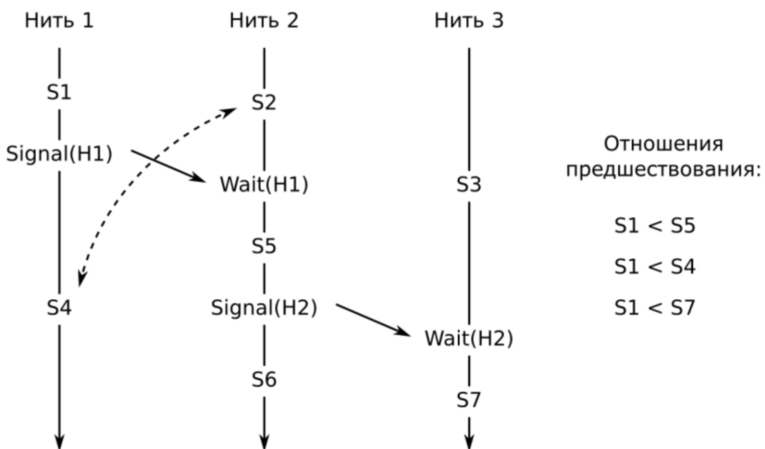


Рис. 2: Отношения предшествования

В терминах ThreadSanitizer, состояние гонки — доступ к общим данным из двух различных потоков в точках, между которыми нет отношения предшествования, если один из доступов есть доступ типа запись.

Если инструмент обнаружил дефект, это означает, что в соответствии с построенными отношениями предшествования, состояние гонки может возникнуть (но не возникает обязательно) на одном из запусков приложения. Обнаруженный дефект может быть ложным в случае, если построены не все отношения предшествования. Например, это может происходить, если некоторые части программы не доступны для анализа — не была произведена инструментация библиотеки или отдельного класса и соответствующие события не были зафиксированы в трассе.

Даже если обнаруженный дефект действительно имеет место, возникает проблема воспроизведения данной конкретной ошибки.

Основная часть проекта ThreadSanitizer — инструмент для обнаружения состояния гонки для программ, написанных на языке C++, однако здесь речь пойдёт о других, экспериментальных инструментах, созданных в рамках этого проекта — ThreadSanitizer Offline и Java ThreadSanitizer. Оба этих инструмента также основаны на методе построения отношений предшествования.

4.2. ThreadSanitizer Offline

ThreadSanitizer Offline — экспериментальный инструмент, реализующий вторую часть алгоритма обнаружения. Этот инструмент принимает на вход информацию о выполнившихся синхронизационных инструкциях в виде трассы событий и строит наборы отношений предшествования для поиска состояний гонки.

Таким образом, ThreadSanitizer Offline не зависит ни от языка программирования, на котором написано анализируемое приложения, ни от механизма генерации трассы событий. Необходимо лишь, чтобы трасса событий соответствовала синтаксису инструмента.

4.3. Java ThreadSanitizer

Java ThreadSanitizer — инструмент, производящий инструментацию байт-кода. Он реализует первую часть алгоритма обнаружения ошибок.

Инструмент производит динамическую инструментацию в ходе выполнения анализируемого приложения. Такая инструментация позволяет извлекать информацию о выполнении и строить трассу событий. Построенная трасса может быть использована инструментом ThreadSanitizer Offline для обнаружения состояний гонки.

Таким образом, совместно Java ThreadSanitizer и ThreadSanitizer Offline позволяют произвести поиск ошибок синхронизации в приложении на языке Java.

Для инструментации байт-кода Java ThreadSanitizer использует библиотеку ASM [10]. Эта библиотека использует шаблон проектирования посетитель и java agent для динамической инструментации байт-кода.

Для работы инструмента на платформе Android это представляет проблему. В связи с тем, что в операционной системе Android вместо виртуальной машины Java используется виртуальная машина Dalvik, использование java agent не представляется возможным. Другая проблема связана с отсутствием доступа к системным библиотекам и классам виртуальной машины Dalvik и таким образом часть информации о процессе выполнения может быть потеряна.

4.4. Coffee Machine

Coffee Machine — разработанный в рамках данного исследования инструмент для динамического анализа приложений, код которых может быть транслирован в байт-код виртуальной машины Java.

В соответствии с несколькими задачами, которые решает инструмент, его структура состоит из нескольких компонент:

- **Instrumentator**. Обработка анализируемго приложения, поиск зависимостей, построение списка классов для инструментации и статическая инструментация.
- **Interpreter**. Анализ трассы исполнения и построение булевых ограничений для решателя STP [11].
- **Main module**. Обеспечение взаимодействия компонент и запуск анализируемого приложения на исполнение.

Coffee Machine производит статическую инструментацию байт-кода, посредством добавления инструкций вызова методов классов `Daemon` и `Concurrency`, включёнными в состав инструмента.

Методы класса `Daemon` производят трассировку помеченных данных (информации, которая зависит от входных данных) и генерируют набор булевых ограничений для решателя STP. В ходе выполнения приложения `Daemon` для каждого условного перехода строит булево выражение, определяющее переход по альтернативной ветке. STP проверяет набор ограничений на выполнимость и в случае успеха возвращает список присваиваний для входных данных. В соответствии с построенными входными данными один или несколько условных переходов инвертируются и процесс выполнения происходит по ранее не покрытому пути.

Методы класса `Concurrency` вызываются для встретившихся в ходе выполнения операций синхронизации. В их задачи входит сбор информации о выполняющихся потоках, отслеживание блокировок, сигналов и объединение полученных данных в трассу событий. Трасса

событий соответствует синтаксису, который понятен инструменту ThreadSanitizer Offline.

Процесс инструментирования байт-кода в Coffee Machine основывается на статическом подходе. Для этих целей используется библиотека BCEL (Byte Code Engineering Library [12]). Статический подход означает, что инструментирование кода происходит один раз, до процесса выполнения приложения. У используемого подхода есть ряд недостатков. Инструментирования с использованием BCEL происходит медленнее, например, фреймворка для динамической инструментации ASM и требует предварительного определения списка классов для инструментирования, которые могут быть потенциально использованы при выполнении приложения (при динамическом подходе исключается возможность инструментации не использованных классов). Основное преимущество данного подхода в возможности конвертирования инструментированного байт-кода в формат для требуемой виртуальной машины.

В рамках поставленой задачи это означает, что после этапа инструментирования код может быть преобразован в байт-код формата DEX, интерпретируемый виртуальной машиной Dalvik. Таким образом он может исполняться на платформе Android.

Построенная после выполнения приложения трасса событий подаётся на вход инструменту ThreadSanitizer Offline, который осуществляет поиск ошибок синхронизации.

Таким образом, Coffee Machine, в отличие от Java ThreadSanitizer, имеет возможность извлекать трассу событий для Java-приложений на платформе Android.

Одной из существенных проблем является невозможность инструментирования системных библиотек и стандартных синхронизационных методов Java. В связи с этим, класс `Concurrency` отдельно отслеживает вызовы таких методов и производит их частичную симуляцию. Например, при вызове в приложении метода инициализации класса

```
java.util.concurrent.locks.ReentrantReadWriteLock
```

также будет вызван метод

```
a_java_util_concurrent_locks_ReentrantReadWriteLock_
init(Object o)
```

класса `Concurrency` с соответствующим образом переданными ему параметрами. Этот метод сохранит необходимую информацию.

Для выполнения подобных действий могут быть необходимы параметры, передаваемые методу и его возвращаемое значение. Для этого вершина стека виртуальной машины Java, включающая в себя передаваемые параметры и ссылку на объект (в случае, если метод не статический) дублируются дважды, затем происходит вызов метода,

производящего предварительные действия. У этого метода нет возвращаемого значения, а параметры соответствуют входным параметрам целевого метода. Далее происходит вызов целевого метода. Метод, производящий завершающие действия, принимает на вход входные параметры целевого метода и его возвращаемое значение, которое возвращает без изменений. В случае, если для выполнения действий не требуются входные параметры или возвращаемое значение, отдельные шаги процесса могут быть опущены. Схема процесса приведена на рис. 3.

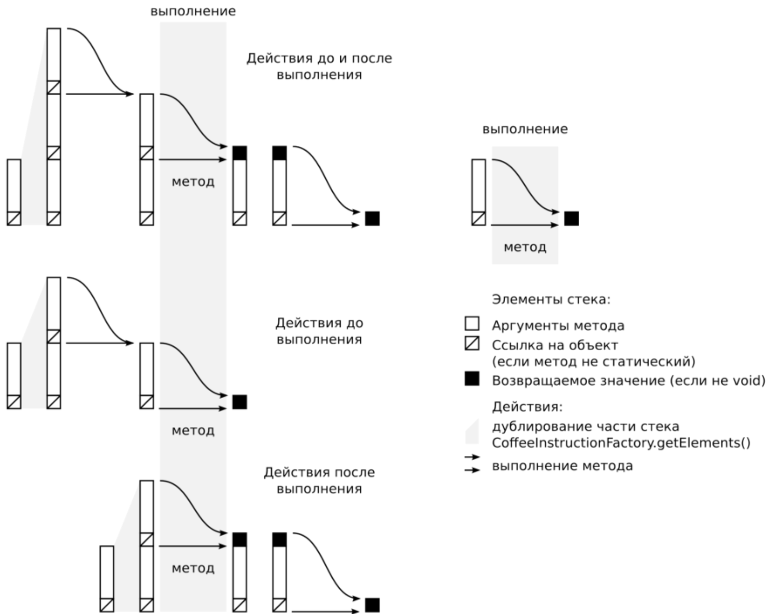


Рис. 3: Схема выполнения подготовки стека для проведения дополнительных действий до и после выполнения метода

4.5. Экспериментальные результаты

Проведение анализа с помощью инструмента Coffee Machine проводилось на наборе тестов, разработанном в рамках Java ThreadSanitizer, а также на основных приложениях операционной системы Android (Browser, Contacts, Mms и некоторые другие). Тесты в составе Java ThreadSanitizer покрывают большую часть API стандартных классов языка Java для синхронизации нескольких нитей (семафоры, барьеры и т. п.), а также дефекты параллельного выполнения, обнаруженные в реальных проектах. В рамках разработки инструмента

Coffee Machine набор тестов был расширен для покрытия специализированных ситуаций использования инструментов синхронизации Java. Для автоматический работы с приложениями операционной системы Android (генерация сценариев взаимодействия) был использована модифицированная версия комплекса инструментов для автоматического тестирования графических приложений GUItar [13, 14]. Такие факторы, как невозможность проведения инструментации непосредственно системных классов, привели к снижению эффективности анализа и увеличению числа ложных срабатываний. Были разработаны дополнительные правила для инструментации кода, использующего конструкции Android Java, позволяющие снизить число ложных срабатываний, связанных с механизмом передачи сообщений, активно используемым при работе с системным интерфейсом. Рассмотренные результаты анализа выявили несколько реальных дефектов синхронизации, которые потенциально могут происходить при разрешении специализированных ситуаций взаимодействия нескольких сервисов. Примером одной из таких ситуаций может завершение работы сервиса, отвечающего за сохранение прикреплённого к электронному письму файла, в приложении Email. Искусственное внедрение простых временных задержек в одну из нитей, работающих параллельно, позволило воспроизвести некорректную ситуацию гонки, которая привела к исключению `NullPointerException` и к аварийному завершению приложения Email. Появление исключительной ситуации требует выполнения инструкций двух нитей в строго определённой последовательности, что практически устраняет возможность выявления ошибки при обычном тестировании.

При проведении анализа набора приложений Android был произведена оценка степени замедления скорости работы, связанного с использованием инструментированного кода, на основе нескольких факторов (степень покрытия интерфейса работы с нитями, механизм генерации и последующей передачи трассы компоненту ThreadSanitizer Offline). Итоговая степень замедления при проведении инструментации исходного кода непосредственно самого приложения не превышает пятикратной отметки, причём основное увеличение приходится на этапы изначальной загрузки компонентов графического интерфейса приложения, в то время как выполнение действий над уже загруженными компонентами затрагивается в меньшем объёме (взаимодействие с приложением укладывается в рамки необходимых временных задержек в комплексе GUITAR, использующихся в модуле эмуляции наборов событий).

5. Заключение

В данной статье рассмотрены общие принципы методов автоматического обнаружения ошибок синхронизации в программном обеспечении, а также особенности данной задачи в контексте

приложений, разработанных для платформы Android. Был проведён краткий обзор методов динамического анализа программ с целью выявления ошибок подобного рода и современных средств, предлагающих реализацию данных методов, для работы с нативным кодом на языке программирования C и работы с кодом на языке Java (при разработке приложений Android используются компоненты как на языке C, так и на языке Java). Для инструментов, производящих анализ нативного кода, приведены основные сложности, связанные с особенностями работы приложений в рамках платформы Android (специфическая системы запуска виртуальной машины Dalvik), и рассмотрен подход к использованию механизма обёртки, необходимый для проведения анализа. В качестве наиболее эффективного среди существующих инструментов выделено средство ThreadSanitizer.

В заключительной части статьи рассматриваются инструменты для анализа компонентов, реализованных на языке Java, и, в частности, инструмент Java ThreadSanitizer. Приведён анализ особенностей платформы Android (виртуальной машины Dalvik), требующих модификации инструмента. Наконец, предложена непосредственное описание реализации инструмента Coffee Machine, использующего подход Java ThreadSanitizer, на основе статической инструментации. Приводится оценка результатов использования полученного инструмента на расширенном наборе тестов для Java ThreadSanitizer и на наборе стандартных приложений платформы Android.

Список литературы

[1]. Официальный анонс выпуска Android 3.0 [HTML] (http://developer.android.com/about/versions/android-3.0-highlights.html)

[2]. L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, 1978.

[3]. R. O’Callahan and J.-D. Choi. Hybrid dynamic data race detection. In PPoPP ’03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 167–178, New York, NY, USA, 2003.

[4]. Julian Seward. Анализ текущего состояния инструмента Valgrind в контексте использования на платформе Android [HTML] (http://blog.mozilla.org/jseward/2011/09/27/valgrind-on-android-current-status)

[5]. U. Banerjee, B. Bliss, Zh. Ma, P. Petersen. Unraveling Data Race Detection in the Intel® Thread Checker. In Proceedings of STMCS ’06. Manhattan, NY, USA, 2006

[6]. Руководство по инструменту Sun Thread Analyzer [HTML] (http://docs.oracle.com/cd/E19205-01/820-4155/tha.html)

[7]. K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. WBIA '09, New York City, NY, USA, 2009

[8]. N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, 2007.

[9]. Примечания к выпускам версий Valgrind [HTML] (http://valgrind.org/docs/manual/dist.news.html)
[10]. Bruneton E. ASM 4.0. A Java bytecode engineering library, 2011 [PDF] (http://download.forge.objectweb.org/asm/asm4-guide.pdf)
[11]. V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In CAV 2007, LNCS 4590, pages 519–531
[12]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)
[13]. Atif Memon. An event-flow model of GUI-based applications for testing. Software Testing, Verification & Reliability. Volume 17 Issue 3, pages 137-157. John Wiley and Sons Ltd. Chichester, UK, 2007
[14]. Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In Proceedings of The 10th Working Conference on Reverse Engineering, Nov. 2003

Automatic concurrency defect detection for Android applications

V. P. Ivannikov, S. P. Vartanov, M. K. Ermakov
ivan@ispras.ru, svartanov@ispras.ru, mermakov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. This paper describes issues related to automatic detection of concurrency defects using dynamic analysis methods with primary target subject as the Android platform. Due to the increased popularity and spread of Android platform and, more importantly, high importance of concurrent work flow for all user applications on this platform, automatic tools for defect detection are likely to be beneficial for many developers. The paper is organised as follows: section 1 provides a brief introduction to the discussed subjects (concurrency basics, Android platform basics). Section 2 discusses theoretical base for a set of dynamic concurrency detection methods. Section 3 contains an overview of several key points in Android work flow related to native code execution in system utilities and in user applications through the JNI mechanisms. This overview is followed by a description of existing tools implementing concurrency defect detection methods for native code and the practical considerations of applying these tools for Android analysis. Section 4 provides the information on Java-oriented tools for concurrency defect detection (Java ThreadSanitizer and Coffee Machine) with the major focus on Coffee Machine, developed within the scope of this project. An overview of Coffee Machine tool functionality and base methods is given, followed by an evaluation of practical application of the tool to Android platform. Section 5 concludes the paper with an evaluation of key issues for both native and Java dynamic analysis for automatic concurrency defect detection.

Keywords: dynamic analysis, concurrency defect detection, Android

References

[1]. Android 3.0 official release statement. [HTML] (http://developer.android.com/about/versions/android-3.0-highlights.html)
[2]. L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 1978. pp. 558–565. doi: 10.1145/359545.359563
[3]. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming 2003. pp. 167–178. doi: 10.1145/781498.781528
[4]. Julian Seward. Valgrind on Android — Current Status. [HTML] (http://blog.mozilla.org/jseward/2011/09/27/valgrind-on-android-current-status)
[5]. U. Banerjee, B. Bliss, Zh. Ma, P. Petersen. Unraveling Data Race Detection in the Intel® Thread Checker. Presented at The First Workshop on Software Tools for Multi-core Systems (STMCS), in conjunction with IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2006.
[6]. Sun Thread Analyzer documentation [HTML] (http://docs.oracle.com/cd/E19205-01/820-4155/tha.html)

[7]. K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. Proceedings of the Workshop on Binary Instrumentation and Applications, 2009. pp. 62-71. doi: 10.1145/1791194.1791203
[8]. Nethercote N., Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. Proceedings of ACM SIGPLAN conference on Programming language design and implementation, 2007. pp. 89-100. doi: 10.1145/1250734.1250746
[9]. Valgrind changelog [HTML] (http://valgrind.org/docs/manual/dist.news.html)
[10]. Bruneton E. ASM 4.0. A Java bytecode engineering library, 2011 [PDF] (http://download.forge.objectweb.org/asm/asm4-guide.pdf)
[11]. V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. Proceedings of the 19th international conference on Computer aided verification, 2007. pp. 519-531
[12]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)
[13]. Atif Memon. An event-flow model of GUI-based applications for testing. Software Testing, Verification & Reliability. Volume 17 Issue 3, 2007. pp. 137-157. doi: 10.1002/stvr.v17:3
[14]. Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In Proceedings of The 10th Working Conference on Reverse Engineering, 2003. pp. 260 – 269. doi: 10.1109/WCRE.2003.1287256

Опыт использования UniTESK как зеркало развития технологий тестирования на основе моделей

Иванников В. П., Петренко А. К., Кулямин В. В., Максимов А. В.
Институт системного программирования РАН (ИСП РАН), Москва
{ivan,petrenko,kuliamin,andrew}@ispras.ru

Аннотация. UniTESK — технология тестирования, основанная на формальных моделях (или спецификациях) требований к поведения программных или аппаратных компонентов. В статье описываются наиболее значимые применения технологии UniTESK в промышленных проектах, суммируется их опыт и оцениваются перспективные направления развития компонентных технологий тестирования на основе моделей.

Ключевые слова: тестирование на основе моделей; спецификации; верификация; автоматизированная генерация тестов

1. Введение

Тестирование на основе моделей (Model Based Testing, MBT) является одной из наиболее интенсивно развивающихся областей программной инженерии. Одна из причин такого активного развития — тот факт, что MBT находится на пересечении нескольких более широких областей. Эти области включают, в частности, методы описания, формализации и моделирования требований, методы анализа формальных моделей, методы статического и динамического анализа кода, методы управления уровнем абстракции и трансформации моделей. Такое положение вещей дает возможность технологиям MBT быстро адаптироваться к последним достижениям в смежных областях. Однако, до сих пор не существует готовых для промышленного использования инструментов MBT, которые можно было бы применять с достаточной эффективностью в широком классе проектов разработки и тестирования. Для дальнейшего развития технологий MBT необходим анализ причин такой ситуации, основанный на опыте их использования в последние 15-20 лет. Это поможет выявить проблемы существующих технологий и увидеть возможные пути их решения. В данной статье мы кратко описываем этапы развития технологии UniTESK (UNIfied TEsting and Specification toolKit,

унифицированный набор инструментов спецификации и тестирования) как одной из первых технологий, нацеленных на применение для тестирования широкого класса систем. По ходу статьи мы выделим позитивные и негативные итоги использования этой технологии. Эта статья является расширенным вариантом работы [36] и представляет собой анализ опыта промышленного использования технологий и инструментов, и поэтому не содержит постановок научных проблем или изложения новых техник и методов решения каких-либо задач. Тем не менее, проведенный анализ может быть полезен исследователям в области MBT.

Технология UniTESK [1,2] была создана на основе опыта, полученного при разработке системы автоматизированного тестирования KVEST [3] для ядра операционной системы реального времени. Работа над KVEST началась в 1994 году, когда термин «тестирование на основе моделей» еще не использовался. Этот термин появился в самом начале XXI столетия. Сейчас MBT очень быстро развивается. Технологии MBT поддерживаются многими энтузиастами и само понятие «тестирования на основе моделей» интерпретируется по-разному. Для аккуратного анализа места UniTESK среди других подходов нужно сначала определить, что подразумевается под этим термином в рамках технологии UniTESK.

Обычно говорят, что тестирование на основе моделей использует различные модели для построения тестов. Однако, у подобного утверждения очень много различных толкований — большинство исследователей и инженеров, использующих MBT на практике, подразумевают под этим термином более специфические техники тестирования. Первый водораздел — что именно является моделируемым объектом: некоторые моделируют поведение тестируемой системы (system under test, SUT), другие — окружение тестируемой системы, в частности, может моделироваться поведение тестов или тестовой системы как части этого окружения. В UniTESK моделируется поведение тестируемой системы. В рамках такого подхода выделяются методы MBT, использующие различные способы описания поведения. О первом из этих способов Ян Пелеска (Jan Peleska) говорит [4]: «поведение SUT задается моделью, которая представлена в том же стиле, как и модели, используемые для разработки». Такие спецификации или модели называются *исполнимыми*. Роль исполнимой модели может играть прототип реализуемой системы или другая модель, для которой есть понятие выполнения, например, конечный автомат, сеть Петри, машина с абстрактным состоянием (ASM) [5]. Примерами моделей другого вида, *неисполнимых*, служат алгебраические спецификации, а также программные контракты в виде пред- и постусловий функций. Различные виды моделей имеют свои достоинства и недостатки, более или менее подходят для тестирования систем различных типов. Так, при генерации тестов нужно строить не только тестовые данные и последовательности вызовов различных операций, но и другие артефакты, например, тестовые оракулы — компоненты тестов, автоматически оценивающие результаты работы SUT и выносящие решение, соответствуют

ли они требованиям или нет. Исполнимые модели не всегда подходят для генерации оракулов, но достаточно эффективно используются для построения тестовых последовательностей. Программные контракты упрощают генерацию оракулов, но для построения тестовых последовательностей не так удобны. Другими словами, для эффективного построения всех элементов тестовой системы лучше использовать модели различных типов. Возвращаясь к UniTESK, отметим, что основной вид моделей, используемый в этой технологии — это программные контракты. Однако, для генерации нетривиальных тестовых последовательностей в ней используются автоматные модели.

Технология UniTESK может быть реализована в рамках различных систем разработки и на различных языках. Сейчас используются реализации подхода UniTESK на основе языков C, C++, Java, Python, это, соответственно, инструменты CTESK, C++TESK, JavaTESK и PyTESK [6].

UniTESK является исследовательской разработкой ИСП РАН. Инструменты UniTESK доступны под свободной лицензией. Вместе с тем, имеется опыт промышленного использования этой технологии. Некоторые тестовые наборы, созданные с помощью UniTESK, используются для сертификации и аттестации промышленного программного обеспечения (ПО). Примером является тестовый набор OLVER [7] — один из самых больших тестовых наборов, созданных при помощи технологий MBT, уступающий лишь системе тестов, разработанной в рамках Инициативы по поддержке взаимодействия Microsoft (Microsoft Interoperability Initiative) [8].

2. Обзор применений UniTESK

Рассмотрим наиболее интересные примеры использования технологии UniTESK и полученные в рамках этого использования уроки.

Первым применением UniTESK был поддержанный Microsoft Research проект по разработке тестового набора для реализации IPv6 [9]. Проект стартовал в 2000 году. Тогда UniTESK только начинал создаваться, поэтому пришлось использовать облегченную реализацию технологии на языке C — CTESK-light. Несмотря на нестабильность инструмента удалось построить эффективный тестовый набор, который нашел дефекты, ранее не обнаруженные другими тестовыми наборами. Это был первый опыт использования контрактных спецификаций для тестирования телекоммуникационных протоколов. Было показано, что контрактные спецификации в сочетании с техникой тестирования систем с асинхронными интерфейсами, разработанной в рамках UniTESK [10,11] позволяют строить эффективные тесты. Эти тесты выявили больше ошибок и требовали меньше усилий для создания и сопровождения при заданном уровне качества тестирования, чем тесты, построенные по традиционным технологиям. Вместе с тем, опыт тестирования протоколов показал, что для эффективной генерации

последовательностей тестовых воздействий при тестировании протоколов полезно иметь и исполнимые модели.

Одним из первых опытов использования UniTESK для тестирования программных компонентов Java [12] через программный интерфейс (Application Programming Interface, API) стал проект тестирования одной из реализаций стандартной библиотеки поддержки времени исполнения Java. Разработка моделей и тестов не вызывала особых проблем, так как интерфейсы были хорошо документированы. Помимо интерфейсов на Java в системе также были интерфейсы на C++, но больших проблем и это не вызвало, поскольку архитектура тестов UniTESK предусматривает слой тестовых адаптеров. Более серьезные проблемы появились, когда началось собственно тестирование, в рамках которого генератор тестовой последовательности должен был работать на базе самой тестируемой системы, еще не стабильной в это время.

Одним из значимых примеров применения UniTESK на платформе Java является проект тестирования инфраструктуры распределенной информационной системы одного из крупных операторов мобильной связи, который продолжается и сейчас. Возможность формальной и строгой фиксации интерфейсов компонентов этой чрезвычайно большой и разнородной системы стала для заказчика самым главным преимуществом UniTESK по сравнению с другими инструментами функционального тестирования. В рамках проекта были формально специфицированы и протестированы сотни компонентов. Уже к концу первого года применения технологии положительный эффект проявился в ускорении сроков интеграции новых версий распределенной системы. Вместе с тем, вскрылась серьезная проблема. Если в предыдущих проектах применения UniTESK требования к большей части интерфейсов определялись стандартами или другими тщательно разработанными документами, здесь уровень документирования часто оказывался недостаточным для построения консистентных спецификаций. Восстановление документации или требований к интерфейсам в системах такого размера оказывается практически неразрешимой задачей, что не позволяет использовать MBT в полном объеме. О путях решения этой проблемы будет кратко сказано в Заключении.

Самым крупным примером применения UniTESK стал проект OLVER (Open Linux VERification) [7], который проводился в 2005-2007 годах при поддержке Министерства образования и науки РФ. Целью проекта была формальная спецификация интерфейсов стандарта Linux Standard Base (LSB), точнее его центральная части LSB Core. В LSB Core входят наиболее важные библиотеки операционной системы Linux, которые в значительной части реализуют стандарт POSIX. Строгое описание стандарта LSB и наличие набора тестов, который мог бы качественно проверить соответствие разных реализаций библиотек Linux требованиям стандарта, являются необходимыми условиями для обеспечения переносимости приложений для Linux с одного дистрибутива на другой. Проблема переносимости приложений под Linux является крайне

острой, поскольку сейчас доступно уже несколько сотен различных дистрибутивов. Результаты проекта опубликованы [7]. Были построены контрактные спецификации более чем 1500 интерфейсов на языке C. Инструментом моделирования и генерации тестов был выбран CTESK. В ходе проекта были выявлены проблемы в самих стандартах: LSB (ISO/IEC 23360) и The Single UNIX Specification, основную часть которого составляет стандарт POSIX.1 (он же IEEE Std 1003.1, он же ISO/IEC 9945, он же The Open Group Base Specifications Issue 6). Разработанный тестовый набор включен в пакет сертификационных тестов международного консорциума The Linux Foundation [13].

Опыт формализации интерфейсов большого индустриального стандарта и разработки тестового набора для него дал много полезных уроков. Одним из них стало понимание важности организации информационного и методического обеспечения такого проекта. Массив документации и исходных текстов библиотек Linux, особенно с учетом многочисленности версий и вариантов, связанных с различными аппаратными платформами, является огромным. Кроме того, в разработку собственно стандарта и в разработку его реализаций вовлечены тысячи людей, распределенных по всему миру. Из этого следует, что организация «документооборота» является одной из важнейших составляющих проектов такого калибра. В организационно-методическом плане мы столкнулись с тем, что обучение новых сотрудников и контроль за качеством спецификаций и тестов требует много усилий и при этом быстро достичь необходимого уровня профессионализма невозможно. То есть, масштабируемость проектов по использованию MBT в плане расширения числа участников проекта, задействованных в самой работе по спецификации требований и отладке тестов, — это одна из самых сложных проблем, мешающая широкому внедрению MBT.

Одной из методических проблем является выбор уровня абстракции, на котором строится модель. Более абстрактные модели или разделение моделей на два-три слоя, отличающиеся уровнем абстракции, упрощают задачу повторного использования моделей и тестов, при этом общий размер системы возрастает, и ее сложность также растет. В долгосрочном плане выгоднее иметь многослойные модели, в краткосрочном — модели, близкие по уровню детальности к реализации, конечно, если реализация уже есть. Профессиональный опытный верификатор умеет находить баланс между абстрактным описанием поведения, например, файловой системы и особенностями, деталями интерфейса конкретной реализации файловой системы. В UniTESK имеется специальная поддержка разделения уровней абстракции. В частности, специфика конкретных интерфейсов может быть скрыта в слой адаптеров. Выбор баланса во многом определяется долгосрочными планами по использованию и развитию моделей и тестового набора. То есть, такого рода работа требует достаточно широкого кругозора, чего трудно требовать то обычных инженеров-тестировщиков.

Результаты проекта OLVER впоследствии были использованы в разработке тестового набора для российской операционной системы реального времени ОС 2000/3000 [14]. Эта система поддерживает две группы интерфейсов. Первая отвечает требованиям POSIX, вторая — требованиям ARINC 653, международного стандарта для критических встроенных систем. Выделение уровня адаптеров, разделяющего модельное и реализационное представление интерфейсов, заложенное в архитектуру UniTESK, существенно упростило повторное использование OLVER в данном проекте.

Одновременно с началом работ по OLVER были развернуты работы по применению UniTESK для имитационного тестирования моделей отдельных блоков микропроцессоров [15]. Полученные тесты использовались при разработке российских микропроцессоров с архитектурой MIPS и микропроцессоров с элементами VLIM/EPIC. Размер типовых блоков в таких микропроцессорах — несколько миллионов вентилей. Для целей спецификации и генерации тестов не потребовалось больших изменений в инструментах, за основу был взят CTESK. В техническом плане привязка CTESK к API на языке C не стала проблемой, так как большинство симуляторов, работающих с языками моделирования логики микропроцессоров (High Level Design languages, HLD), например, VHDL или Verilog, предоставляют удобный интерфейс для взаимодействия с программами на C. Несколько изменилась семантика предусловий в контрактных спецификациях, они стали описывать не столько разрешенную область входных данных, сколько условия возможности выполнения микрооперации на соответствующем такте, что, кстати, характерно и для семантики предусловий асинхронных событий при тестировании параллелизма. Так же, как и в случае моделирования протоколов, выявилась потребность в использовании наряду с постусловиями, явных моделей поведения тестируемого устройства.

Как и в проектах по верификации программных систем одной из главных проблем, мешающих внедрению MBT при верификации аппаратного обеспечения (как и большинства других методов верификации), является отсутствие четких и детальных описаний функциональных требований к компонентам. Вместе с тем, ситуация в разработке микропроцессоров несколько лучше, так как в ее ходе принято наряду с HLD моделями строить и системные или архитектурные модели, описывающие семантику набора инструкций. Элементы таких архитектурных моделей можно использовать для восполнения недостающих знаний о поведении некоторых блоков микропроцессоров [16]. Хорошей новостью оказалось достаточно простое решение задачи распараллеливания выполнения теста на кластерах. Типичные размеры конечного автомата, который генерируется при выполнении теста одного сложного блока микропроцессора — это несколько миллионов узлов и десятки миллионов переходов. Оказалось, что генерация теста с помощью обхода неизвестного конечного автомата хорошо распараллеливается на кластерах до 200 узлов, с накладными расходами всего лишь 10-15%. При

этом надо помнить, что при имитационном тестировании высокоуровневой модели устройства основное время уходит на работу симулятора HDL, т.е. такая масштабируемость связана с возможностью запустить на каждом узле отдельный симулятор и выполнять параллельно много действий в различных состояниях. Масштабируемость распараллеливания тестирования систем других типов, скорее всего, будет не такой высокой.

Важно отметить задачи верификации, которые не удалось свести к моделированию на основе контрактных спецификаций, из-за чего они дали толчок для разработки новых методов MBT. В первую очередь следует упомянуть задачу тестирования компиляторов и задачу тестирования микропроцессора в целом, так называемый «core testing». В обоих случаях это задачи системного тестирования, где требуется подавать на вход большого «черного ящика» тестовые воздействия (в нашем случае это тестовые программы, которые подаются на вход компилятору или загружаются в память симулятора микропроцессора), но при этом интересно тестировать не все подряд, а лишь некоторые заданные режимы или заданную группу модулей компилятора или процессора. В случае тестирования компиляторов был разработан инструмент OTK, который использовался для тестирования оптимизирующих компиляторов Intel и Simulink [17,18]. Он позволяет нацеливаться на заданные виды оптимизаций. Для системного тестирования микропроцессоров был разработан инструмент MicroTESK [19,20,21]. Основной задачей этого инструмента была проверка разнообразных ситуаций, связанных с наиболее сложными подсистемами управления памяти: блоком трансляции адресов, кэшами разных уровней или блоком управления памятью в целом.

3. Выводы и направления дальнейшего развития

Начнем с позитивных выводов.

3.1. Позитивные выводы по современному состоянию MBT

- История применений UniTESK в промышленных разработках подтверждает мировой опыт [22]: MBT можно эффективно применять в промышленных проектах, при этом по сравнению с традиционным тестированием этот подход дает уникальное преимущество — удается находить достаточно много ошибок в требованиях, которые часто существенно дороже ошибок в реализации.
- Удается достичь уровень тестового покрытия существенно выше традиционного (даже в сравнении с тестированием «белого ящика»). Так, в случае применения OTK для тестирования компилятора gcc удалось довести уровень покрытия до 95%, а в случае компилятора

Intel до 75%, что существенно превышало уровень покрытия? полученного традиционными тестами [18].

- Хотя многослойная структура спецификаций (несколько уровней абстракции) редко используется на практике, явное выделение слоя адаптеров упрощает перенос и сопровождение, и, наоборот, его отсутствие существенно усложняет разработку тестовых наборов, что было показано в программе Microsoft Interoperability Initiative [23].
- Генерация тестовых последовательностей в виде обхода неявно заданного конечного автомата хорошо распараллеливается и позволяет использовать вычислительные мощности кластеров при накладных расходах в 10-15%, по крайней мере в случае тестирования моделей микропроцессоров.
- Растет востребованность MBT в области критических систем и приложений. Это, в частности, находит отражение в стандартах, определяющих требования к процессам разработки таких систем, например, в DO178C [24] и в Common Criteria [25].

3.2. Негативные аспекты современного состояния дел в области MBT

- Самая главная трудность, препятствующая широкому распространению MBT — это отсутствие достаточно четких и детальных моделей (спецификаций). Такое положение дел часто является не только следствием недостаточного внимания к разработке спецификаций или экономии средств, часто основная причина его — отсутствие грамотных специалистов, которые одновременно владеют предметной областью и в состоянии построить модель, необходимую для построения тестов.
- Чаще всего в MBT используются модели, создаваемые специально для дальнейшего построения тестов на их основе. Однако из-за постепенного распространения разработки на основе моделей (Model-Driven Development, MDD) на руках разработчиков возникают другие модели. Создание особых моделей для тестирования при этом воспринимается как неприемлемые дополнительные расходы. Оказывается, что при тщательном планировании многие компоненты моделей разработки можно использовать и для генерации тестов, что показано, например, в диссертации М. М. Чупилко [16]. Однако получение реальной пользы от использования моделей разработки при тестирования требует дополнительной квалификации и тщательного анализа получаемых результатов — иначе чаще всего тесты, созданные по тем же лекалам, что и тестируемая система, просто не в состоянии обнаружить в ней никаких ошибок.

- Двуязычные системы генерации тестов типа первых версий UniTESK и SpecExplorer [26], а также специальные спецификационные нотации, даже приближенные к языкам программирования, например, JML [27], затрудняют внедрение технологий MBT. Двуязычные нотации требуют специальной подготовки персонала и затрудняют развитие созданных на их основе тестовых наборов. При этом современные ОО языки уже имеют развитые средства, которые позволяют описывать спецификации средствами базового языка [26-31].

3.3. Направления развития

- Для обеспечения высокой эффективности использования инструментов MBT при тестировании разнообразных систем нужно обеспечивать разнообразие поддерживаемых ими парадигм моделирования, в частности, контрактные спецификации совместно с, например, конечными автоматами или структурами Крипке. Каждый из видов моделей нацелен на облегчение анализа лишь определенных аспектов поведения системы и может затруднять анализ других. К сожалению, чаще всего наиболее выгодно использование разных моделей на разных уровнях абстракции, и все их приходится создавать вручную, без возможности автоматизации перехода от одной модели к другой. Любые продвижения в направлении автоматизации таких сложных, связанных с изменением уровня абстракции, трансформаций моделей, могли бы оказать значительную помощь в практическом использовании основанных на них методов MBT.
- Требуется развитие средств моделирования и описания спецификаций для целей MBT. Хотя наблюдается прогресс в области технологий разработки с использованием проблемно-ориентированных языков (Domain Specific Languages, DSL), в практическом плане системы, базирующиеся на универсальных языках выигрывают за счет наличия большого числа специалистов, владеющих такими языками, и широкой их поддержки разнообразными инструментами. То же можно сказать и про моноязычные системы — они выигрывают у мультиязычных за счет большей простоты использования и развития.
- Современные достижения в области статического и динамического анализа позволяют интегрировать эти техники в системы MBT, причем объектом анализа должны быть как модели, так и программный код тестируемой системы.
- Для преодоления проблем, связанных с катастрофическим недостатком детальных спецификаций в реальной практике, нужно развивать и внедрять инструменты работы с требованиями и моделями (см., например, [33]), в частности, с моделями программно-

аппаратных систем [34,35]. Для многокомпонентных систем нужно интегрировать MBT со средствами выявления и реинжиниринга архитектурных и проектных решений.

Литература

[1]. Bourdonov, I.B., Kossatchev, A.S., Kuliamin, V.V., Petrenko, A.K.: UniTesK Test Suite Architecture. In: FME 2002. LNCS, vol. 2391, pp. 77-88. Springer-Verlag (2002)

[2]. Кулямин, В.В., Петренко, А.К., Косачев, А.С., Бурдонов, И.Б.: Подход UniTesK к разработке тестов. Программирование, 29(6), 25-43 (2003)

[3]. Bourdonov, I.B, Kossatchev, A.S., Petrenko, A.K., Galter. D.: KVEST: Automated Generation of Test Suites from Formal Specifications. In: Proceedings of Formal Method Congress, Toulouse, France, 1999. LNCS, vol. 1708, pp. 608-621 (1999)

[4]. Peleska, J.: Industrial-Strength Model-Based Testing – State of the Art and Current Challenges. Invited Talk. In: Petrenko, A.K., Schlingloff, H. (eds.) Proceedings Eighth Workshop on Model-Based Testing (MBT 2013), Rome, Italy, 17th March 2013. Electronic Proceedings in Theoretical Computer Science, 111, pp. 3–28. DOI: 10.4204/EPTCS.111.1 (2013)

[5]. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag (2003)

[6]. UniTESK technology, http://unitesk.ispras.ru

[7]. OLVER project, http://linuxtesting.org

[8]. Microsoft Interoperability Initiative, http://www.microsoft.com/openspecifications

[9]. Пакулин, Н.В., Хорошилов, А.В.: Разработка формальных моделей и тестирование соответствия для систем с асинхронными интерфейсами и телекоммуникационных протоколов. Программирование, 33 (6), 26-55 (2007)

[10]. Хорошилов, А.В.: Спецификация и тестирование компонентов с асинхронными интерфесами. Диссертация на степень к.ф.-м.н., Москва (2006)

[11]. Kuliamin, V.V., Petrenko, A.K., Pakulin, N.V.: Extended Design-by-Contract Approach to Specification and Conformance Testing of Distributed Software. In: Proceedings of WMSCI'2005, Orlando, USA, July 10-13, 2005. Model Based Development and Testing, v. VII, pp. 65-70 (2005)

[12]. Bourdonov, I.B., Demakov A.V., Jarov, A.A., Kossatchev, A.S., Kuliamin, V.V., Petrenko, A.K., Zelenov, S.V.: Java Specification Extension for Automated Test Development. In: Proceedings of PSI'01. LNCS, vol. 2244, pp. 301-307. Springer-Verlag (2001)

[13]. The Linux Foundation consortium. LSB certification test suite, http://ispras.linuxbase.org/index.php/LSB_Certification_System

[14]. Maksimov, A.V.: Requirements-based conformance testing of ARINC 653 real-time operating systems. In: Proceedings of the Data Systems In Aerospace (DASIA 2010) conference, 2010. ESA SP-682, ISBN 978-92-9221-246-9 (2010)

[15]. Иванников, В.П., Камкин, А.С., Косачев, А.С., Кулямин, В.В., Петренко, А.К.: Использование контрактных спецификаций для представления требований и функционального тестирования моделей аппаратуры. Программирование, 33(5), 47-61 (2007).

[16]. Chupilko, M.M.: Developing Test Systems of Multi-Modules Hardware Designs. ISSN 0361-7688, Programming and Computer Software, 2012, Vol. 38, No. 1, pp. 34-42. Pleiades Publishing, Ltd. (2012)

[17]. Zelenov, S.V., Zelenova, S.A.: Model-Based Testing of Optimizing Compilers. In: Proc. of the 19th IFIP TC6/WG6.1 International Conference on Testing of Software and Communicating Systems – 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007). LNCS, vol. 4581, pp. 365-377. Springer-Verlag, Berlin (2007)
[18]. Zelenov, S.V., Silakov, D.V., Petrenko, A.K., Conrad, M., Fey I.: Automatic Test Generation for Model-Based code Generators. In: IEEE ISoLA 2006 Second Intern. Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Paphos, Cyprus, pp. 68-75 (2006)
[19]. Камкин, А.С.: Метод автоматизации имитационного тестирования микропроцессоров конвейерной архитектуры на основе формальных спецификаций. Диссертация на степень к.ф.-м.н., Москва (2009)
[20]. Корныхин, Е.В.: Метод автоматизации генерации тестовых программ для верификации MMU. Диссертация на степень к.ф.-м.н., Москва (2010)
[21]. Kamkin, A.S., Tatarnikov, A.: MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. In: Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2012), May 30-31, 2012, Perm, Russia (2012)
[22]. MBT survey, http://www.robertvbinder.com/docs/arts/MBT-User-Survey.pdf
[23]. Grieskamp, W.: Microsoft's Protocol Documentation Program: A Success Story for Model-Based Testing. In: Testing – Practice and Research Techniques. Lecture Notes in Computer Science, vol. 6303, p. 7 (2010)
[24]. Adams, C.: Safety-critical software for mission-critical applications to get boost with release of DO-178C. Military & Aerospace Electronics, 10 (2010)
[25]. Common Criteria, http://www.commoncriteriaportal.org
[26]. SpecExplorer, http://research.microsoft.com/en-us/projects/specexplorer
[27]. The Java Modelling Language (JML), http://www.eecs.ucf.edu/~leavens/JML/index.shtml
[28]. Pakulin, N.V.: Integrated Modular Avionics: New Challenges for MBT. In: ETSI TTCN-3 User Conference and Model Based Testing Workshop, Bangalore, India, 11-14 June 2012 (2012)
[29]. Code Contracts, http://research.microsoft.com/en-us/projects/contracts
[30]. C++TESK, http://forge.ispras.ru/projects/cpptesk-toolkit
[31]. Кулямин, В.В.: Компонентная архитектура среды для тестирования на основе моделей. программирование, 36(5), 54-75 (2010)
[32]. Kuliamin, V.V.: Multi-paradigm Models as Source for Automated Test Construction. In: Proceedings of the 1-st Workshop on Model Based Testing (MBT'2004, in ETAPS'2004), Barcelona, Spain, March 27-38, 2004, Electronic Notes in Theoretical Computer Science, 111:137-160, Elseveir, (2005)
[33]. ReQuality tool, http://requality.org/en/doc.en.html
[34]. Khoroshilov, A.V., Albitskiy, D., Koverninskiy, I.V., Olshanskiy, M.Yu., Petrenko, A.K., Ugnenko, A.A.: AADL-Based Toolset for IMA System Design and Integration. SAE Int. J. Aerosp. 5(2), DOI:10.4271/2012-01-2146 (2012)
[35]. Systems Modeling Language (SysML), http://www.sysml.org
[36]. Petrenko, A.K., Kuliamin, V.V., Maksimov A.V.: UniTESK: Component Model Based Testing. Proceedings of ICTERI 2013.

How the story of UniTESK technology applications mirrors development of model based testing

Ivannikov V.P., Petrenko A.K., Kuliamin V.V., Maksimov A.V.
ISP RAS, Moscow, Russia
{ivan,petrenko,kuliamin,andrew}@ispras.ru

Abstract. UniTESK (UNIfied TEsting and Specification toolKit) is a testing technology based on formal models (or specifications) of requirements to behavior of software or hardware components. It was created with experience gained during development of the framework for automated testing of a real-time operating system kernel during 1994-2000. Software contracts were the main kind of models in the first version of the technology. Automata models were also implemented to support generation of complicated test sequences. UniTESK was first used in 2000 in the development of the test suite for IPv6 implementation. It was the first experience of using contract specifications in testing of implementations of telecommunication protocols. This project demonstrated that contract specifications in combination with the technique of testing software systems with asynchronous interface developed within UniTESK are very effective. Applications of UniTESK to testing software components in Java include the project on testing of implementation of standard library for Java runtime support and the project on testing of infrastructure of information system for one of large mobile operators in Russia. The most significant application of UniTESK happened in 2005-2007 in the Open Linux Verification project. Formal specifications and tests for interfaces of the Linux Standard Base were created during the project. These results then were used in development of the test suite for Russian avionics real-time operating system.

Positive lessons learned during development and using UniTESK include effective application of model based testing methods in large industrial projects, high level of test coverage achieved by the generated tests, applicability of model based testing to critical systems and applications. Negative lessons include the lack of well-defined and detailed models and specifications, extra development expenses caused by creation of test specific models, problems with introduction of model based testing technologies using bilingual test generation systems and specific notations.

Keywords: model based testing, specification, verification, automated test generation

References

[1]. Bourdonov, I.B., Kossatchev, A.S., Kuliamin, V.V., Petrenko, A.K.: UniTesK Test Suite Architecture. In: FME 2002. LNCS, vol. 2391, pp. 77-88. Springer-Verlag (2002)

[2]. V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, I. B. Burdonov: The UniTesK Approach to Designing Test Suites. Programming and Computer Software, 29(6), 310-322 (2003)

[3]. Bourdonov, I.B, Kossatchev, A.S., Petrenko, A.K., Galter. D.: KVEST: Automated Generation of Test Suites from Formal Specifications. In: Proceedings of Formal Method Congress, Toulouse, France, 1999. LNCS, vol. 1708, pp. 608-621 (1999)

[4]. Peleska, J.: Industrial-Strength Model-Based Testing – State of the Art and Current Challenges. Invited Talk. In: Petrenko, A.K., Schlingloff, H. (eds.) Proceedings Eighth Workshop on Model-Based Testing (MBT 2013), Rome, Italy, 17th March 2013. Electronic Proceedings in Theoretical Computer Science, 111, pp. 3–28. DOI: 10.4204/EPTCS.111.1 (2013)

[5]. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag (2003)

[6]. UniTESK technology, http://unitesk.ispras.ru

[7]. OLVER project, http://linuxtesting.org

[8]. Microsoft Interoperability Initiative, http://www.microsoft.com/openspecifications

[9]. N. V. Pakulin, A. V. Khoroshilov: Development of formal models and conformance testing for systems with asynchronous interfaces and telecommunications protocols. Programming and Computer Software, 33 (6), 316-335 (2007)

[10]. A. V. Khoroshilov: Specifikacija i testirovanie komponentov s asinhronnymi interfesami [Specification and testing of components with asynchronous interfaces]. Dissertacija na stepen' k.f.-m.n. [PhD Thesis], Moskva (2006)

[11]. Kuliamin, V.V., Petrenko, A.K., Pakulin, N.V.: Extended Design-by-Contract Approach to Specification and Conformance Testing of Distributed Software. In: Proceedings of WMSCI'2005, Orlando, USA, July 10-13, 2005. Model Based Development and Testing, v. VII, pp. 65-70 (2005)

[12]. Bourdonov, I.B., Demakov A.V., Jarov, A.A., Kossatchev, A.S., Kuliamin, V.V., Petrenko, A.K., Zelenov, S.V.: Java Specification Extension for Automated Test Development. In: Proceedings of PSI'01. LNCS, vol. 2244, pp. 301-307. Springer-Verlag (2001)

[13]. The Linux Foundation consortium. LSB certification test suite, http://ispras.linuxbase.org/index.php/LSB_Certification_System

[14]. Maksimov, A.V.: Requirements-based conformance testing of ARINC 653 real-time operating systems. In: Proceedings of the Data Systems In Aerospace (DASIA 2010) conference, 2010. ESA SP-682, ISBN 978-92-9221-246-9 (2010)

[15]. V. P. Ivannikov, A. S. Kamkin, A. S. Kossatchev, V. V. Kuliamin, A. K. Petrenko: The use of contract specifications for representing requirements and for functional testing of hardware models. Programming and Computer Software, 33(5), 272-282 (2007).

[16]. Chupilko, M.M.: Developing Test Systems of Multi-Modules Hardware Designs. ISSN 0361-7688, Programming and Computer Software, 2012, Vol. 38, No. 1, pp. 34-42. Pleiades Publishing, Ltd. (2012)

[17]. Zelenov, S.V., Zelenova, S.A.: Model-Based Testing of Optimizing Compilers. In: Proc. of the 19th IFIP TC6/WG6.1 International Conference on Testing of Software and Communicating Systems – 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007). LNCS, vol. 4581, pp. 365-377. Springer-Verlag, Berlin (2007)

[18]. Zelenov, S.V., Silakov, D.V., Petrenko, A.K., Conrad, M., Fey I.: Automatic Test Generation for Model-Based code Generators. In: IEEE ISoLA 2006 Second Intern. Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Paphos, Cyprus, pp. 68-75 (2006)

[19]. A. S. Kamkin: Metod avtomatizacii imitacionnogo testirovanija mikroprocessorov konvejernoj arhitektury na osnove formal'nyh specifikacij [Method for automating simulation testing microprocessor pipeline architecture based on formal specifications]. Dissertacija na stepen' k.f.-m.n. [PhD Thesis], Moskva (2009)

[20]. E. V. Kornykhin: Metod avtomatizacii generacii testovyh programm dlja verifikacii MMU [Method for automating simulation testing method microprocessors automation test program generation for verification of MMU]. Dissertacija na stepen' k.f.-m.n. [PhD Thesis], Moskva (2010)

[21]. Kamkin, A.S., Tatarnikov, A.: MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. In: Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2012), May 30-31, 2012, Perm, Russia (2012)

[22]. MBT survey, http://www.robertvbinder.com/docs/arts/MBT-User-Survey.pdf

[23]. Grieskamp, W.: Microsoft's Protocol Documentation Program: A Success Story for Model-Based Testing. In: Testing – Practice and Research Techniques. Lecture Notes in Computer Science, vol. 6303, p. 7 (2010)

[24]. Adams, C.: Safety-critical software for mission-critical applications to get boost with release of DO-178C. Military & Aerospace Electronics, 10 (2010)

[25]. Common Criteria, http://www.commoncriteriaportal.org

[26]. SpecExplorer, http://research.microsoft.com/en-us/projects/specexplorer

[27]. The Java Modelling Language (JML), http://www.eecs.ucf.edu/~leavens/JML/index.shtml

[28]. Pakulin, N.V.: Integrated Modular Avionics: New Challenges for MBT. In: ETSI TTCN-3 User Conference and Model Based Testing Workshop, Bangalore, India, 11-14 June 2012 (2012)

[29]. Code Contracts, http://research.microsoft.com/en-us/projects/contracts

[30]. C++TESK, http://forge.ispras.ru/projects/cpptesk-toolkit

[31]. V. V. Kuliamin: Component architecture of model-based testing environment. Programming and Computer Software, 36(5), 289-305 (2010)

[32]. Kuliamin, V.V.: Multi-paradigm Models as Source for Automated Test Construction. In: Proceedings of the 1-st Workshop on Model Based Testing (MBT'2004, in ETAPS'2004), Barcelona, Spain, March 27-38, 2004, Electronic Notes in Theoretical Computer Science, 111:137-160, Elseveir, (2005)

[33]. ReQuality tool, http://requality.org/en/doc.en.html

[34]. Khoroshilov, A.V., Albitskiy, D., Koverninskiy, I.V., Olshanskiy, M.Yu., Petrenko, A.K., Ugnenko, A.A.: AADL-Based Toolset for IMA System Design and Integration. SAE Int. J. Aerosp. 5(2), DOI:10.4271/2012-01-2146 (2012)

[35]. Systems Modeling Language (SysML), http://www.sysml.org

[36]. Petrenko, A.K., Kuliamin, V.V., Maksimov A.V.: UniTESK: Component Model Based Testing. Proceedings of ICTERI 2013.

Введение в метод CEGAR — уточнение абстракции по контрпримерам

Мандрыкин М. У., Мутилин В. С., Хорошилов А. В.
mandrykin@ispras.ru, mutilin@ispras.ru, khoroshilov@ispras.ru

Аннотация. Точность, полнота и масштабируемость применяемых на практике инструментов статической верификации значительно возросла за последнее десятилетие. В частности, успешный автоматизированный анализ программных систем среднего размера с использованием проверки моделей, получаемых при помощи предикатной абстракции, стал возможен благодаря развитию метода уточнения абстракции по контрпримерам — CEGAR (Counter Example Guided Abstraction Refinement). Этот метод так или иначе используется в таких инструментах, как SLAM, BLAST, SATABS и CPAchecker. В рамках данной статьи мы рассмотрим метод CEGAR в том виде, как он реализован в инструментах статической верификации BLAST и CPAchecker.

Ключевые слова: статическая верификация, предикатная абстракция, проверка моделей, уточнение абстракции по контрпримерам, интерполяция Крейга, крупноблочное кодирование.

1 Введение

По мере того, как растет сложность программных систем и их роль в жизни общества, растет и потребность в эффективных методах проверки их корректности. Выделяют две основные группы методов проверки корректности программ: динамические методы (или методы тестирования) и статические.

Динамические методы выполняют проверку корректности программы на основе результатов наблюдения за её поведением в ходе работы в реальном или модельном окружении. Этот подход позволяет обнаружить только те ошибки, которые проявляются в ситуациях, воспроизводимых в процессе тестирования, поэтому качество проверки во многом и определяется тем, какие ситуации удается воспроизвести. Поскольку даже в умеренно сложной программе число различных ситуаций, потенциально приводящих к ошибкам, таково, что протестировать все их за разумное время не представляется возможным, то динамические методы нацелены скорее на обнаружение как можно большего числа ошибок, чем на доказательство их полного отсутствия. Опыт использования программных систем подтверждает этот факт примерами

печальных последствий вовремя не обнаруженных ошибок в программном обеспечении [1], в случае которых тщательное тестирование было либо трудноосуществимо, либо эти ошибки не выявило.

Статические методы выполняют анализ корректности исходного или бинарного кода программы без его выполнения и потенциально имеют возможность для обнаружения всех ошибок и доказательства корректности программы. Доказанная в 1936 году теорема об алгоритмической неразрешимости проблемы останова [2] показала невозможность существования полного и корректного алгоритма для автоматического доказательства корректности программ в рамках любой достаточно полной модели вычислителя. Поэтому исследования по развитию статических методов велись по двум направлениям: более или менее полное доказательство корректности программы с привлечением подсказок человека и полностью автоматический анализ с менее амбициозными целями, которые могут варьироваться от нахождения хоть каких-то типовых ошибок до доказательства выполнения определенных свойств программ при условии выполнения некоторых предположений.

1.1 Методы полуавтоматического анализа программ

Изначально преобладало направление разработки методов ручного доказательства, а также аксиоматической семантики и логических методов для работы с программами как с логическими объектами [3, 4, 5]. По мере роста размера и сложности программных систем ручное доказательство корректности их работы становилось слишком обременительным и порой давало повод усомниться в том, что получаемые в результате большие и сложные формальные доказательства заслуживают доверия [6].

Так появилась тенденция к автоматизации процесса верификации, при которой человек управляет процессом доказательства корректности, предоставляя инструменту необходимые утверждения (например, пред- и постусловия функций и инварианты циклов [3]). Это позволило расширить область применения методов автоматизации доказательства корректности программ, как в смысле их размеров, так и в смысле разнообразия проверяемых свойств.

Позднее область автоматизированной верификации программ испытала влияние трёх различных направлений исследований

- Развитие подходов к решению задачи выполнимости, сформулированной в рамках различных логических теорий, и непосредственно связанные с этим разработки решателей [7, 8, 9]. Эти инструменты предоставили собой практически применимое средство автоматического доказательства утверждений в бесконечных пространствах состояний.

- Разработка техник автоматической проверки моделей программ [10, 11, 12] на выполнимость в них свойств, сформулированных на языке темпоральных логик [13, 14], и соответствующих инструментов, основанных на обходе конечного пространства состояний модели.
- Формализация анализа программ в виде абстрактной интерпретации позволила использовать логические методы, оперирующие с бесконечными пространствами состояний при помощи алгоритмических конечных представлений.

В промежутке между 1980-ми и 1990-ми годами эти три направления развивались по большей части независимо с относительно редкими пересечениями. Однако к началу 1990-х произошло их сближение, в результате которого современные подходы к верификации программ основаны на объединении и одновременном использовании всех упомянутых техник. В частности, когда говорят "инструмент проверки моделей программ" – это, возможно, не отражает верного представления о современных инструментах автоматической статической верификации, поскольку большинство из них одновременно осуществляют различные виды анализа программ, такие как автоматическое доказательство теорем, проверка моделей и анализ потока данных [15].

В настоящее время направления исследований в области формальной верификации достаточно разнообразны. Для практического применения представляют интерес методы, которые достаточно хорошо автоматизируются и масштабируются для того, чтобы справляться с верификацией больших и сложных программных систем. Обзоры современных техник автоматизированной верификации программ можно найти, например, в [16], [17] и [18].

1.2 Методы автоматической верификации

Ключевыми характеристиками методов автоматической проверки программ являются

- полнота, то есть набор ограничений на используемые возможности языка программирования и виды обнаруживаемых ошибок;
- точность анализа, то есть количество выдаваемых ложных сообщений об ошибках;
- скорость их работы.

На основании этих характеристик существующие подходы можно условно разделить на два больших класса – легковесные и тяжеловесные.

1.2.1 Легковесные подходы

Легковесные подходы берут своё начало из области разработки компиляторов и программной оптимизации. С самого начала к ним предъявлялись высокие требования к скорости работы, потому что с их помощью должны были

достаточно быстро обрабатываться большие объёмы исходного кода. Для достижения скорости работы, сравнимой по порядку величины со временем компиляции анализируемого приложения, в данных подходах обычно применяется анализ потока данных [15] в сочетании с множеством различных эвристик, использование которых может привести к пропуску ошибок и снижению точности анализа. На сегодняшний день легковесные подходы развиты достаточно хорошо и существует большое количество реализующих их инструментов, как для индустриального, так и для академического использования [19, 20, 21, 22, 23].

Основной целью этих инструментов является нахождение наибольшего числа типовых ошибок, встречающихся при разработке программ на распространенных языках программирования, при приемлемом уровне ложных срабатываний. Примерами обнаруживаемых ошибок являются разыменование нулевого указателя, обращение за границами буфера, обращение к памяти из кучи после её освобождения и т. д. Для добавления поиска нового вида ошибок в легковесные инструменты, как правило, требуется разработка модуля на языке программирования, используемом для реализации инструмента.

1.2.2 Тяжеловесные подходы

Тяжеловесные подходы характеризуются тем, что они предъявляют существенно меньшие требования к скорости анализа. Это позволяет применять более полные и точные методы, которые при выполнении определенных ограничений на код программы позволяют доказать отсутствие в программе ошибок определенных видов. Большинство тяжеловесных подходов предназначено для работы с ошибками, выражаемыми в виде свойства достижимости определенной точки в программе. Примером таких ошибок является нарушение assertion'ов, расставленных в коде программы, или некорректное использование библиотечных функций.

Два наиболее развитых на сегодняшний день тяжеловесных метода анализа программ – это ограничиваемая проверка моделей [24] и уточнение абстракции по контрпримерам [25].

Ограничиваемая проверка моделей

Метод ограничиваемой проверки моделей (от англ. Bounded Model Checking, BMC) изначально рассматривался как одна из наиболее удачных техник верификации применительно к разработке полупроводниковых устройств. Схемы, используемые в таких устройствах, часто вначале моделируют программой на языке высокого уровня, например, Си, а затем, после тщательного тестирования или верификации, реализуют на языке описания аппаратуры (например, Verilog [26]).

Алгоритмы ограничиваемой проверки моделей основаны на разворачивании циклов программы на конечное фиксированное число шагов k и последующей проверки возможности нарушения проверяемого свойства на глубине, не большей, чем эти k шагов. Принудительное ограничение сверху числа k является основным ограничением на применимость этого метода для формального доказательства корректности.

Значительные достижения в области производительности современных SAT- и SMT-решателей (SAT от англ. Satisfiability – задача проверки выполнимости для формул пропозициональной логики, SMT от англ. Satisfiability Modulo Theories – задача проверки выполнимости для логических формул с учетом лежащих в их основе теорий, например, линейной арифметики) позволили анализировать поведение программ методом ограничиваемой проверки моделей с очень высокой точностью через побитовое кодирование значений переменных и операций над ними. Метод ограничиваемой проверки моделей позволяет проверять для языка программирования широкий набор конструкций, включая динамически выделяемые структуры в памяти, адресную арифметику, операции с массивами и побитовые операции. На сегодняшний день данный метод является наилучшим для поиска неглубоких – в смысле длины пути выполнения – ошибок в небольших программных системах.

Инструменты, реализующие метод ограничиваемой проверки моделей, такие как CBMC [27] или F-Soft [28] с успехом используются для поиска ошибок в системном программном обеспечении, например, в автомобильной промышленности [29]. Есть также примеры использования этого метода для доказательства отсутствия состояний гонки в низкоуровневом программном обеспечении [30].

Уточнение абстракции по контрпримерам

Метод уточнения абстракции по контрпримерам (от англ. Counter-Example Guided Abstraction Refinement, CEGAR) применяется для проверки достижимости определенной точки в программе. Основная идея метода заключается в следующем. Детальная модель анализируемой программы (то есть наиболее точная модель, семантически эквивалентная самой программе) слишком большая и сложная, чтобы за разумное время можно было проверить свойства всех возможных путей в ней, поэтому достижимость заданной точки проверяется на упрощенной модели. Но это может привести к некорректному результату: анализ упрощенной модели может показать, что точка недостижима, хотя в действительности она достижима, или наоборот.

Проблемы первого вида метод уточнения абстракции по контрпримерам решает за счет того, что выбирается такой способ построения упрощённой модели, при котором такой ситуации не может произойти «по построению». То есть всегда, когда анализ упрощённой модели говорит, что точка

недостижима, то она является недостижимой и в детальной модели программы.

Для борьбы с проблемами второго вида применяется «уточнение абстракции по контрпримерам». Если анализ упрощённой модели показывает, что заданная точка является достижимой, то это сопровождается предъявлением пути в исходной программе, по которому можно достичь целевую точку. Этот путь называется *контрпримером*. Поскольку задача проверки реализуемости одного конкретного пути в детальной модели программы уже не столь сложна, как анализ всех путей и, более того, текущий уровень развития инструментов позволяет её решать достаточно эффективно, то именно это и делается для всех контрпримеров, обнаруживаемых при помощи анализа упрощённой модели. Если проверка подтверждает реализуемость найденного пути, то на вопрос о достижимости заданной точки получен корректный ответ. В противном случае, контрпример является следствием абстрагирования от существенных деталей при построении упрощённой модели. Тогда необходимо провести анализ контрпримера, чтобы выявить те детали, из-за игнорирования которых появился некорректный контрпример, и уточнить упрощённую модель, чтобы учесть в ней эти детали. Работа с новой упрощённой моделью повторяется по тому же самому алгоритму.

В результате ряда итераций метод уточнения абстракции по контрпримерам либо найдет путь в заданную точку программы, либо докажет её недостижимость, либо упрется в ограничения по ресурсам. Таким образом, метод не гарантирует нахождения решения в общем случае. Тем не менее, на практике он показывает достаточно хорошие результаты.

Метод уточнения абстракции по контрпримерам лежит в основе инструмента статической верификации SLAM [58], который входит в состав системы Static Driver Verifier (SDV) [31] и в настоящее время активно используется разработчиками драйверов устройств для операционной системы Microsoft Windows. Другие инструменты тяжеловесного статического анализа, такие как BLAST [32], SATabs [33] и CPAchecker [34], также работающие по методу уточнения абстракции, имеют широкую известность среди исследователей в области автоматической верификации. В рамках проекта Linux Driver Verification (LDV) [35, 36] разрабатывается система статической верификации драйверов операционной системы Linux, в которой на практике активно используются инструменты BLAST и CPAchecker. Результаты использования этих инструментов говорят о возможности их успешного применения для покомпонентного анализа больших объёмов промышленного кода.

2 План статьи

Данная статья описывает основные принципы метода уточнения абстракции по контрпримерам (CEGAR) и затрагивает основные моменты реализации CEGAR в инструментах статической верификации BLAST и CPAchecker.

В следующем (третьем) разделе статьи вводятся основные понятия и определения, необходимые для изложения сущности рассматриваемого метода. Затем рассматривается сам метод в том его варианте, в котором он реализован в инструментах BLAST и CPAchecker. Поскольку инструменты, работающие на основе CEGAR, устроены достаточно сложно, рассмотрение метода проводится в несколько этапов. Сначала в разделах 4, 5, 6 и 7 рассматривается наиболее простой вариант предикатной абстракции и её уточнения по методу CEGAR. Рассмотрение проводится на примере верификации очень простой программы на языке C с указанным для неё свойством недостижимости. Затем в разделе 8 рассмотренный вариант CEGAR с предикатной абстракцией расширяется для применения его к более сложным программам, в этом же разделе рассказывается об используемых модификациях и оптимизациях метода, направленных на увеличение его точности, масштабируемости и производительности. В конце (в разделе 9) описываются внешние инструменты, которые используются при реализации метода CEGAR в BLAST и CPAchecker (такие как SAT-, SMT- и интерполирующие решатели).

3 Основные понятия и определения

3.1 Простейшие программы

В этом разделе мы будем рассматривать *простейшие программы* на C-подобном языке, каждая из которых удовлетворяет следующим ограничениям:

- Простейшая программа состоит из одной главной функции `main`, первый оператор которой считается единственной точкой входа в программу. Функция `main` имеет объявление

  ```
  void main(void);
  ```

  Таким образом, эта функция не принимает никаких параметров и не возвращает никакого значения.
- В простейшей программе могут быть объявлены только локальные переменные функции `main` целочисленного типа `int`. Объявления переменных допускаются только до первого оператора функции `main`. Все переменные должны быть явно инициализированы при объявлении недетерминированными значениями, то есть произвольными, не заданными наперёд значениями типа `int`. Для этого используется специальный инициализатор `nondet`.

- В качестве операторов допускаются только следующие:
  - присваивание переменной выражения без побочных эффектов;
  - ветвление с использованием оператора `if` (`if-else`) по условию, заданному выражением без побочных эффектов;
  - безусловный переход на метку (оператор `goto`).

  Данное ограничение, в частности, означает, что пустые операторы в простейших программах недопустимы.
- Выражения без побочных эффектов состоят только из целочисленных констант, переменных и операций над ними.
- В качестве операций в выражениях без побочных эффектов допускаются только линейные арифметические операции `+`, `-`, `*`(умножение на константу), операции сравнения `>`, `<`, `==`, `!=`, `>=`, `<=` и логическая операция `!`.

Будем считать, что в рассматриваемой простейшей программе все операторы, а также точка выхода из программы помечены своими уникальными метками.

Для рассматриваемых простейших программ будем проверять свойство недостижимости некоторого ошибочного оператора, помеченного заданной *ошибочной меткой*. При этом рассматривать свойство недостижимости будем вне зависимости от условий завершения программы. *Завершением программы* называется достижение в процессе выполнения программы выхода из этой программы при отсутствии ошибок фазы выполнения. В случае простейших программ ошибки фазы выполнения всегда исключены, потому что все допустимые операторы и операции в выражениях таких программ определены всегда, в частности, отсутствует операция деления. Поэтому простейшая программа не может завершиться аварийно. Она может либо достигнуть выхода, либо зациклиться. Если простейшая программа зацикливается, никогда не достигая в ходе выполнения ошибочного оператора, свойство недостижимости ошибочной метки для такого выполнения простейшей программы считается выполненным.

Пример простейшей программы с ошибочной меткой `ERR` представлен на рис. 1а. Приведенная программа вычисляет модуль (абсолютное значение) разности двух произвольно заданных чисел – x и y и сохраняет его в переменной z. Меткой `ERR` помечен ошибочный оператор, который не должен достигаться в ходе корректного выполнения данной программы.

```
void main()
{
        int x = nondet;
        int y = nondet;
        int z = nondet;
L1:     if (x > y) {
L2:             z = x - y;
        } else {
L3:             z = y - x;
        }
L4:     if (z < 0)
ERR:    goto ERR;
L5:}
```

Рис. 1а. Пример простейшей программы.

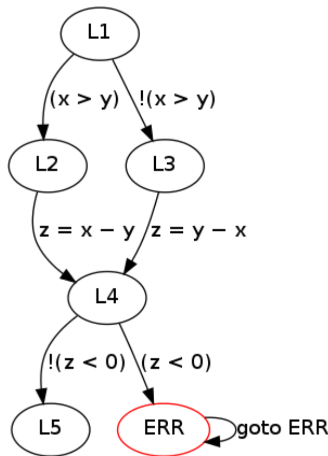


Рис. 1б. Граф потока управления для данной программы

3.2 Граф потока управления

Введем для простейшей программы понятие *графа потока управления* (*ГПУ*, от англ. Control flow graph, CFG). *Граф потока управления* – ориентированный граф, вершинам которого соответствуют уникальные метки, которыми помечен каждый оператор программы, а также точка выхода из неё, а направленным помеченным дугам соответствуют операторы программы. При этом каждому оператору присваивания или безусловного перехода однозначно соответствует единственная дуга в ГПУ, а оператору ветвления – две дуги: одна для указанного в операторе условия перехода и одна для отрицания этого условия. Вершина ГПУ, соответствующая метке первого оператора программы, называется его *начальной вершиной*. Для программы из

примера 1 на рис. 1а соответствующий граф потока управления представлен на рисунке 1б.

3.3 Состояния программы

Назовём *состоянием* простейшей программы пару (L, v), где L – уникальная метка следующего за ней (то есть помеченного этой меткой) выполняемого оператора программы или точки выхода, v – означивание всех переменных (то есть отображение имен всех переменных программы в их конкретные значения) непосредственно перед выполнением оператора, помеченного меткой L. Отметим, что состояние программы может как быть достижимо в ходе какого-либо её реального выполнения, так и не быть таковым.

Для программы в примере 1 состояниями считаются, например, такие пары:

$$(L1, \{x \mapsto 0, y \mapsto 0, z \mapsto 0\}),$$

$$(L2, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\}),$$

$$(L5, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\}),$$

$$(L5, \{x \mapsto 1, y \mapsto 0, z \mapsto -1\}).$$

Состояния программы (L, v), в которых L — имя метки первого оператора программы, назовём *начальными состояниями* этой программы. Примеры начальных состояний простейшей программы из примера 1:

$$(L1, \{x \mapsto 0, y \mapsto 0, z \mapsto 0\}),$$
$$(L1, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\}),$$
$$(L1, \{x \mapsto 1, y \mapsto 2, z \mapsto -1\}).$$

Если ГПУ простейшей программы не содержит кратных дуг (формальное описание процесса построения ГПУ без кратных дуг приведено в секции 4.1), то любой упорядоченной паре состояний программы можно поставить в соответствие не более одной дуги ГПУ.

Например, в нашей программе упорядоченным парам состояний $((L1, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}), (L2, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}))$ и $((L1, \{x \mapsto 0, y \mapsto 0, z \mapsto 0\}), (L2, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\}))$ в ГПУ соответствует одна и та же дуга $L1 \to L2$. А паре состояний $((L1, \{x \mapsto 0, y \mapsto 0, z \mapsto 0\}), (L7, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\}))$ не соответствует никакой дуги ГПУ.

Для упорядоченной пары состояний, которой соответствует некоторая дуга ГПУ, может быть выполнено отношение *непосредственной достижимости*. Смысл этого отношения в том, что второе состояние получается из первого в результате выполнения оператора, которым помечена соответствующая дуга ГПУ. В случае оператора ветвления по некоторому условию c договоримся считать, что одна из соответствующих ему дуг ГПУ помечена оператором перехода по условию c, а другая — оператором перехода по условию $\neg c$.

Будем обозначать непосредственную достижимость состояния (L_2, v_2) из состояния (L_1, v_1) по дуге ГПУ, помеченной оператором op, с помощью символа $\rightarrow$: $(L_1, v_1) \overset{op}{\rightarrow} (L_2, v_2)$. Формально отношение непосредственной достижимости для произвольной упорядоченной пары состояний $((L_1, v_1), (L_2, v_2))$ и произвольного оператора op можно определить следующим образом:

$$(L_1, v_1) \overset{op}{\rightarrow} (L_2, v_2) \Leftrightarrow \text{в ГПУ есть дуга } L_1 \overset{op}{\rightarrow} L_2 \text{ и}$$

$$\begin{cases} v_2 = v_1, \quad \text{если } op - \text{оператор } goto \\ v_2 = v_1\{x_k \mapsto e(v_1(x_1), \ldots, v_1(x_n))\}, \\ \text{если } op - \text{оператор присваивания } x_k = e(x_1, \ldots, x_n) \\ v_2 = v_1 \text{и } c\big(v_1(x_1), \ldots, v_1(x_n)\big), \\ \text{если } op - \text{оператор перехода по условию } c(x_1, \ldots, x_n) \end{cases}$$

Здесь $x_1, \ldots, x_n$ — переменные простейшей программы; $v_1\{x_k \mapsto e(v_1(x_1), \ldots, v_1(x_n))\}$ — означивание, которое получается из v_1 заменой значения переменной x_k на значение выражения $e(x_1, \ldots, x_n)$ (из правой части оператора присваивания), вычисленное при значениях переменных, заданных означиванием v_1, $c(x_1, \ldots, x_n)$ — условие, которым помечена одна из двух дуг ГПУ, соответствующих оператору ветвления.

Для нашей программы (на рис. 1) примеры состояний, связанных отношением непосредственной достижимости:

$$(L1, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}) \xrightarrow{(x>y)} (L2, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}),$$

$$(L2, \{x \mapsto 1, y \mapsto -1, z \mapsto -1\}) \xrightarrow{z = x - y} (L4, \{x \mapsto 1, y \mapsto -1, z \mapsto 2\}),$$

$$(ERR, \{x \mapsto 1, y \mapsto 0, z \mapsto -1\}) \xrightarrow{goto\ ERR} (ERR, \{x \mapsto 1, y \mapsto 0, z \mapsto -1\}).$$

3.4 Граф достижимости

Рассмотрим теперь *граф достижимости* — ориентированный граф, задающий отношение непосредственной достижимости на множестве всевозможных состояний простейшей программы. Вершинами этого графа являются состояния, а дуга от одной вершины к другой есть тогда и только тогда, когда соответствующая упорядоченная пара состояний связана отношением непосредственной достижимости. Дуги графа достижимости помечены соответствующими операторами.

Если в графе достижимости есть путь P из вершины (L_1, v_1) в вершину (L_n, v_n):

$(L_1, v_1) \xrightarrow{op_1} (L_2, v_2) \xrightarrow{op_2} \dots \xrightarrow{op_{n-1}} (L_n\ v_n)$, то говорят, что состояние (L_n, v_n) достижимо из состояния (L_1, v_1) по пути P. Будем также говорить о достижимости состояния (L_n, v_n), подразумевая его достижимость из какого-либо начального состояния.

В примере 1 достижимо, например, состояние $(L5, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\})$, из начального состояния $(L1, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\})$ по пути

$$(L1, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}) \xrightarrow{(x>y)} (L2, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}) \xrightarrow{z\,=\,x\,-\,y}$$
$$(L4, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\}) \xrightarrow{!(z<0)} (L5, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\})$$

3.5 Задача инструмента верификации

Недостижимость, то есть отсутствие достижимости, всех состояний с ошибочной меткой (состояний (ERR, v), где ERR — ошибочная метка) соответствует для простейшей программы недостижимости самой ошибочной метки (в ходе любого выполнения программы). Состояния с ошибочной меткой будем также называть *ошибочными состояниями.* Формально можно сказать, что задача инструмента верификации – проверить для заданной простейшей программы достижимость какого-либо ошибочного состояния, то есть существование в графе достижимости пути из какого-либо начального состояния в какое-либо состояние с ошибочной меткой. В случае достижимости ошибочного состояния найденный инструментом путь из начального состояния в ошибочное будем называть *примером ошибочного пути*. Итак, задача инструмента верификации – либо доказать для данной на вход простейшей программы недостижимость ошибочных состояний, либо привести соответствующий пример ошибочного пути. В программе из примера 1 ошибочные состояния с меткой ERR (и соответственно сама ошибочная метка ERR) являются недостижимыми.

Оценим некоторым образом ту работу, которую необходимо было бы проделать инструменту статической верификации для проверки условия недостижимости, если бы эта проверка осуществлялась, к примеру, непосредственным поиском в графе достижимости. По крайней мере, инструменту потребовалось бы проверять достижимость из каждого возможного начального состояния программы. Если в нашем простейшем примере считать все переменные программы 32-битными, то инструменту верификации пришлось бы перебрать

$$(2^{32})^3 = 2^{3 \cdot 32} = 2^{96} > 7{,}9 \cdot 10^{28} > (2{,}5 \cdot 10^{12}) \cdot 365 \cdot 24 \cdot 60^2 \cdot 10^9$$

возможных начальных состояний. Такое количество состояний уже неприемлемо велико для современных вычислительных машин. Если бы в секунду удавалось проверять достижимость из миллиарда начальных состояний, то на верификацию простейшей программы из примера 1 ушло бы больше 2 триллионов лет.

Проблема резкого («взрывного») увеличения числа состояний программы с ростом числа её переменных и операторов получила название проблемы *комбинаторного взрыва*. Для её решения применяются различные подходы. Рассмотрим один из способов борьбы с проблемой комбинаторного взрыва, который называется предикатной абстракцией.

3.6 Предикатная абстракция и метод CEGAR

При использовании абстракции вместо состояний исходной программы рассматриваются *абстрактные состояния*, которые представляют собой множества состояний программы. Эти множества могут пересекаться или быть вложены одно в другое. На основе некоторых заданных вначале абстрактных состояний строится абстрактное дерево достижимости (АДД, определение АДД вводится в разделе 3.9), состоящее из абстрактных состояний и переходов между ними по операторам программы. Строится оно таким образом, что если существует переход между какими-либо двумя состояниями программы, то существует и переход между соответствующими им абстрактными состояниями, которые включают эти состояния. По построению получается, что АДД обязательно включает все достижимые в исходной программе состояния (как элементы соответствующих абстрактных состояний), а также, возможно, и некоторые другие. Поэтому пути, выполнение по которым возможно в исходной программе, обязательно оказываются представленными в АДД в виде последовательности переходов между абстрактными состояниями. Обратное, вообще говоря, неверно. Из-за использования абстрактных состояний вместо состояний исходной программы в АДД могут быть представлены и фиктивные пути, выполнение по которым на самом деле невозможно.

Если в АДД нет *ошибочных абстрактных состояний* (то есть абстрактных состояний, включающих ошибочные состояния программы), то ошибочные состояния недостижимы и в исходной программе. Если же в АДД находится ошибочное абстрактное состояние, то это не обязательно говорит о достижимости ошибочного состояния в исходной программе, так как выбранные абстрактные состояния могут быть слишком неточными, для того чтобы показать недостижимость ошибочного состояния.

Для описания абстрактных состояний будем использовать частный случай абстракций — *предикатные абстракции* — то есть абстракции, в которых абстрактные состояния представляются логическими выражениями (предикатами) над переменными программы. В таких абстракциях состояния программы объединяются во множества (абстрактные состояния) по признаку истинности в них некоторых логических выражений (например, состояния, где $x > 0$, состояния, где $x = y$ или состояния, где $x > 0 \land y = 0$).

Для построения переходов между вершинами АДД при использовании предикатных абстракций операторы программы кодируются в виде логических формул. Для проверки истинности того или иного предиката после

выполнения оператора также составляется логическая формула, задающая множество возможных состояний программы после выполнения оператора. Затем проверяется логическое следование (импликация) из этой формулы каждого из заданных логических выражений. Это соответствует проверке вложенности множества возможных состояний программы после выполнения оператора во множество состояний, в которых истинно заданное логическое выражение. Пересечение таких множеств, соответствующее конъюнкции логических выражений, выбирается в качестве следующего абстрактного состояния после выполнения оператора.

Если в построенном АДД существует путь, ведущий в одно из ошибочных абстрактных состояний, то этот путь называют контрпримером. Его наличие может означать как реальную ошибку, так и неточность выбранной предикатной абстракции.

Для установления истинности ошибки по контрпримеру составляется формула сильнейшего постусловия пути (определяется в разделе 6.3), описывающая существование аналогичного пути в графе достижимости по состояниям программы. Если формула выполнима, то программа содержит ошибку. Иначе данная формула используется для уточнения предикатной абстракции, то есть для пополнения набора выбранных предикатов. После этого АДД перестраивается заново. АДД, которое строится после добавления в набор новых предикатов, не должно более включать тот же самый ложный ошибочный путь.

Цикл повторных построений АДД повторяется, пока мы не докажем корректность программы (отсутствие ошибочных абстрактных состояний в АДД) или не найдем реальную ошибку.

Это общая схема работы метода CEGAR – уточнения абстракции (в данном случае – абстрактного дерева достижимости) по контрпримерам. Начнем рассмотрение его реализации в инструментах BLAST и CPAchecker с более подробного разбора понятия абстрактного состояния и предикатной абстракции (точнее, ее частного случая – декартовой предикатной абстракции) на примере.

3.7 Абстрактные состояния и декартова предикатная абстракция

3.7.1 Предикатная абстракция на примере

Предикатная абстракция основана на введении на множестве всевозможных означиваний всех переменных программы некоторой системы подмножеств, упрощенно представляющих основные и наиболее существенные свойства входящих в эти подмножества означиваний. Для примера возьмём простейшую программу с одной переменной (рис. 2).

```
void main() {
      int x = nondet;
L1:   if (x >= 5)
L2:         if (x <= 10) {
L3:               x = x + 1;
L4:               if (!(x >= 5))
ERROR:                  goto ERROR;
            }
L5:}
```

Рис. 2. Простейшая программа с одной переменной x.

На рис. 3 схематически изображено множество всех возможных значений (фактически, означиваний) единственной переменной x этой программы.

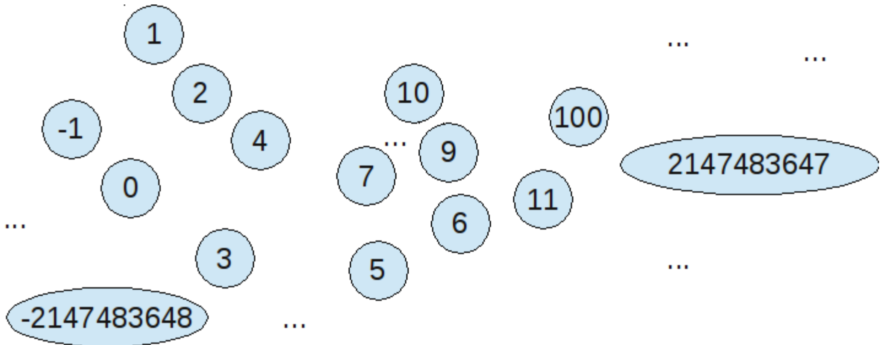


Рис. 3. Множество всех возможных значений переменной x.

Будем считать далее, что переменная x может принимать любые целочисленные значения, в том числе вне диапазона типа `int`. Таким образом, схематически изображенные на рисунке значения -2147483648 и 2147483647 являются просто одними из возможных значений этой переменной, а не наименьшим и наибольшим возможным её значением соответственно. В тексте программы встречаются условия $x \geq 5$ и $x \leq 10$, поэтому выберем их для примера. Рассмотрим систему подмножеств,

представляющую свойства $x \geq 5$ и $x \leq 10$ для входящих в эти подмножества означиваний.

На рис. 4 схематически изображена такая система подмножеств.

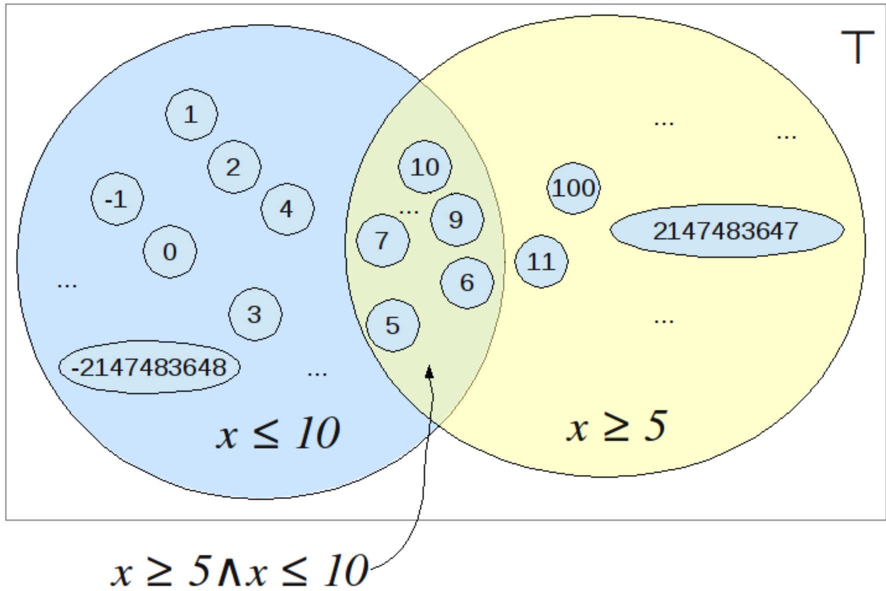


Рис. 4. Система подмножеств множества значений переменной x.

На рисунке выделены два подмножества таких означиваний переменной x, в которых выполнены условия $x \geq 5$ и $x \leq 10$ соответственно. Эти условия – *предикаты* над переменными (в данном случае – единственной переменной) простейшей программы. Показано также пересечение этих подмножеств, которому соответствует предикат $x \geq 5 \land x \leq 10$, и множество всех означиваний $\top$ («топ»), которому соответствует тождественная истина. Такие системы подмножеств называют *предикатными абстракциями* над множеством означиваний переменных программы.

В данной статье будем рассматривать только предикатные абстракции, в которые входят:

- подмножества означиваний, элементы которых удовлетворяют некоторому заранее заданному предикату (например, $x \geq 5$),
- пересечения этих подмножеств (соответствующие конъюнкциям предикатов, например, $x \geq 5 \land x \leq 10$),
- подмножество всех означиваний $\top$ (соответствующее тождественной истине) и

- пустое подмножество означиваний $\bot$ («ботом», «дно», соответствующее тождественной лжи).

В эти абстракции *не* входят объединения, разности, дополнения и результаты каких-либо других теоретико-множественных операций над подмножествами означиваний и соответственно не рассматриваются дизъюнкции и отрицания заданных предикатов. Такие предикатные абстракции называют *декартовыми*. Подмножества означиваний всех переменных программы, пересечения этих подмножеств, а также специальные подмножества $\top$ и $\bot$ называют *абстрактными состояниями данных*, а соответствующие им предикаты и их конъюнкции (в том числе тождественную истину и особый случай – тождественную ложь) – *регионами абстрактных состояний*. В абстрактное состояние программы также входит состояние её потока управления (имя уникальной метки следующего выполняемого оператора), поэтому *абстрактным состоянием программы* (или просто *абстрактным состоянием*) будем называть пару (L, r), где L – имя уникальной метки, задающее состояние потока управления программы, r – регион абстрактного состояния (конъюнкция предикатов или тождественная ложь, соответствующая абстрактному состоянию данных).

Рассмотрим теперь один из операторов нашей простейшей программы с одной переменной. Возьмём оператор $x = x + 1$, помеченный меткой $L3$. В данном разделе будем изображать на одном рисунке означивания переменной x в двух последовательных состояниях программы, например, на метке $L3$ (непосредственно перед выполнением оператора $x = x + 1$) и на метке $L4$ (перед проверкой условия $!(x \geq 5)$). Переходы между значениями переменной x при выполнении перехода из одного состояния в другое (например, из состояния с меткой $L3$ в состояние с меткой $L4$) будем изображать стрелками. На рис. 5 показаны переходы между означиваниями переменной x при выполнении выбранного нами оператора $x = x + 1$.

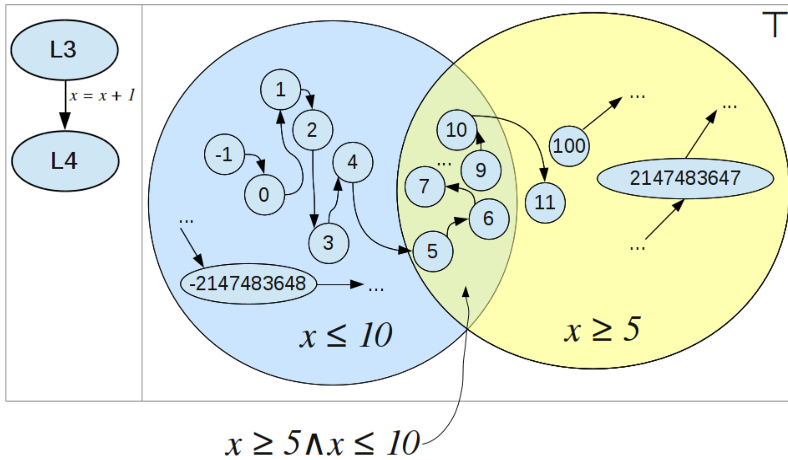


Рис. 5. Переходы между означиваниями переменной x при выполнении оператора $x = x + 1$.

На рисунке показано, что множество значений переменной рассматривается как неограниченное. Предположим, что при некотором состоянии потока управления (в нашем случае – перед выполнением оператора $x = x + 1$, помеченного меткой $L3$) все возможные в этом месте программы означивания переменной x входят в абстрактное состояние с регионом $x \geq 5 \wedge x \leq 10$. Рассмотрим теперь переходы по оператору $x = x + 1$ только из означиваний, входящих в это абстрактное состояние. Эти переходы показаны на рис. 6.

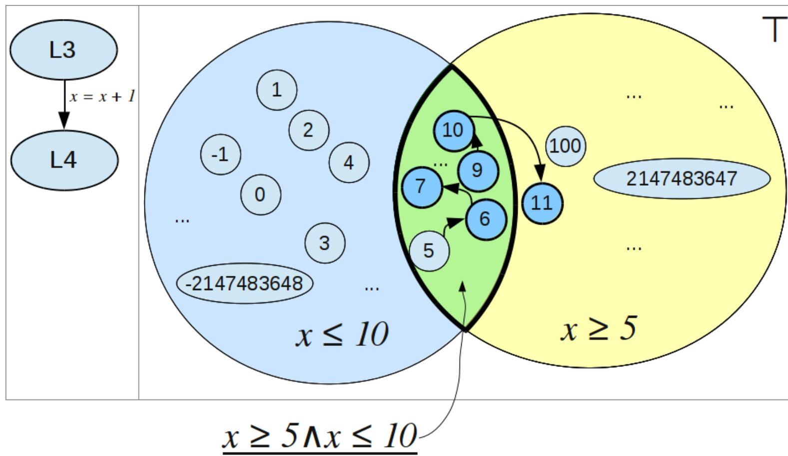


Рис. 6. Переходы по оператору $x = x + 1$ из абстрактного состояния с регионом $x \geq 5 \wedge x \leq 10$.

На рисунке выделены возможные значения переменной x после выполнения оператора $x = x + 1$ из абстрактного состояния с регионом $x \geq 5 \wedge x \leq 10$. Это подмножество выделенных возможных значений переменной x можно покрыть подмножеством $x \geq 5$, так как все выделенные значения принадлежат этому подмножеству. Таким образом, можно сказать, что после выполнения оператора $x = x + 1$ из абстрактного состояния $(L3,\ x \geq 5 \wedge x \leq 10)$, все возможные означивания переменной x можно ограничить (покрыть) абстрактным состоянием с регионом $x \geq 5$. В этом случае говорят, что переход из абстрактного состояния $(L3,\ x \geq 5 \wedge x \leq 10)$ по оператору $x = x + 1$ ведет в абстрактное состояние $(L4,\ x \geq 5)$. Вместо абстрактного состояния $x \geq 5$, вообще говоря, можно рассматривать абстрактное состояние $\top$, которое включает его. Но при использовании описанного далее (в следующем подразделе) метода построения переходов между абстрактными состояниями в данном случае будет получаться именно $x \geq 5$. Построенный переход изображен на рис. 7.

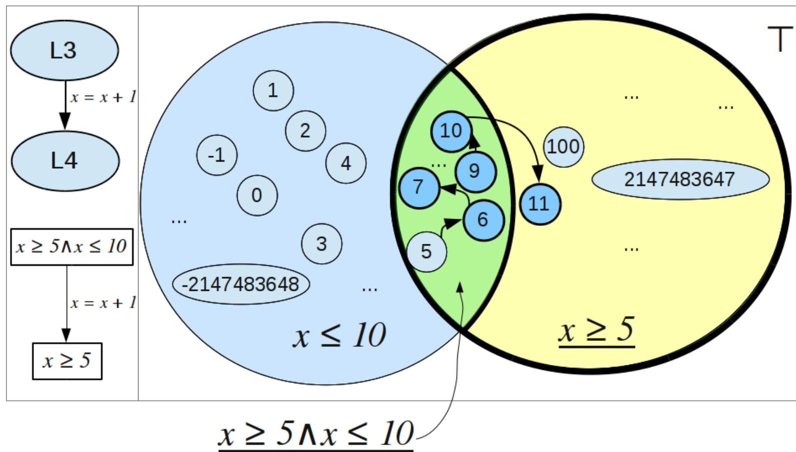


Рис. 7. Переход из абстрактного состояния с регионом $x \geq 5 \wedge x \leq 10$ по оператору $x = x + 1$ в абстрактное состояние с регионом $x \geq 5$.

3.7.2 Построение переходов между абстрактными состояниями с помощью решателей на примере

Для построения переходов между абстрактными состояниями на практике заметим, что отношение вложенности подмножеств означиваний соответствует отношению логического следования (импликации) между соответствующими логическими предикатами (в широком смысле). Множество означиваний A вложено в подмножество означиваний B тогда и

только тогда, когда для соответствующих предикатов a и b выполнено отношение $a \to b$ (рис. 8).

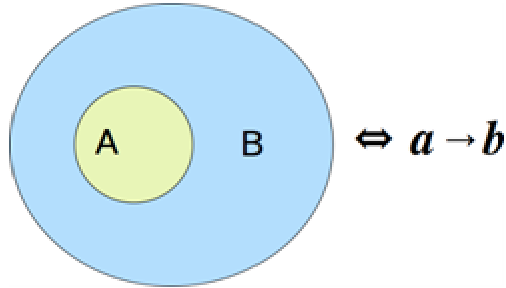


Рис. 8. Иллюстрация к соответствию отношений $A \subseteq B$ и $a \to b$.

Это означает, что абстрактное состояние данных A вложено в абстрактное состояние данных B тогда и только тогда, когда для соответствующих им регионов a и b выполнено логическое отношение $a \to b$.

Воспользуемся этим свойством для проверки вложенности подмножества всевозможных означиваний переменной x из предыдущего примера (после выполнения оператора $x = x + 1$ из абстрактного состояния $(L3,\ x \geq 5 \wedge x \leq 10)$) в различные абстрактные состояния данных ($x \geq 5$ и $x \leq 10$). В предположении неограниченности множества значений переменной x подмножеству всевозможных её означиваний можно поставить в соответствие предикат (в широком смысле) $x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1$. Это предикат от переменных x_1 и x_2, соответствующих означиваниям переменной x до и после выполнения оператора $x = x + 1$ соответственно. Вложенность подмножества означиваний x после выполнения оператора $x = x + 1$ в подмножество $x \geq 5$ можно проверить, проверив импликацию $(x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1) \to x_2 \geq 5$. Эта импликация является тождеством тогда и только тогда, когда подмножество всех возможных означиваний переменной x после выполнения перехода вложено в абстрактное состояние данных $x \geq 5$. Отношение является тождеством тогда и только тогда, когда невыполнимо его отрицание (в силу полноты и непротиворечивости логики первого порядка). Поэтому для проверки вложенности подмножества возможных означиваний x в абстрактное состояние данных $x \geq 5$ можно проверить выполнимость формулы

$$\neg((x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 +\ 1) \to x_2 \geq 5) =$$

$$\neg(\neg(x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1) \vee x_2 \geq 5) =$$

$$(x_1 \geq 5 \wedge x1 \leq 10 \wedge x_2 = x_1 + 1) \wedge \neg(x_2 \geq 5)$$

Аналогично можно проверить вложенность в абстрактное состояние данных $x \leq 10$. Для проверки выполнимости таких логических формул – нулевого порядка, то есть без кванторов, с равенством и линейной целочисленной арифметикой – можно использовать специальные инструменты, называемые SMT-решателями.

Когда подмножество одновременно вложено в два или более абстрактных состояния данных, оно является вложенным в пересечение этих абстрактных состояний данных. Регион, соответствующий пересечению абстрактных состояний данных, является конъюнкцией регионов соответствующих абстрактных состояний.

При этом следует учитывать, что абстрактное состояние данных $\perp$ является вложенным в любое абстрактное состояние данных. В зависимости от заданных предикатов, абстрактное состояние данных $\perp$ может совпадать с пересечением двух или более (в том числе, всех) абстрактных состояний данных (например, $x \geq 5 \wedge x < 5 \equiv \perp$), а может быть и строго вложено в пересечение всех абстрактных состояний данных (как в данном примере). В то же время абстрактное состояние данных $\perp$ имеет большое значение, поскольку соответствует пустому множеству означиваний и, как следствие, пустому множеству состояний программы и неосуществимости перехода по некоторому оператору из какого-либо абстрактного состояния. В дальнейшем неосуществимость такого перехода будет использована для доказательства недостижимости ошибочной метки. Поэтому, чтобы отличать состояние $\perp$ от пересечения всех абстрактных состояний данных (в общем случае они не совпадают), вначале проверяют выполнимость предиката (в широком смысле), соответствующего множеству всевозможных означиваний переменных после выполнения перехода по оператору (в частности, по оператору проверки условия).

В нашем примере предикат $x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1$ как логическая формула является выполнимым. Новое абстрактное состояние данных после перехода не равно $\perp$. В качестве нового абстрактного состояния данных берем конъюнкцию всех абстрактных состояний данных, в которые вложено подмножество всевозможных означиваний переменной x после перехода. Формула $(x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1) \wedge \neg(x_2 \geq 5)$ невыполнима, а формула $(x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1) \wedge \neg(x_2 \leq 10)$ выполнима. В качестве результата берем конъюнкцию из одного абстрактного состояния данных $x \geq 5$.

Так как в примере оператор $x = x + 1$ помечен меткой $L3$, а следующий за ним оператор – меткой $L4$, то графически построенный переход можно изобразить так, как показано на рис. 9а.

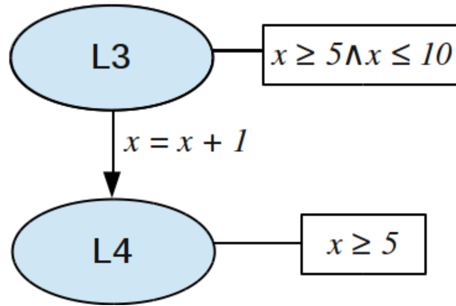


Рис. 9а. Переход из абстрактного состояния $(L3, x \leq 5 \land x \geq 10)$ *по оператору* $x = x + 1$ *в абстрактное состояние* $(L4, x \geq 5)$.

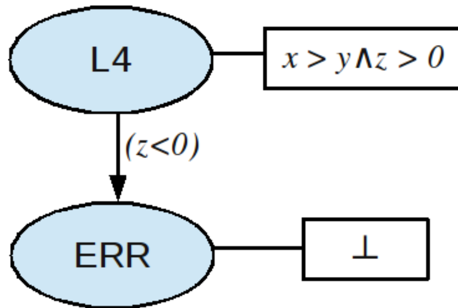


Рис. 9б. Переход из абстрактного состояния $(L4, x > y \land z > 0)$ *в абстрактное состояние* $(ERR, \bot)$.

Приведем еще один пример. Для программы из примера 1 построим переход из абстрактного состояния $(L4, x > y \land z > 0)$ по оператору ветвления с условием $(z < 0)$. Предикат на возможные означивания переменных после перехода: $x > y \land z > 0 \land z < 0$ – не является выполнимым как логическая формула. Следовательно, новое абстрактное состояние равно $(ERR, \bot)$. Построенный переход показан на рисунке 9б.

3.7.3 Общий случай декартовой предикатной абстракции

Итак, инструменты статической верификации, использующие абстракцию, рассматривают вместо каждого отдельного состояния сразу подмножества состояний исходной программы. Каждое такое подмножество может включать

в себя очень большое число состояний, например, 2^{96} или даже намного больше. В подмножество объединяют некоторые состояния, имеющие одинаковые метки. Мы рассматриваем случай, когда в подмножество объединяются состояния, в которых переменные программы удовлетворяют некоторому набору логических предикатов нулевого порядка (без кванторов).

Пусть задан некоторый предикат $p(x_1, \ldots, x_n)$ над переменными программы $x_1, \ldots, x_n$ (предикат не обязательно существенно зависит от всех переменных программы). В каждом состоянии программы (L, v) означивание v задаёт значения всех переменных $x_1, \ldots, x_n$. Подставим эти значения вместо соответствующих переменных в предикат $p(x_1, \ldots, x_n)$. Получим $p\big(v(x_1), \ldots, v(x_n)\big)$ —*значение предиката p в состоянии* (L, v). Это значение всегда можно вычислить, оно будет являться логической (булевой) константой и будет равно либо истине, которую мы обозначим символом $\top$, либо лжи, которую мы обозначим символом $\bot$. В случае $p\big(v(x_1), \ldots, v(x_n)\big) = \top$ будем писать $v \vdash p(x_1, \ldots, x_n)$ или просто $v \vdash p$.

Пусть теперь дана некоторая фиксированная уникальная метка L какого-либо оператора исходной простейшей программы. Для данного предиката p можно рассмотреть подмножество всех возможных состояний программы с заданной меткой L, в которых значение предиката p равно тождественной истине. Обозначим это подмножество через $[\![p]\!]_L$: $[\![p]\!]_L \stackrel{\text{def}}{=} \{(L, v) \colon v \vdash p\}, L$ фиксировано.

Одним из возможных способов задания и представления набора множеств состояний исходной программы (для последующей работы с полученным набором представлений этих множеств) является *декартова предикатная абстракция*. Пусть задан некоторый конечный набор предикатов $\{p_1, \ldots, p_n\}$. Рассмотрим некоторое подмножество предикатов из этого набора: $\{p_{i_1}, \ldots, p_{i_m}\} \subseteq \{p_1, \ldots, p_n\}$. В декартовой предикатной абстракции подмножеству предикатов $\{p_{i_1}, \ldots, p_{i_m}\} \subseteq \{p_1, \ldots, p_n\}$ ставят в соответствие конъюнкцию этих предикатов

$$p_{i_1} \wedge \ldots \wedge p_{i_m} = \bigwedge_{k=1}^{m} p_{i_k},$$

которую, в свою очередь, рассматривают как новый предикат, и ставят в соответствие этому предикату подмножество состояний исходной программы с фиксированной меткой L:

$$\left[\!\!\left[\bigwedge_{k=1}^{m} p_{i_k}\right]\!\!\right]_L = [\![p_{i_1} \wedge \ldots \wedge p_{i_m}]\!]_L$$

$$= \{(L, v): v \vdash p_{i_1} \wedge \ldots \wedge p_{i_m}\}, L \text{ фиксировано.}$$

Формально в декартовой предикатной абстракции *абстрактным состоянием* называют пару (L, r), где L — уникальная метка, а r — либо конъюнкция подмножества предикатов из заранее заданного набора, $r \equiv p_{i_1} \wedge \ldots \wedge p_{i_m}, \{p_{i_1}, \ldots, p_{i_m}\} \subseteq \{p_1, \ldots, p_n\}, 0 \leq m \leq n$, либо тождественная ложь, $r \equiv \perp$. В частности, если $m = 0$, то $r \equiv \top$ и $v \vdash r$ для любого v; это означает что абстрактное состояние $(L, \top)$ задаёт все возможные состояния программы с заданной меткой L. Если же $r \equiv \perp$, то $v \vdash r$ не верно ни для какого означивания v, то есть абстрактное состояние $(L, \perp)$ задаёт пустое множество состояний программы, которому формально приписана некоторая уникальная метка L. Обозначим множество состояний программы, заданных абстрактным состоянием (L, r) через $[\![(L, r)]\!]$. Тогда формально можно записать:

$$[\![(L, r)]\!] \stackrel{\text{def}}{=} [\![r]\!]_L = \{(L, v): v \vdash r\}, L \text{ фиксировано.}$$

В абстрактном состоянии (L, r) r является предикатом, который называют *регионом абстрактного состояния*. Для состояния программы (L, v) такого, что $v \vdash r$или, что то же самое, $(L, v) \in [\![(L, r)]\!]$ будем говорить, что состояние программы (L, v) входит в абстрактное состояние (L, r).

В качестве пояснения приведём примеры возможных регионов для заданного набора из двух предикатов:

$$p_1 \equiv x > 0$$

$$p_2 \equiv x = 0$$

Возможные регионы:

$$r_1 \equiv \top$$

$$r_2 \equiv p_1 \equiv (x > 0)$$

$$r_3 \equiv p_2 \equiv (x = 0)$$

$$r_4 \equiv p_1 \wedge p_2 \equiv (x > 0) \wedge (x = 0) \equiv \perp$$

$$r_5 \equiv \perp \equiv r_4$$

Здесь, к примеру, абстрактное состояние $(L, r_2) = (L, p_1) = (L, x > 0)$ (x — целое) задаёт множество конкретных состояний с меткой L, в которых

переменная x принимает строго положительные значения ($x = 1, 2, ...$), а остальные переменные принимают произвольные значения.

3.8 Сильнейшие постусловия и вычисление регионов

В инструментах статической верификации на основе предикатной абстракции, в том числе в BLAST и CPAchecker, логические формулы являются основным средством представления абстрактных состояний и работы с ними. Теперь для дальнейшего изложения нам требуется определить некоторую аналогию непосредственной достижимости для абстрактных состояний.

Пусть дано некоторое абстрактное состояние (L, r) и дуга в ГПУ с меткой op. Будем формально говорить для заданной метки L и предиката от переменных программы $p(x_1, \dots, x_n)$, что все состояния программы из множества $[\![p]\!]_L = \{(L, v): v \vdash p\}$ заданы меткой L и предикатом p. *Сильнейшим постусловием оператора* op при данном *предусловии* p называется предикат от переменных программы, задающий вместе с некоторой меткой L' те и только те состояния программы, которые непосредственно достижимы из состояний, заданных предусловием (предикатом) p и некоторой меткой L, по дуге, помеченной оператором op. Если задать дугу $L \overset{op}{\rightarrow} L'$ в ГПУ, и некоторое предусловие $p(x_1, \dots, x_n)$, то можно сказать, что сильнейшее постусловие p' оператора op при заданном предусловии p задаёт вместе с меткой L' множество $[\![p']\!]_{L'} = \{(L', v'): \exists (L, v),\text{ такое, что } v \vdash p \text{ и } (L, v) \overset{op}{\rightarrow} (L', v')\}$. Предикат сильнейшего постусловия можно задавать в виде, зависящем только от предусловия p и оператора op. Поэтому сильнейшее постусловие оператора op при данном предусловии p мы будем обозначать через $SP(p, op)$ (то есть $p' = SP(p, op)$).

Регион является предикатом от переменных программы и может быть использован в качестве предусловия. Рассмотрим сильнейшее постусловие $SP(r, op)$ для региона r абстрактного состояния (L, r) и оператора op на дуге $L \overset{op}{\rightarrow} L'$ ГПУ. Использование декартовой предикатной абстракции даёт возможность достаточно просто подобрать регион, являющийся следствием сильнейшего постусловия $SP(r, op)$ и таким образом задать новое абстрактное состояние (L', r'), в которое входят все конкретные состояния, достижимые из данного абстрактного состояния (L, r) по дуге с пометкой op. В это абстрактное состояние могут также войти и другие конкретные состояния, в которые не входит дуга с пометкой op, выходящая из (L, r). Поэтому (L', r') называют *верхним приближением (*от англ. *over-approximation)* множества конкретных состояний, достижимых из (L, r) по дуге с меткой op. По сути, верхнее приближение для абстрактных состояний означает то же, что и надмножество.

Введём обозначение выполнимости логической формулы: будем писать $\varphi - SAT$, если формула φ выполнима[1] и $\varphi - UNSAT$, если φ — невыполнима.

Для того чтобы подобрать регион, являющийся следствием сильнейшего постусловия $SP(r, op)$, воспользуемся свойством для произвольных предикатов $P, p_1, \ldots, p_m$: $P \to p_1 \wedge \ldots \wedge p_m \Leftrightarrow \neg P \vee (p_1 \wedge \ldots \wedge p_m) \Leftrightarrow (\neg P \vee p_1) \wedge \ldots \wedge (\neg P \vee p_m)$. Последняя конъюнкция выполнена тогда и только тогда, когда $P \wedge \neg p_1 - UNSAT, \ldots, P \wedge \neg p_m - UNSAT$. Поэтому если выбрать из заданного набора предикатов $\{p_1, \ldots, p_n\}$ те и только те предикаты $p_{i_k}, 1 \leq k \leq m$, для которых $P \wedge \neg p_{i_k} - UNSAT$, то для региона $\bigwedge_{k=1}^{m} p_{i_k}$будет выполнено $P \to \bigwedge_{k=1}^{m} p_{i_k}$.

Исходя из этих соображений будем пользоваться для вычисления региона r' абстрактного состояния (L', r') следующим правилом:

- В декартовой предикатной абстракции для множества предикатов $\{p_1, \ldots, p_n\}$ регион r' абстрактного состояния (L', r'), которое является верхним приближением множества конкретных состояний, заданных сильнейшим постусловием $SP(r, op)$ оператора op для предусловия r, равен:
    - $\perp$, если формула $SP(r, op)$ невыполнима, т.е. $SP(r, op) - UNSAT$;
    - конъюнкции предикатов $p_{i_1}, \ldots, p_{i_m}$, для каждого из которых формула $SP(r, op) \wedge \neg p_{i_k}$невыполнима, т.е.

$$\bigwedge_{\substack{p_{i_k} \in \{p_1, \ldots, p_n\}: \\ SP(r, op) \wedge \neg p_{i_k} - UNSAT}} p_{i_k},$$

если $SP(r, op) - SAT$.

Пустая дизъюнкция считается соответствующей тождественной истине ($\top$).

Формальное доказательство того, что использование этого правила всегда даёт верхнее приближение множества конкретных состояний, достижимых из (L, r) по оператору op, можно найти в статье [37], в которой оно и было впервые

---

[1] Логическая формула (в логике первого порядка) на основе теорий (SMT-формула) называется выполнимой, если можно назначить всем входящим в неё неинтерпретируемым символам, то есть символам (константам, функциям, предикатам), не являющимся пропозиционными связками (такими как $\top, \wedge, \neg$) или символами теорий (такими как $2, +, \leq$), значения из соответствующих доменов (областей определения) так, чтобы формула стала истинной.

предложено. Регион, соответствующий абстрактному состоянию (L', r'), вычисленный по приведённому правилу для множества предикатов $\pi = \{p_1, \dots, p_n\}$ и предусловия r, будем в дальнейшем обозначать через $\mathrm{post}_\pi(r, op)$.

3.9 Абстрактное дерево достижимости

Дадим теперь, наконец, определение основной структуры данных, используемой инструментами BLAST и CPAchecker для представления верхнего приближения множества всех достижимых состояний исходной программы, то есть для приближенного решения поставленной перед ними задачи проверки свойства недостижимости. *Абстрактное дерево достижимости* или *АДД* (от англ. *Abstract Reachability Tree, ART)* – это ориентированное дерево с помеченными дугами и вершинами, представляющее верхнее приближение некоторой части состояний исходной программы, достижимых из её начального состояния. Каждая вершина АДД помечена абстрактным состоянием (L, r), представленным в виде пары (метка, регион). Будем обозначать помеченные вершины АДД так: $N\colon(L, r)$ или просто N, где N — уникальный идентификатор вершины. Договоримся использовать в качестве уникальных идентификаторов вершин АДД имена меток соответствующих абстрактных состояний, приписывая к ним разделитель # и дополнительный индекс для обеспечения уникальности. Например, для вершины с меткой $(L6, x > y \wedge z \geq 0)$ будем использовать идентификатор $L6\#1$, а для вершины с меткой $(L6, \top)$ — идентификатор $L6\#2$. Дуги АДД помечаются теми же операторами, что и соответствующие дуги ГПУ. Соответствие это устанавливается во время построения АДД уже рассмотренным ранее способом по именам меток в вершинах АДД. Путь от корня до произвольной вершины в АДД может соответствовать одному из представленных в графе достижимости путей P возможного выполнения программы до достижения соответствующей метки. Регион же в вершине АДД представляет (с точностью до метки) верхнее приближение множества состояний программы, достижимых при условии её выполнения по этому пути P.

Будем называть АДД *полным*, если:

1. Его корень помечен $(L_1, \top)$, где L_1 — метка первого оператора программы, или, иначе, метка начального состояния программы.
2. АДД *замкнуто относительно постусловий*, то есть для любой его внутренней вершины $N\colon(L, r)$, такой, что $r - SAT$ $(r \not\equiv \perp)$, и для любой дуги ГПУ, исходящей из его вершины с меткой L и помеченной оператором op — $L \overset{op}{\rightarrow} L'$ — из вершины N АДД исходит

дуга с меткой op в его вершину N': (L', r'), такую что $\text{post}_\pi(r, op) \rightarrow r'$, где π — выбранное множество предикатов.

3. Для любого листа АДД N: (L, r) верно что:
    a. либо у вершины с меткой L в ГПУ нет исходящих дуг,
    b. либо $\varphi \equiv \perp$ (то же, что и $\varphi - UNSAT$),
    c. либо в АДД существует внутренняя вершина N': (L, r'), такая, что $r \rightarrow r'$.

В случае 3с говорят, что вершина N: (L, r) *покрыта* вершиной N': (L, r'), потому что выполнена вложенность соответствующих множеств состояний программы
($[\![(L, r)]\!] \subseteq [\![(L, r')]\!]$) и, как следствие, любое выполнение программы из вершины N с точностью до выбранной абстракции возможно также из вершины N'.

Полное АДД приближает сверху множество достижимых состояний программы. Интуитивно АДД представляет собой конечную развёртку ГПУ, вершины которого помечены регионами.

Полное АДД называют *безопасным по отношению к конфигурации* (E, p), где E — ошибочная метка, p — предикат над переменными программы, если для любой его вершины N: (E, r) конъюнкция $r \wedge p$ невыполнима ($r \wedge p - UNSAT$).

Полное АДД, безопасное по отношению к конфигурации $(E, \top)$, является доказательством недостижимости в исходной программе ошибочного оператора с меткой E. Это частный случай применения следующей теоремы:

Теорема 1. Пусть C — ГПУ, T — полное АДД для C, а p — предикат. Для любой вершины L из C если T безопасно по отношению к конфигурации (L, p), то ни одно конкретное состояние с меткой L и значениями переменых, обращающими p в истину, не является достижимым в программе, соответствующей C.

Доказательство этой теоремы можно найти в [38].

4 Построение абстрактных деревьев достижимости

4.1 Построение ГПУ

Формально построение ГПУ по данной простейшей программе с уникальными метками можно описать следующим образом:

- оператору присваивания op_a, помеченному меткой L_1, за которым следует оператор (или точка выхода) с меткой L_2 в ГПУ ставится в

соответствие дуга из вершины L_1 в вершину L_2, помеченная оператором op_a;

- оператору ветвления по условию c_b, помеченному меткой L_1, который передаёт управление на метку L_2 в случае выполнения условия c_b или на метку L_3 в противном случае, в ГПУ ставится в соответствие две дуги – одна из вершины L_1 в вершину L_2 с пометкой (c_b), а другая – из вершины L_1 в вершину L_3 с пометкой $!\,(c_b)$;
- оператору безусловного перехода на метку L_2, помеченному L_1, в ГПУ ставится в соответствие дуга с меткой $goto\ L_2$, направленная из вершины L_1 в вершину L_2.

Таким образом, более одной исходящей дуги могут иметь только вершины ГПУ, соответствующие оператору ветвления. Но так как в простейших программах недопустимы пустые операторы, дуги, исходящие из этих вершин, будут всегда входить в различные вершины (соответствующие меткам операторов в ветвях оператора if). Поэтому в построенном ГПУ не будет кратных дуг.

4.2 Представление предикатов

При построении АДД будем представлять предикаты с помощью логических формул. В логических формулах будем использовать функции сложения (+), вычитания ($-$), умножения на константу ($\cdot$), предикаты $>, <, =, \neq, \geq, \leq$, логические связки $\wedge$ и $\vee$, а также логическое отрицание $\neg$. Таким образом, каждому допустимому выражению простейшей программы можно будет однозначно поставить в соответствие логическую формулу, заменив используемые в этом выражении операции соответствующими функциями и предикатами .

При установлении такого соответствия делается упрощающее предположение о неограниченности диапазона возможных значений типа `int` и о соответствии конкретных аппаратных реализаций операций сложения, вычитания, умножения, сравнения и т.п. (с учётом возможных переполнений) их математическим аналогам. Поскольку на практике такие упрощающие предположения не верны, использование логических формул с функциями +, $-$, $\cdot$ и предикатами $>, <, =, \neq, \geq, \leq$ снижает точность верификации и может приводить как к выдаче инструментом верификации невыполнимых на практике примеров ошибочного пути, так и к фиктивному (основанному на ложных предположениях) доказательству недостижимости ошибочной метки.

Строго говоря, к примеру, при выполнении на компьютере с архитектурой, в которой размер типа `int` равен 32 битам, программа на языке С, аналогичная простейшей программе из примера 1, является некорректной в соответствии со свойством недостижимости ошибочной метки ERR. Вот выполнимый пример ошибочного пути:

$$(L1, \{x \mapsto 2147483647, y \mapsto -2147483648, z \mapsto 0\}) \xrightarrow{(x>y)}$$
$$(L2, \{x \mapsto 2147483647, y \mapsto -2147483648, z \mapsto 0\}) \xrightarrow{z=x-y}$$
$$(L4, \{x \mapsto 2147483647, y \mapsto -2147483648, z \mapsto -1\}) \xrightarrow{(z<0)}$$
$$(ERR, \{x \mapsto 2147483647, y \mapsto -2147483648, z \mapsto -1\})$$

На практике все существующие на сегодняшний день инструменты статической верификации, использующие предикатную абстракцию (в частности, декартову) с указанными функциями и предикатами, не гарантируют не только абсолютную, но даже самую высокую точность верификации среди вообще всех существующих инструментов. К примеру, инструмент CBMC, использующий подход BMC (ограничиваемую проверку моделей), найдёт для программы из примера 1 выполнимый пример ошибочного пути. Инструменты BLAST и CPAchecker, использующие CEGAR с предикатной абстракцией, считают приведённую в примере 1 программу корректной. Существуют, впрочем, и гибридные подходы к верификации с использованием CEGAR и предикатной абстракции, но без использования математических функций $+, -, \cdot$ и т.п., которые также могут найти для данной программы пример ошибочного пути (пример такого инструмента – SATabs). Основным преимуществом инструментов BLAST и CPAchecker является их применимость к реальным промышленным программам среднего размера (порядка 20 тыс. строк кода на С), которые практически не поддаются верификации с помощью более точных инструментов.

Выполнимость логических формул с указанными нами функциями и предикатами могут проверять инструменты проверки выполнимости (SMT-решатели). Подробнее о таких инструментах рассказано в 8 разделе данной статьи.

4.3 Задание сильнейших постусловий с помощью представления SSA

Сильнейшие постусловия можно задавать непосредственно для дуг ГПУ в виде логических формул, например, с использованием так называемого представления SSA [39]. В представлении SSA (от англ. Single State

Assignment) каждой переменной значение присваивается лишь единожды. Для этого к каждой переменной программы приписывается индекс, который увеличивается при каждом присваивании.

Предполагая, что индексы всех переменных в SSA представлении изначально равны 1, сильнейшие постусловия для операторов простейшей программы можно записать так:

$$SP\big(p(x_1, \dots, x_k, \dots, x_n), x_k = e(x_1, \dots, x_k, \dots, x_n)\big) =$$

$$p({x_1}^1, \dots, {x_k}^1, \dots, {x_n}^1)$$

$$\wedge \big({x_k}^2 = e({x_1}^1, \dots, {x_k}^1, \dots, {x_n}^1)\big), \text{индекс } x_k \text{ теперь равен } 2$$

$$SP(p(x_1, \dots, x_n), (\, c(x_1, \dots, x_n))) = p({x_1}^1, \dots, {x_n}^1) \wedge c({x_1}^1, \dots, {x_n}^1)$$

$$SP(p(x_1, \dots, x_n), goto\ L) = p({x_1}^1, \dots, {x_n}^1)$$

Здесь верхний индекс переменной указывает её индекс в представлении SSA. Аналогично сильнейшие постусловия можно выписать при других значениях индексов переменных. Если обозначить индекс произвольной переменной x_k через $\sigma(x_k)$, то:

$$SP\big(p(x_1, \dots, x_n), x_k = e(x_1, \dots, x_n)\big)$$

$$= p\big({x_1}^{\sigma(x_1)}, \dots, {x_n}^{\sigma(x_n)}\big)$$

$$\wedge \Big({x_k}^{\sigma(x_k)+1} = e\big({x_1}^{\sigma(x_1)}, \dots, {x_n}^{\sigma(x_n)}\big)\Big); \sigma(x_k) \leftarrow \sigma(x_k) + 1$$

$$SP(p(x_1, \dots, x_n), c(x_1, \dots, x_n)) = p\big({x_1}^{\sigma(x_1)}, \dots, {x_n}^{\sigma(x_n)}\big) \wedge c({x_1}^{\sigma(x_1)}, \dots, {x_n}^{\sigma(x_n)})$$

$$SP(p(x_1, \dots, x_n), goto\ L) = p({x_1}^{\sigma(x_1)}, \dots, {x_n}^{\sigma(x_n)})$$

Здесь $\sigma(x_k) \leftarrow \sigma(x_k) + 1$ означает увеличение на 1 индекса переменной x_k.

4.4 Вычисление регионов через сильнейшие постусловия

При вычислении регионов по приведённому ранее (в разделе 3.7) правилу будем использовать уточнённую SSA-индексами запись формулы конъюнкции предикатов для случая $SP(r, op) - SAT$:

$$\text{post}_{\{p_1,\dots,p_n\}}(r, op) = \bigwedge_{\substack{p_{i_k} \in \{p_1,\dots,p_n\}: \\ SP(r,op) \wedge \neg p_{i_k}({x_1}^{\sigma(x_1)},\dots,{x_n}^{\sigma(x_n)}) - UNSAT}} p_{i_k}({x_1}^{\sigma(x_1)}, \dots, {x_n}^{\sigma(x_n)}),$$

Здесь $p_{i_k}({x_1}^{\sigma(x_1)}, \dots, {x_n}^{\sigma(x_n)})$ получается из $p_{i_k}(x_1, \dots, x_n)$ приписыванием самых последних индексов SSA представления всем его переменным.

4.5 Построение АДД на основе ГПУ

Будем строить соответствующее полное АДД, последовательно перебирая пути в ГПУ с помощью обхода в глубину. Взяв начальный регион $\top$, мы построим начальную вершину АДД, соответствующую начальной вершине ГПУ. Затем для последней построенной вершины АДД $N\colon (L, r)$ будем строить переход по одной из соответствующих исходящих дуг ГПУ (из L в L' с меткой op — $L \overset{op}{\rightarrow} L'$) в новую вершину $N'\colon (L', r')$, где $r' = \text{post}_\pi(r, op)$, π — выбранное множество предикатов. Для каждой вновь построенной вершины $N'\colon (L', \varphi')$ будем проверять наличие соответствующих исходящих дуг в ГПУ, выполнимость соответствующего региона φ', а также покрытие какой-либо уже построенной вершиной $N''\colon (L', \varphi'')$, где $\varphi' \rightarrow \varphi''$. При выполнении хотя бы одного из этих условий будем прекращать построение дуг из последней вершины и возвращаться к предыдущей построенной вершине. Построение АДД будем заканчивать тогда, когда для каждой его непокрытой вершины с выполнимым регионом будут построены переходы по всем соответствующим ей исходящим дугам в ГПУ. Полученное АДД будет являться полным по построению, поэтому для проверки свойства достижимости ошибочной метки E достаточно будет проверить его безопасность по отношению к конфигурации $(E, \top)$, то есть, по сути, проверить выполнимость регионов в его вершинах с меткой E (что на самом деле уже будет сделано во время построения).

Конечность АДД обеспечивается конечностью числа дуг ГПУ и описанным способом построения, обеспечивающим, в частности, конечность числа дуг АДД, соответствующих одной дуге ГПУ, за счёт проверки покрытия вершин. Дуге ГПУ, исходящей из вершины L, может соответствовать лишь конечное число дуг АДД, не большее, чем число различных возможных абстрактных состояний с меткой L, то есть $2^n + 1$, где n — число предикатов в выбранной абстракции. Это объясняется тем, что если из какой-либо уже построенной вершины АДД $N\colon (L, r)$ исходит дуга, то другая его вершина с тем же абстрактным состоянием $N'\colon (L, r)$ окажется покрыта ранее построенной вершиной N (т.к. $\forall \varphi . \varphi \rightarrow \varphi$) и, как следствие, не будет иметь исходящих дуг.

5 Пример построения АДД

Чтобы лучше осознать основные проблемы, которые позволяет решить метод CEGAR, рассмотрим в начале процесс построения АДД по заранее заданному набору предикатов на примере уже рассмотренной ранее простейшей программы (пример 1). Для этой программы мы попытаемся с помощью полного АДД доказать недостижимость ошибочного оператора, помеченного меткой ERR. Данный простой пример наглядно демонстрирует, что подбор предикатов, подходящих для доказательства недостижимости ошибочного состояния даже в очень простых программах, является непростой задачей.

Итак. ГПУ для примера 1 представлен на рис. 1б.

Выберем множество из пяти предикатов $\pi = \{p_1, \ldots, p_5\}$:

$$p_1 \equiv x > 0$$

$$p_2 \equiv x = 0$$

$$p_3 \equiv y > 0$$

$$p_4 \equiv y = 0$$

$$p_5 \equiv z \geq 0$$

Построим АДД описанным выше способом:

1. Строим вершину $L1\#1\colon (L1, \top)$.
2. Вершина $L1\#1$имеет соответствующую исходящую дугу $L1 \xrightarrow{x>y} L2$ в ГПУ, её регион $\top - SAT$ и она, очевидно, не покрыта никакой ранее построенной вершиной.
3. Строим дугу $L1\#1\colon (L1, \top) \xrightarrow{x>y} L2\#1\colon (L2, \mathrm{post}_\pi(\top, (x > y)))$.
4. Вычисляем $\mathrm{post}_\pi(\top, (x > y))$, пользуясь соответствующим правилом:

   предполагаем все индексы SSA равными 1 и пользуемся представлением SSA для записи сильнейшего постусловия:

   $$SP\big(\top, (x > y)\big) \equiv x_1 > y_1 - SAT\ (x_1 = 1, y_1 = 0)$$

   (в скобках указана модель для выполнимой формулы),

   значит $\mathrm{post}_\pi(\top, (x > y)) \not\equiv \bot$

$$p_1: SP\big(\top, (x > y)\big) \wedge \neg p_1$$

$$\equiv \top \wedge (x_1 > y_1) \wedge \neg(x_1 > 0)$$

$$- SAT(x_1 = 0, y_1 = -1)$$

значит p_1 не входит в искомую конъюнкцию

$$p_2: SP\big(\top, (x > y)\big) \wedge \neg p_2$$

$$\equiv \top \wedge (x_1 > y_1) \wedge \neg(x_1 = 0)$$

$$- SAT(x_1 = 1, y_1 = 0)$$

$$p_3: SP\big(\top, (x > y)\big) \wedge \neg p_3$$

$$\equiv \top \wedge (x_1 > y_1) \wedge \neg(y_1 > 0)$$

$$- SAT(x_1 = 1, y_1 = 0)$$

$$p_4: SP\big(\top, (x > y)\big) \wedge \neg p_4$$

$$\equiv \top \wedge (x_1 > y_1) \wedge \neg(y_1 = 0)$$

$$- SAT(x_1 = 2, y_1 = 1)$$

$$p_5: SP\big(\top, (x > y)\big) \wedge \neg p_5$$

$$\equiv \top \wedge (x_1 > y_1) \wedge \neg(z_1 \geq 0)$$

$$- SAT(x_1 = 1, y_1 = 0, z_1 = -1)$$

Искомая конъюнкция пуста. Значит, $\mathrm{post}_\pi\big(\top, (x > y)\big) \equiv \top$.

5. Переходим к вершине $L2\#1: (L2, \top)$. Она имеет соответствующую исходящую дугу $L2 \xrightarrow{z = x - y} L4$ в ГПУ, её регион $\top - SAT$ и она не покрыта никакой ранее построенной вершиной.
6. Строим дугу $L2\#1: (L2, \top) \xrightarrow{z = x - y} L4\#1: (L4, \mathrm{post}_\pi(\top, z = x - y))$.
7. Вычисляем $\mathrm{post}_\pi(\top, z = x - y)$, пользуясь правилом: полагаем индексы переменных равными 1 и пользуемся представлением SSA для записи сильнейшего постусловия:

$$SP(\top, z = x - y) \equiv \top \wedge z_2$$

$$= x_1 - y_1 - SAT(x_1 = 1, y_1 = 2, z_2 = -1),$$

значит $\mathrm{post}_\pi(\top, z = x - y) \not\equiv \bot$

$$p_1: SP(\top, z = x - y) \wedge \neg p_1 \equiv \top \wedge z_2 = x_1 - y_1 \wedge \neg(x_1 > 0) - SAT(x_1 = 0, y_1 = 0, z_2 = 0)$$

...

$$p_5: SP(\top, z = x - y) \wedge \neg p_5 \equiv \top \wedge z_2 = x_1 - y_1 \wedge \neg(z_2 \geq 0) - SAT(x_1 = 0, y_1 = 1, z_2 = -1)$$

Значит, $\mathrm{post}_\pi(\top, z = x - y) \equiv \top$.

8. Переходим к вершине $L4\#1: (L4, \top)$. Дуга $L4 \xrightarrow{!(z<0)} L5$в ГПУ, регион $\top - SAT$, вершина не покрыта.
9. Строим дугу $L4\#1: (L4, \top) \xrightarrow{!(z<0)} L5\#1: (L5, \mathrm{post}_\pi(\top, !\,(z < 0)))$.
10. Вычисляем $\mathrm{post}_\pi(\top, !\,(z < 0))$ по правилу:

    полагаем индексы переменных равными 1 и пользуемся представлением SSA:

$$SP(\top, !\,(z < 0)) \equiv \top \wedge \neg(z_1 < 0) - SAT(z_1 = 0)$$

$$p_1: SP(\top, !\,(z < 0)) \wedge \neg p_1 \equiv \top \wedge \neg(z_1 < 0) \wedge \neg(x_1 > 0) - SAT(z_1 = x_1 = 0)$$

...

$$p_5: SP(\top, !\,(z < 0)) \wedge \neg p_5 \equiv \top \wedge \neg(z_1 < 0) \wedge \neg(z_1 \geq 0) \equiv \neg(z_1 < 0) \wedge (z_1 < 0) - UNSAT$$

    Значит, $\mathrm{post}_\pi(\top, !\,(z < 0)) \equiv z \geq 0$

11. Переходим к вершине $L5\#1: (L5, z \geq 0)$. Эта вершина не имеет соответствующих исходящих дуг в ГПУ (соответствующая ей вершина $L5$ не имеет исходящих дуг).
12. Возвращаемся к предыдущей построенной вершине $L4\#1$. Рассматриваем следующую соответствующую ей исходящую дугу $L4 \xrightarrow{(z<0)} ERR$ в ГПУ.

13. Строим дугу $L4\#1: (L64\top) \xrightarrow{(z<0)} ERR\#1: \left(ERR, \text{post}_{\pi}\left(\top, (z < 0)\right)\right)$.
14. Вычисляем $\text{post}_{\pi}\left(\top, (z < 0)\right)$. Получаем $\text{post}_{\pi}\left(\top, (z < 0)\right) = \top$.
15. Переходим к вершине $ERR\#1: (ERR, \top)$. Рассматриваем дугу $ERR \xrightarrow{goto\ ERR} ERR$в ГПУ. Регион $\top - SAT$, вершина не покрыта.
16. Строим дугу $ERR\#1: (ERR, \top) \xrightarrow{goto\ ERR} ERR\#2: (ERR, \text{post}_{\pi}(\top, goto\ ERR))$.
17. Вычисляем результат $\text{post}_{\pi}(\top, goto\ ERR) = \top$.
18. Переходим к вершине $ERR\#2(ERR, \top)$. Она имеет исходящую дугу

    $ERR \xrightarrow{goto\ ERR} ERR$в ГПУ. Регион $\top - SAT$. Но в АДД есть уже построенная вершина $ERR\#1: (ERR, \top)$, такая, что $\top \rightarrow \top$. Значит, вершина $ERR\#2$ покрыта вершиной $ERR\#1$.
19. Возвращаемся к предыдущей построенной вершине $ERR\#1$. Соответствующая ей единственная исходящая дуга ГПУ ($ERR \xrightarrow{goto\ ERR} ERR$) уже рассмотрена.
20. Возвращаемся к предыдущей построенной вершине $L4\#1$. Все соответствующие ей исходящие дуги ГПУ $L4 \xrightarrow{!(z<0)} L5$ и $L4 \xrightarrow{(z<0)} ERR$ уже рассмотрены.
21. Возвращаемся к предыдущей построенной вершине $L2\#1$. Соответствующая ей исходящая дуга ГПУ $L2 \xrightarrow{z\,=\,x\,-\,y} L4$ уже рассмотрена.
22. Возвращаемся к предыдущей построенной вершине $L1\#1$. Рассматриваем следующую ещё не рассмотренную соответствующую ей исходящую дугу в ГПУ $L1 \xrightarrow{!(x>y)} L3$.
23. Строим дугу $L1\#1: (L3, \top) \xrightarrow{!(x>y)} L3\#1: (L3, \text{post}_{\pi}(\top, !\,(x > y)))$.
24. Вычисляем регион $\text{post}_{\pi}(\top, !\,(x > y)) = \top$.

25. Переходим к вершине $L3\#1:(L3,\top)$. Рассматриваем дугу $L3 \xrightarrow{z=y-x} L4$в ГПУ. Регион $\top - SAT$, вершина $L3\#1$ не покрыта.
26. Строим дугу $L3\#1:(L5,\top) \xrightarrow{z=y-x} L4\#2:(L4,\mathrm{post}_\pi(\top,z=y-x))$.
27. Вычисляем регион $\mathrm{post}_\pi(\top,z=y-x)=\top$.
28. Переходим к вершине $L4\#2:(L4,\top)$. Она покрыта вершиной $L4\#1:(L4,\top)$.
29. Убеждаемся, что для каждой непокрытой вершины АДД с выполнимым регионом построены переходы по всем соответствующим ей исходящим дугам в ГПУ. Таким образом, построение АДД закончено.

Полученное АДД показано на рис. 10. Регионы, соответствующие вершинам дерева, показаны в прямоугольниках рядом с вершинами. Покрытие обозначено пунктирной стрелкой, помеченной словами “covered by”, в направлении от покрываемой вершины к покрывающей.

Очевидно, что построенное АДД *не* является безопасным по отношению к конфигурации $(ERR,\top)$ и таким образом не позволяет доказать недостижимость ошибочной метки ERR. В то же время легко видно, что (в рамках наших упрощающих предположений) в исходной программе эта метка не является достижимой.

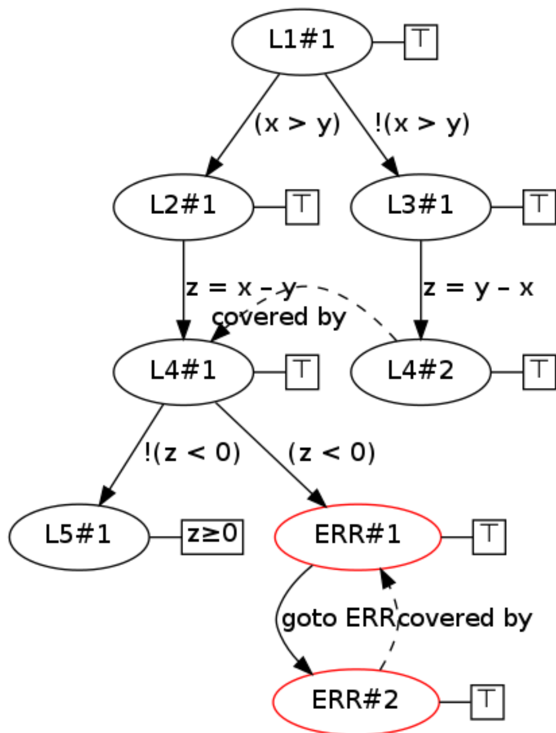


Рис. 10. Абстрактное дерево достижимости для заранее заданного набора предикатов $b_1 - b_5$.

На рассмотренном примере хорошо видно, что подбор набора предикатов, подходящего для доказательства недостижимости ошибочного состояния, является непростой задачей. Подобрать нужные предикаты заранее трудно, так как часто они сильно зависят от конкретной программы. Поэтому исследования в области методов поиска набора предикатов для абстракции ведутся в основном в направлении получения новых предикатов на основе исходной программы и какой-либо уже построенной для неё абстракции (например, АДД), которая не позволила доказать корректность программы относительно свойства недостижимости.

6 Метод CEGAR

6.1 Контрпример

В этом разделе *контрпримером* будем называть некоторый путь в ГПУ из его начальной вершины в вершину, помеченную ошибочной меткой. Основное отличие этого понятия от примера ошибочного пути в графе достижимости состоит в том, что контрпример не включает состояния программы. Это, в частности, означает, что контрпример может не соответствовать никакому пути в графе достижимости. В таком случае контрпример называют *ложным*. Для рассмотренной простейшей программы (пример 1) ложным контрпримером является, например, такой путь:

$$L1 \xrightarrow{x = nondet} L2 \xrightarrow{y = nondet} L3 \xrightarrow{(x>y)} L4 \xrightarrow{z = x - y} L6 \xrightarrow{(z<0)} ERR$$

В этом разделе мы рассмотрим метод CEGAR – Counter Example Guided Abstraction Refinement, то есть метод уточнения абстракции по контрпримеру, суть которого, как следует из названия, заключается в получении новых предикатов на основе ложного контрпримера. Ложный контрпример получают в результате построения АДД, которое не позволят доказать корректность исходной программы относительно свойства недостижимости.

6.2 Уточнение абстракции на примере

Рассмотрим сначала простейшую программу на рис. 11:

```
void main()

{

      int x = nondet;

      L1: x = 0;

      L2: if (x < 0)

      ERROR: goto ERROR;

}
```

Рис 11. Простейшая программа

Будем считать, что для неё построено АДД для пустого набора предикатов, так что во всех его вершинах регионы абстрактных состояний равны $\top$.

Соответствующий простейший ложный контрпример: $L1 \xrightarrow{x=0} L2 \xrightarrow{(x<0)} ERROR$. Запишем для первого перехода в этом пути сильнейшее постусловие: $SP(\top, x = 0) \equiv (x^2 = 0)$ (верхние индексы, как и ранее, обозначают индексы в представлении SSA). Запишем теперь постусловие для второго перехода, подставив в него вместо региона $\top$ первое сильнейшее постусловие: $SP(SP(\top, x = 0), x < 0) \equiv (x^2 = 0) \wedge (x^2 < 0)$. Полученная конъюнкция состоит из двух частей. Первая задаёт состояние переменной x перед выполнением оператора условного перехода, а вторая – дополнительные ограничения, наложенные на переменную x в результате этого перехода. Эта конъюнкция как логическая формула является невыполнимой.

Вспомнив правила построения абстрактных состояний с помощью оператора post_π, можно попытаться подобрать новый подходящий предикат для исключения найденного ложного контрпримера из АДД, исходя из следующих соображений:

При вычислении региона абстрактного состояния $(L2, \text{post}_\pi(\top, x = 0))$ новый предикат должен войти в результирующую конъюнкцию, чтобы изменить (уточнить) абстрактное состояние. Для этого должно выполняться условие $SP(\top, x = 0) \rightarrow p(x^2)$, или в более общем виде, $SP(\top, op_1) \rightarrow p(x^{\sigma(x)})$, где $p(x)$ – новый предикат, $\sigma(x)$ — значение SSA-индекса переменной x в состоянии $(L2, \text{post}_\pi(\top, op_1))$.

Исходя из предыдущего соображения, $p(x)$ войдёт в регион абстрактного состояния $(L2,\ \text{post}_\pi(\top, x = 0))$. Будем считать $\text{post}_\pi(\top, x = 0) \equiv p(x)$. При вычислении абстрактного состояния $(L3,\ \text{post}_\pi(p(x),\ x < 0))$ нужно получить результат $\bot$. Это покажет недостижимость метки $ERROR$. Для этого должно выполняться условие $SP(p(x), x < 0) - UNSAT$, то есть $p(x^1) \wedge (x^1 < 0) - UNSAT$. В более общем виде: $SP(p(x), op_2) - UNSAT$.

- В условиях $SP(\top, x = 0) \rightarrow p(x^2)$ и $SP(p(x), x < 0) - UNSAT$, то есть $(x^2 = 0) \rightarrow p(x^2)$ и $p(x^1) \wedge (x^1 < 0) - UNSAT$ в предикат $p(x)$ подставляются, вообще говоря, различные переменные (в данном случае x^1 и x^2). Однако эти условия можно рассмотреть и совместно, если воспользоваться конъюнкцией $SP(SP(\top, x = 0), x < 0) \equiv (x^2 = 0) \wedge (x^2 < 0)$ и взять предикат $p(x)$ только над общими переменными обеих частей этой конъюнкции. Тогда, взяв условия $(x^2 = 0) \rightarrow p(x^2)$ и $p(x^2) \wedge (x^2 < 0) - UNSAT$, мы можем найти удовлетворяющий им предикат в виде $p(x^2)$, то есть в общем случае $p(x^{\sigma(x)})$. При этом простое переименование переменных $x_2 \rightarrow x_1$ во

втором условии даст в точности нужное исходное условие $p(x^1) \wedge (x^1 < 0) - UNSAT$.

В более общем случае для конъюнкции $SP(SP(\top, op_1), op_2) \equiv SP(\varphi, op_2) \equiv \varphi \wedge \psi$ следует искать предикат ρ над общими переменными φ и ψ такой, что $\varphi \rightarrow \rho$ и $\rho \wedge \psi - UNSAT$. Для такого ρ будут выполняться условия

$SP(\top, op_1) \rightarrow \rho(x_{i_1}^{\sigma(x_{i_1})}, \ldots, x_{i_m}^{\sigma(x_{i_m})})$ и $SP\big(\rho\big(x_{i_1}, \ldots, x_{i_m}\big), op_2\big) - UNSAT$,

где $x_{i_1}, \ldots, x_{i_m}$ — общие переменные, используемые как в операторах op_1 и op_2.

Итого для постусловия вида $SP(SP(\top, op_1), op_2) \equiv SP(\varphi,\ op_2) \equiv \varphi \wedge \psi$ и искомого предиката ρ имеем следующие три условия:

- $\varphi \rightarrow \rho$
- $\rho \wedge \psi - UNSAT$
- ρ — формула над общими переменными, входящими в формулы φ и ψ.

Задача поиска такого предиката ρ является известной [8, 19, 53, 54, 55, 56, 57, 58] задачей построения интерполянта Крейга для конъюнкции двух логических формул (в данном случае, двух SMT-формул без кванторов). Такие предикаты при определенных ограничениях можно искать с помощью специальных инструментов – интерполирующих решателей. Для дальнейшего использования с целью уточнения абстракции предикат ρ при этом должен быть получен в рамках логики нулевого порядка, то есть не содержать кванторов. В этом случае новый предикат для абстракции (то есть без индексов) можно получить из ρ, опустив индексы всех входящих в него переменных.

6.3 Сильнейшее постусловие пути

Итак, вернувшись к рассмотренному в разделе 5 примеру, заметим, что построенное АДД не только не доказывает недостижимость ошибочной метки ERR, но также позволяет легко получить ложный контрпример, рассмотрев какой-либо путь из своей начальной вершины (соответствующей начальной вершине ГПУ) до одной из вершин с выполнимым регионом, помеченных ошибочной меткой ERR. По построению этот путь будет соответствовать

некоторому пути в ГПУ из его начальной вершины в вершину с ошибочной меткой.

Предположим вначале, что полученный контрпример представляет собой одно из реально возможных выполнений исходной программы, приводящих к достижению ею ошибочного состояния. Это означает, что существует соответствующий пример ошибочного пути в графе достижимости. Чтобы указать путь в графе достижимости, нужно найти состояния программы, соответствующие вершинам этого пути. Эти состояния содержат означивания переменных программы непосредственно до выполнения каждого оператора найденного контрпримера.

Найденный контрпример, то есть путь P в ГПУ, с другой стороны, состоит из последовательности меток и операторов исходной программы, выполнение которых по нашему предположению приводит к достижению метки ERR. Вспомним, что данное нами определение сильнейшего постусловия оператора op при данном предусловии p позволяет получить предикат, задающий (вместе с некоторой меткой) ограничения на состояния программы, непосредственно достижимые из состояний, заданных предусловием p (и некоторой меткой), в результате выполнения оператора op. Пусть мы имеем два последовательно выполняемых оператора op_1 и op_2 ($op_1; op_2$). Если в качестве предусловия p_2 для оператора op_2 взять сильнейшее постусловие оператора op_1 при некотором предусловии p ($p_2 = SP(p, op_1)$), то сильнейшее постусловие оператора op_2 при предусловии p_2 ($SP(p_2, op_2) = SP(SP(p, op_1), op_2)$) будет задавать состояния программы, достижимые из состояний, заданных предусловием p, в результате последовательного выполнения операторов op_1 и op_2 ($op_1; op_2$). Аналогично можно построить сильнейшее постусловие для трех и более последовательно выполняемых операторов $op_1; \dots; op_n (n \geq 3)$. Имея теперь последовательность операторов найденного пути $P: L_1: op_1; L_2: op_2; \dots; L_n: op_n$, мы можем по индукции определить сильнейшее постусловие для всей этой последовательности операторов при некотором предусловии r_0, задающем начальные состояния. Данное сильнейшее постусловие будет представлять собой предикат, задающий состояния, достижимые из какого-либо начального состояния по этому пути P. Так как P — путь в вершину с ошибочной меткой, то полученный предикат будет задавать ошибочные состояния, достижимые из какого-либо начального состояния по пути P. Возьмём в качестве r_0 тождественную истину $r_0 \equiv \top$. Получим:

$$
\begin{aligned}
SP(P) \equiv SP(r_0, P) \equiv SP(\top, P) &\equiv SP(\top, op_1; op_2; \dots; op_{n-1}; op_n) \\
&\stackrel{\text{def}}{=} SP(SP(\top, op_1; op_2; \dots; op_{n-1}), op_n) = \cdots \\
&= SP(SP(\dots SP(SP(\top, op_1), op_2) \dots, op_{n-1}), op_n)
\end{aligned}
$$

Здесь через $SP(P)$ мы обозначили сильнейшее постусловие пути P.

Если представить предикат $SP(P)$ в виде логической формулы, можно говорить о выполнимости, либо невыполнимости этой формулы, а также в случае её выполнимости — о модели, обращающей эту формулу в тождественную истину.

Если формула $SP(P)$ выполнима, то это означает, что она задаёт хотя бы одно достижимое из начального по пути P ошибочное состояние. Таким образом, исходная программа не является корректной относительно свойства недостижимости, а найденный контрпример P не является ложным. Кроме этого, при записи постусловий с помощью представления SSA, модель этой формулы, обращающая её в тождественную истину, будет задавать значения некоторых переменных программы непосредственно до выполнения каждого оператора найденного контрпримера. Эти значения можно использовать для указания соответствующего примера ошибочного пути в графе достижимости. Инструменты верификации (в частности, BLAST и CPAchecker) в таком случае, как правило, просто выдают найденный контрпример P в качестве результата своей работы.

Если формула $SP(P)$ невыполнима, можно говорить о том, что путь P до ошибочной метки, полученный в результате построения АДД, не соответствует никакому возможному выполнению исходной программы. В этом случае инструменты верификации обычно пытаются получить на основе невыполнимой формулы $SP(P)$ новые предикаты, а затем построить на основе объединения множеств старых и новых предикатов новое АДД, в котором ошибочная метка будет недостижима, по крайней мере, по пути P. В случае BLAST и CPAchecker получение множества новых предикатов основано на использовании интерполяции Крейга для невыполнимой конъюнкции двух или более логических формул.

6.4 Интерполяция Крейга

В математической логике интерполяционная теорема Крейга утверждает, что если для двух логических формул φ и ψ общезначима (тождественно истинна на любой модели) импликация $\varphi \to \psi$, то существует логическая формула ρ, называемая *интерполянтом Крейга*, которая удовлетворяет трём условиям:

1. $\varphi \to \rho$;
2. $\rho \to \psi$;
3. каждый неинтерпретируемый (не заданный какой-либо логической теорией) символ в формуле ρ является общим для формул φ и ψ.

Теорема была впервые доказана для логики первого порядка У. Крейгом в 1957 году [40]. Третье условие для формулы ρ в этой теореме, в частности, означает, что ρ может содержать только те переменные и

неинтерпретируемые функции, которые являются общими для формул φ и ψ, то есть входят в обе эти формулы.

Подставим вместо формулы ψ в условие интерполяционной теоремы формулу вида $\neg\psi_1$. Тогда общезначимость $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$, $(\neg\varphi \vee \psi)|_{\psi=\neg\psi_1} \equiv \neg\varphi \vee \neg\psi_1$ будет эквивалентна невыполнимости $\neg(\neg\varphi \vee \neg\psi_1) \equiv \varphi \wedge \psi_1$. Значит можно сказать, что соответствующий интерполянт ρ существует для невыполнимой конъюнкции двух логических формул $\varphi \wedge \psi_1$ и удовлетворяет трём условиям:

1. $\varphi \rightarrow \rho$
2. $\rho \rightarrow \neg\psi_1 \Leftrightarrow \neg\rho \vee \neg\psi_1 \Leftrightarrow (\rho \wedge \psi_1 - UNSAT)$
3. каждый неинтерпретируемый (не заданный какой-либо логической теорией) символ в формуле ρ является общим для формул φ и ψ_1.

6.5 Уточнение абстракции

Рассмотрим теперь невыполнимую формулу $SP(P)$. Она получена в результате индуктивного выписывания постусловий операторов в пути $P: L_1: op_1; L_2: op_2; \ldots; L_n: op_n$, причем при использовании представления SSA каждое следующее постусловие получается из предыдущего добавлением к нему через конъюнкцию новой логической подформулы[2] (тождественной истины $\top$ в случае оператора `goto`) с возможным увеличением индекса одной из переменных программы в представлении SSA (см. соответствующие формулы для $SP(p, op)$ в разделе 4.3). Это означает, что полученную формулу $SP(P)$ можно представить в виде $SP(P) = \top \wedge \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_n = \psi_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n)$, где подформулы $\psi_1, \psi_2, \ldots, \psi_n$ соответствуют операторам $op_1; op_2; \ldots; op_n$ пути P.

Рассмотрим интерполянт Крейга ρ_1 для невыполнимой конъюнкции формул ψ_1 и $\psi_2 \wedge \ldots \wedge \psi_n$. Так как по определению интерполянта в формулу ρ_1 могут входить только переменные, общие для формул ψ_1и $\psi_2 \wedge \ldots \wedge \psi_n$, то учитывая формулы, используемые для построения сильнейших постусловий, каждая переменная в формулу ρ_1 может входить только с одним индексом представления SSA. Это означает, что соответствующим переименованием переменных формуле ρ_1 можно поставить в соответствие новый предикат p_1' над переменными программы, опустив индексы представления SSA у входящих в эту формулу переменных. Это можно записать так:

$$p_1'\left({x_1}^{\sigma(x_1)}, \ldots, {x_n}^{\sigma(x_n)}\right) = \rho_1$$

[2] Подформулой называется часть формулы, сама являющаяся формулой.

То есть после приписывания переменным программы в новом предикате p_1' соответствующих индексов представления SSA (имеющихся после вычисления $SP(\top, op_1)$), из этого предиката получится интерполянт ρ_1.

Предположим теперь, что мы строим АДД, имея множество предикатов π_1', включающее в себя p_1'. Тогда при вычислении региона

$$\mathrm{post}_{\pi_1'}(\top, op_1) = \bigwedge_{\substack{p_{j_k} \in \pi_1': \\ SP(\top, op_1) \wedge \neg p_{j_k}\left({x_1}^{\sigma(x_1)}, \dots, {x_n}^{\sigma(x_n)}\right) - UNSAT}} p_{j_k}$$

из условия (1) для интерполянта Крейга ρ_1 ($\psi_1 \to \rho_1$), учитывая, что по построению $\psi_1 = SP(\top, op_1)$, мы получим

$$SP(\top, op_1) \to \rho_1 \Leftrightarrow$$

$$\neg SP(\top, op_1) \vee \rho_1 \Leftrightarrow$$

$$\neg\neg(\neg SP(\top, op_1) \vee \rho_1) \Leftrightarrow$$

$$\neg(SP(\top, op_1) \wedge \neg\rho_1) \Leftrightarrow$$

$$SP(\top, op_1) \wedge \neg p_1'\left({x_1}^{\sigma(x_1)}, \dots, {x_n}^{\sigma(x_n)}\right) - UNSAT)$$

Это означает, что

$$\mathrm{post}_{\pi_1'}(\top, op_1) = p_1' \wedge r_1, \qquad r_1 - \text{некоторая конъюнкция}$$

То есть новый предикат p_1' из множества π_1' обязательно входит в регион для второй вершины L_2 на пути P.

Теперь рассмотрим оставшийся путь $P'\colon L_2\colon op_2; \dots; L_n\colon op_n$.

$$SP\left(\mathrm{post}_{\pi_1'}(\top, op_1), P'\right)$$
$$= p_1'({x_1}^1, \dots, {x_n}^1) \wedge r_1({x_1}^1, \dots, {x_n}^1) \wedge (\psi_2' \wedge \dots \wedge \psi_n')$$
$$= p_1'({x_1}^1, \dots, {x_n}^1) \wedge (\psi_2' \wedge \dots \wedge \psi_n') \wedge r_1({x_1}^1, \dots, {x_n}^1)$$

Переименуем переменные в этой формуле, начав индексирование с последних индексов SSA, которые были получены для $SP(\top, op_1)$. После замены переменных вида ${x_1}^1 \to {x_1}^{\sigma(x_1)}, {x_2}^1 \to {x_2}^{\sigma(x_2)}, \dots, {x_n}^1 \to {x_n}^{\sigma(x_n)}, {x_1}^2 \to {x_1}^{\sigma(x_1)+1}, {x_2}^2 \to {x_2}^{\sigma(x_2)+1}, \dots$ получим:

$$p_1'\left({x_1}^{\sigma(x_1)}, \dots, {x_n}^{\sigma(x_n)}\right) \wedge (\psi_2 \wedge \dots \wedge \psi_n) \wedge r_1\left({x_1}^{\sigma(x_1)}, \dots, {x_n}^{\sigma(x_n)}\right)$$
$$= \rho_1 \wedge (\psi_2 \wedge \dots \wedge \psi_n) \wedge r_1$$

Эта формула выполнима тогда и только тогда, когда выполнима формула $SP\left(\mathrm{post}_{\pi_1'}(\top, op_1), P'\right)$. Поэтому из условия (2) для интерполянта Крейга ρ_1

$(\rho_1 \wedge (\psi_2 \wedge \dots \wedge \psi_n) - UNSAT)$, получаем, что $SP\left(\mathrm{post}_{\pi_1'}(\top, op_1), P'\right) - UNSAT$. Таким образом, невыполнимость сильнейшего постусловия $SP(\top, P)$ для пути P сохраняется для оставшейся части пути P', взятой начиная со второго оператора op_2. Отметим, что для невыполнимости $SP\left(\mathrm{post}_{\pi_1'}(\top, op_1), P'\right)$ существенно лишь присутствие в π_1' нового предиката p_1'. Помимо него π_1' может включать и другие произвольные предикаты, в частности, предикаты, использованные при построении предыдущего АДД $(\pi \subset \pi')$. Это позволяет в ходе уточнения абстракции всё время лишь включать в соответствующие множества новые предикаты, не исключая при этом уже найденных новых предикатов. Поэтому будем предполагать $\pi \subset \pi_1' = \pi \cup \{p_1'\} \subset \pi_2' = \pi \cup \{p_1', p_2'\} \subset \cdots$.

Далее аналогично тому, как мы рассмотрели невыполнимую формулу $SP(\top, P)$, рассмотрим последовательно невыполнимые формулы $SP\left(\mathrm{post}_{\pi_1'}(\top, op_1), P'\right), SP\left(\mathrm{post}_{\pi_2'}\left(\mathrm{post}_{\pi_2'}(\varphi_1, op_1), op_2\right), P''\right), \dots$ Через $\pi_1', \pi_2', \dots$ обозначены соответствующие новые множества предикатов, а через $P', P'', \dots$ — соответствующие оставшиеся части пути P. Обозначим $\mathrm{post}_{\pi_1'}(\varphi_1, op_1) = r_1'$, $\mathrm{post}_{\pi_2'}(\mathrm{post}_{\pi_2'}(\varphi_1, op_1), op_2) = r_2', \dots$ Рано или поздно, то есть, по крайней мере, при рассмотрении $SP\left(\mathrm{post}_{\pi_{n-1}'}(\dots, op_{n-1}), op_n\right) = SP(r_{n-1}', op_n) - UNSAT$ мы по правилу вычисления регионов (см. секцию 4) получим $\mathrm{post}_{\pi_{n-1}'}(r_{n-1}', op_n) = \perp$ (независимо от множества предикатов π_{n-1}'). Условие $SP(r_{k-1}', op_k) - UNSAT$ может быть выполнено и при меньших значениях $k < n$, но т.к. путь P состоит из конечного числа помеченных операторов, оно обязательно будет выполнено, по крайней мере, для самого последнего перехода $L_n \xrightarrow{op_n} ERR$ при $k = n$. Это означает, что в новом АДД, которое затем будет построено для набора предикатов $\pi_{n-1}' = \pi \cup \{p_1', \dots, p_{k-1}'\}$ (здесь $k \leq n$: $SP(r_{k-1}', op_k) - UNSAT$) либо не будет присутствовать соответствующая вершина с ошибочной меткой ERR, либо регион в этой вершине будет невыполним.

При построении АДД описанным ранее способом одному и тому же пути в ГПУ не может соответствовать более одного пути в АДД, так как все пути в АДД по построению различаются хотя бы одной дугой. Поэтому если множество используемых предикатов будет изменяться только за счёт включения в него новых предикатов, то в результате работы инструмента верификации больше не будет получен контрпример, содержащий уже рассмотренный невыполнимый путь P до ошибочной метки. Таким образом, однократное успешное уточнение абстракции позволяет исключить из АДД, по крайней мере, один путь, не соответствующий какому-либо возможному выполнению исходной программы.

На практике в инструментах BLAST и CPAchecker реализация процесса уточнения абстракции отличается от рассмотренного индуктивного варианта.

После получения формулы $SP(\top, P)$ для сильнейшего постусловия пути P в виде $\top \wedge \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_n$ инструмент верификации всего за одно обращение к интерполирующему решателю получает все интерполянты Крейга, необходимые для уточнения набора предикатов.

Для n логических формул $\psi_1, \ldots, \psi_n$, конъюнкция которых невыполнима ($\psi_1 \wedge \ldots \wedge \psi_n - UNSAT$), *набором интерполянтов Крейга* называется набор из $n-1$ формулы $\rho_1, \ldots, \rho_{n-1}$, удовлетворяющих условиям:

1. $\psi_1 \to \rho_1, \psi_1 \wedge \psi_2 \to \rho_2, \ldots, \psi_1 \wedge \ldots \wedge \psi_{n-1} \to \rho_{n-1}$
2. $\rho_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n) - UNSAT, \ldots, \rho_k \wedge (\psi_{k+1} \wedge \ldots \wedge \psi_n) - UNSAT, \ldots, \rho_{n-1} \wedge \psi_n - UNSAT$
3. для любого $k = \overline{1, n-1}$ каждый неинтерпретируемый символ в формуле ρ_k является общим для формул $\psi_1 \wedge \ldots \wedge \psi_k$ и $\psi_{k+1} \wedge \ldots \wedge \psi_n$.

При $n \geq 2$ для невыполнимой конъюнкции формул ($\psi_1 \wedge \ldots \wedge \psi_n - UNSAT$) можно также определить набор индуктивных интерполянтов Крейга $\rho_1, \ldots, \rho_{n-1}$, для которых вместо условий (1) выполняются условия: $\psi_1 \to \rho_1, \rho_1 \wedge \psi_2 \to \rho_2, \ldots, \rho_{k-1} \wedge \psi_k \to \rho_k, \ldots, \rho_{n-2} \wedge \psi_{n-1} \to \rho_{n-1}$. Набор индуктивных интерполянтов является частным случаем набора интерполянтов Крейга. Действительно, предположим, что не выполнено одно из условий (1) для некоторого $1 \leq k \leq n-1$, то есть не выполнено $\psi_1 \wedge \ldots \wedge \psi_k \to \rho_k$. По определению импликации это означает, что $\psi_1, \ldots, \psi_k$ истины, в то время как ρ_k — ложь. Но тогда по условию для индуктивных интерполянтов учитывая, что $\psi_1, \ldots, \psi_k$ истины, получим $\psi_1 \to \rho_1, \rho_1 \to \rho_2, \ldots, \rho_{k-1} \to \rho_k$, то есть ρ_k — истина, что противоречит предположению. Легко видеть, что для индуктивных интерполянтов Крейга одновременное получение всех интерполянтов с помощью одного вызова интерполирующего решателя по определению эквивалентно уже рассмотренному индуктивному способу их получения. Для общего случая интерполянтов Крейга при некоторых ограничениях на формулы $\psi_1, \ldots, \psi_n$ и с существенным использованием условия (3) можно доказать исключение соответствующего невыполнимого ошибочного пути из нового АДД, как это делается, к примеру, в статье [41].

Таким образом, на практике успешное уточнение абстракции также означает исключение из АДД, по крайней мере, одного пути, не соответствующего какому-либо возможному выполнению исходной программы.

6.6 Цикл CEGAR

Верификация исходной программы методом CEGAR в инструментах BLAST и CPAchecker представляет собой цикл, на каждой итерации которого выполняются следующие последовательные шаги:

1. Построение АДД по текущему набору предикатов.

2. Проверка безопасности построенного АДД по отношению к конфигурации $(E, \top)$, где E — ошибочная метка.
   а) Если построенное АДД оказывается безопасным по отношению к конфигурации $(E, \top)$, то оно считается доказательством недостижимости ошибочной инструкции в исходной программе. Таким образом, инструмент верификации успешно завершает работу, доказав корректность программы относительно свойства недостижимости.
   б) Иначе происходит переход к шагу 3.
3. В построенном АДД находится вершина $N\colon(E, \varphi)$, $\varphi \neq \bot$. Строится соответствующий путь P из начальной вершины АДД в вершину N. Путь P затем рассматривается в качестве пути контрпримера. Вычисляется сильнейшее постусловие этого пути — $SP(P) \equiv SP(\top, P)$.
4. Проверка выполнимости $SP(P)$.
   а) Если постусловие $SP(P)$ как логическая формула оказывается выполнимой, то найденный контрпример P выдаётся в качестве результата работы инструмента, как пример выполнения исходной программы, при котором достигается заданная ошибочная метка.
   б) Иначе происходит переход к шагу 5.
5. Выполняется уточнение текущего набора предикатов с использованием невыполнимого ошибочного пути P с помощью интерполяции Крейга. При успешном уточнении набора предикатов происходит переход к шагу 1.

Таким образом, цикл CEGAR либо завершается на шаге 2 (с доказательством недостижимости), либо на шаге 4 (с контрпримером пути до ошибочной метки), либо выполняется вплоть до исчерпания выделенных инструменту верификации ресурсов (памяти и процессорного времени), то есть теоретически бесконечно. Бесконечное выполнение возможно, например, при

существовании подходящей бесконечной последовательности предикатов, выводимых на шаге 5.

7 Пример применения метода CEGAR

Рассмотрим применение метода CEGAR для доказательства недостижимости ошибочной метки в программе из примера 1.

1. Выберем пустой начальный набор предикатов $\pi = \emptyset$. Построим АДД, соответствующее пустому набору предикатов. Получим результат, показанный на рис. 12.

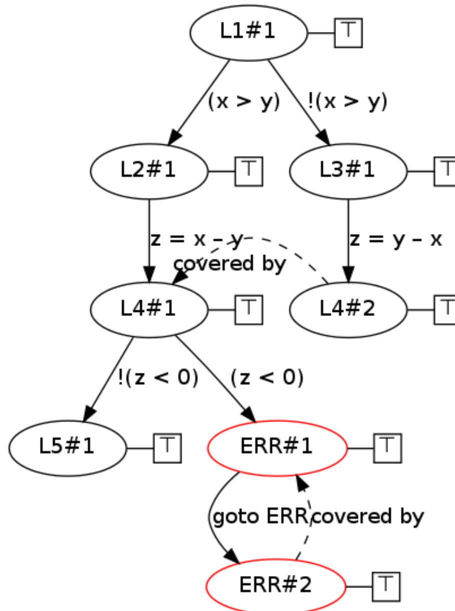


Рис.12. Абстрактное дерево достижимости для пустого набора предикатов $\pi = \emptyset$.

2. Построенное АДД не является безопасным по отношению к конфигурации $(ERR, \top)$, так как в нём присутствует вершина $ERR\#1: (ERR, \top)$.

3. Рассмотрим путь P от начальной вершины АДД $L1\#1$ до вершины $ERR\#1$:

```
L1: (x > y);
```

```
L2: z = x - y;
L4: (z < 0);
```

Рассмотрим этот путь в качестве контрпримера. Вычислим сильнейшее постусловие $SP(P)$:

$$SP(\varphi_1, P) \equiv \top \wedge \psi_1 \wedge \psi_2 \wedge \psi_3 \equiv$$

$$\top \wedge \underbrace{x_1 > y_1}_{\psi_1} \wedge \underbrace{z_2 = x_1 - y_1}_{\psi_2} \wedge \underbrace{z_2 < 0}_{\psi_3} - UNSAT$$

4. Постусловие невыполнимо. Выполним уточнение набора предикатов π на основе интерполянтов Крейга:

$$\rho_1 \equiv x_1 > y_1$$

$$\rho_2 \equiv z_2 > 0$$

Новое множество предикатов $\pi_2' = \{p_1', p_2'\}, p_1' \equiv x > y, p_2' \equiv z > 0$.

5. Строим АДД, соответствующее новому набору предикатов $\pi' = \{x > y, z > 0\}$.

5.1. Обрабатываем переход

$$L1\#1: (L1, \top) \xrightarrow{x>y} L2\#1: (L2, \mathrm{post}_{\pi_2'}(\top, x > y))$$

$$SP(\top, x > y) \equiv x_1 > y_1 - SAT(x_1 = 1, y_1 = 0)$$

$$p_1': \top \wedge x_1 > y_1 \wedge \neg(x_1 > y_1) - UNSAT$$

$$p_2': \top \wedge x_1 > y_1 \wedge \neg(z_1 > 0) - SAT\ (x_1 = 2, y_1 = 1, z_1 = -1)$$

Результат: $L2\#1: (L2, x > y)$.

5.2. Обрабатываем переход

$$L2\#1: (L2, x > y) \xrightarrow{z=x-y} L4\#1: (L4, \mathrm{post}_{\pi_2'}(x > y, z = x - y))$$

$$SP(x > y, z = x - y) \equiv x_1 > y_1 \wedge z_2 = x_1 - y_1 - SAT(x_1 = 1, y_1 = 0, z_2 = 1)$$

$$p_1': x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge \neg(x_1 > y_1) - UNSAT$$

$p_2': x_1 > y_1 \land z_2 = x_1 - y_1 \land \neg(z_2 > 0) - UNSAT$

Результат: $L4\#1: (L4, x > y \land z > 0)$.

5.3. Обрабатываем переход

$$L4\#1: (L4, x > y \land z > 0) \xrightarrow{!(z<0)} L5\#1: (L5, \mathrm{post}_{\pi_2'}(x > y \land z > 0, !(z < 0)))$$

$$SP(x > y \land z > 0, !(z < 0)) \equiv x_1 > y_1 \land z_1 > 0 \land \neg(z_1 < 0) - SAT(x_1 = 1, y_1 = 0, z_2 = 1)$$

$$p_1': x_1 > y_1 \land z_1 > 0 \land \neg(z_1 < 0) \land \neg(x_1 > y_1) - UNSAT$$

$$p_2': x_1 > y_1 \land z_1 > 0 \land \neg(z_1 < 0) \land \neg(z_1 > 0) - UNSAT$$

Результат: $L5\#1: (L5, x > y \land z > 0)$.

5.4. Обрабатываем переход

$$L4\#1: (L4, x > y \land z > 0) \xrightarrow{z<0} ERR\#1: (ERR, \mathrm{post}_{\pi_2'}(x > y \land z > 0, (z < 0)))$$

$$SP(x > y \land z > 0, (z < 0)) \equiv x_1 > y_1 \land z_1 > 0 \land z_1 < 0 - UNSAT$$

Ошибочное состояние недостижимо, результат: $ERR\#1: (ERR, \bot)$.

В результате успешного уточнения абстракции путь, на котором мы в первый раз нашли ошибку, больше не приводит к ошибочному состоянию в АДД.

5.5. Вычисляем переходы

$$L1\#1: (L1, \top) \xrightarrow{!(x>y)} L3\#1(L3, \top),$$

$$L3\#1(L3, \top) \xrightarrow{z=y-x} L4\#2: (L4, \top)$$

$$L4\#2: (L4, \top) \xrightarrow{!(z<0)} L5\#2: (L5, \top)$$

$$L4\#2: (L4, \top) \xrightarrow{(z<0)} ERR\#2: (ERR, \top)$$

$$ERR\#2: (ERR, \top) \xrightarrow{goto\ ERR} ERR\#3: (ERR, \top)$$

Вершина $ERR\#3$ покрыта вершиной $ERR\#2$. Завершаем построение АДД. Результирующее дерево показано на рис. 13.

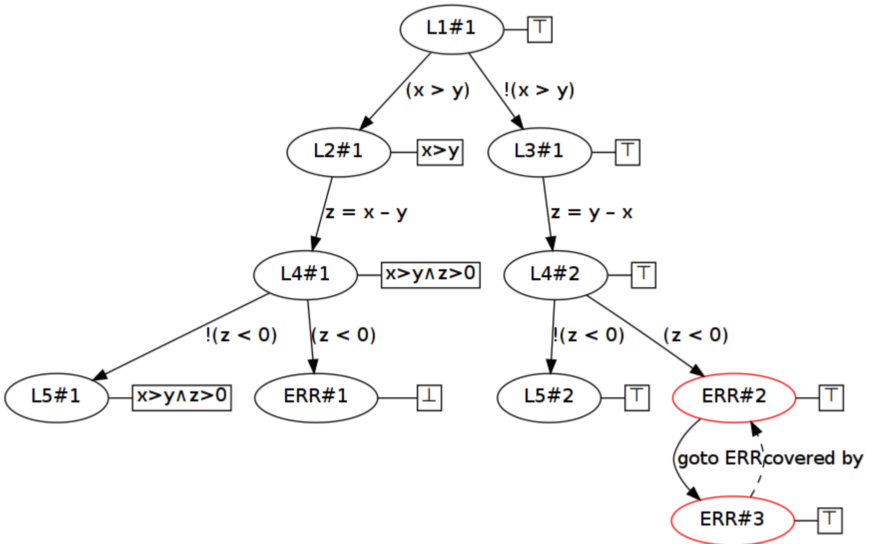


Рис. 13. Абстрактное дерево достижимости для набора предикатов π'_2.

6. Построенное АДД вновь не является безопасным по отношению к конфигурации $(ERR, \top)$, т.к. в нём присутствует вершина $ERR\#2(ERR, \top)$.

7. Соответствующий путь от начальной вершины АДД до вершины $ERR\#2$:

```
L1: !(x > y);
L3: z = y - x;
L4: (z < 0);
```

Построим сильнейшее постусловие пути:

$$SP(P) \equiv \top \wedge \underbrace{\neg(x_1 > y_1)}_{\psi_1} \wedge \underbrace{z_2 = y_1 - x_1 \wedge}_{\psi_2} \underbrace{z_2 < 0}_{\psi_3} - UNSAT$$

8. Постусловие невыполнимо. Выполним уточнение набора предикатов π' на основе интерполянтов Крейга:

$$\rho_1 \equiv x_2 \leq y_2$$

$$\rho_2 \equiv z_2 \geq 0$$

Новое множество предикатов $\pi_2'' = \pi_2' \cup \{p_1'', p_2''\} = \{p_1', p_2', p_1'', p_2''\}$

$$p_1' \equiv x > y$$

$$p_2' \equiv z > 0$$

$$p_1'' \equiv x \leq y$$

$$p_2'' \equiv z \geq 0$$

9. Покажем, как будет построено поддерево АДД, существенно отличающееся от соответствующего поддерева в предыдущем АДД.

9.1. Переход $L1: (L1, \top) \xrightarrow{!(x>y)} L3\#1: (L3, \mathrm{post}_{\pi_2''}(\top, !\,(x > y)))$

$$SP(\top, !\,(x > y)) \equiv \top \wedge \neg(x_1 > y_1) - SAT(x_1 = 0, y_1 = 1)$$

$$p_1'': \top \wedge \neg(x_1 > y_1) \wedge \neg(x_1 \leq y_1) - UNSAT$$

$$p_2'': \top \wedge \neg(x_1 > y_1) \wedge \neg(z_1 \geq 0) - SAT\ (x_1 = 1, y_1 = 2, z_1 = -1)$$

Результат: $L3\#1: (L3, x \leq y)$.

9.2. Переход $L3\#1: (L3, x \leq y) \xrightarrow{z=y-x} L4\#2: (L4, \mathrm{post}_{\pi_2''}(x \leq y, z = y - x))$

$$SP(x \leq y, z = y - x) \equiv x_1 \leq y_1 \wedge z_2 = y_1 - x_1 - SAT(x_1 = 1, y_1 = 0, z_2 = 1)$$

$$p_1'': x_1 \leq y_1 \wedge z_2 = y_1 - x_1 \wedge \neg(x_1 \leq y_1) - UNSAT$$

$$p_2'': x_1 \leq y_1 \wedge z_2 = y_1 - x_1 \wedge \neg(z_2 \geq 0) - UNSAT$$

Результат: $L6\#2: (L6, x \leq y \wedge z \geq 0)$.

9.3. Переход

$$L4\#2: (L4, x \leq y \wedge z \geq 0) \xrightarrow{!(z<0)} L5\#2: (L5, \mathrm{post}_{\pi_2''}(x \leq y \wedge z \geq 0, !\,(z < 0)))$$

$$SP\big(x \le y \land z \ge 0, !\,(z < 0)\big) \equiv x_1 \le y_1 \land z_1 \ge 0 \land \neg(z_1 < 0) - SAT(x_1 = z_1 = 1, y_1 = 0)$$

$$p_1'': x_1 \le y_1 \land z_1 \ge 0 \land \neg(z_1 < 0) \land \neg(x_1 \le y_1) - UNSAT$$

$$p_2'': x_1 \le y_1 \land z_1 \le 0 \land \neg(z_1 < 0) \land \neg(z_1 \ge 0) - UNSAT$$

Результат: $L5\#2: (L5, x \le y \land z \ge 0)$.

9.4. Переход

$$L4\#2: (x \le y \land z \ge 0) \xrightarrow{z<0} ERR\#2: \big(ERR, \mathrm{post}_{\pi_2''}\big(x \le y \land z \ge 0, (z < 0)\big)\big)$$

$$SP(x \le y \land z \ge 0, (z < 0)) = x_1 \le y_1 \land z_1 \ge 0 \land z_1 < 0 - UNSAT$$

Результат: $ERR\#2: (ERR, \bot)$.

Результирующее АДД показано на Рис. 14.

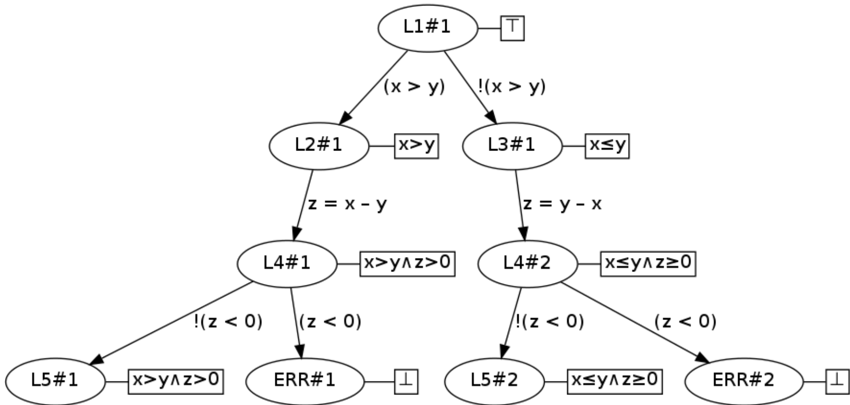


Рис.14. АДД для набора предикатов π_2''.

10. Полученное АДД безопасно по отношению к конфигурации $(ERR, \top)$, т.к. обе вершины $ERR\#1$ и $ERR\#2$, соответствующие ошибочной метке ERR в этом дереве, имеют невыполнимые регионы $\bot$.

Таким образом метод CEGAR в рассмотренном варианте позволил доказать недостижимость ошибочной метки в программе из примера 1 за 3 итерации рассмотренного цикла верификации (шаги 1-4, 5-8, 9-10).

8 Оптимизации и расширения метода CEGAR

Мы рассмотрели применение метода CEGAR для построения и уточнения предикатной абстракции простейшей программы на языке С. В этой программе определена всего одна функция (`main`), используются только глобальные переменные типа `int`. Кроме этого в рассмотренном примере отсутствуют циклы. Рассмотрим возможности ослабления данных ограничений и применяемые при этом оптимизации построения абстракции (АДД) программы.

8.1 Ленивая абстракция

В рассмотренном примере 1 (из предыдущего раздела) можно заметить, что процесс построения предикатной абстракции можно несколько упростить, если

- не перестраивать заново те части абстрактного дерева достижимости, где уточнение абстракции не повлияет на достижимость ошибочной инструкции (недостижимость уже доказана существующими предикатами);
- вычислять значения вновь вводимых при уточнении предикатов только в тех вершинах, для которых эти предикаты были получены, то есть только там, где их значения будут существенно использоваться в доказательстве исключения из АДД невыполнимого ошибочного пути.

В нашем примере это будет означать следующее:

- не пересчитывать значение предикатов $b_1' \equiv x \leq y$ и $b_2' \equiv z \geq 0$ в вершинах $L1\#1, L2\#1, L4\#1, L5\#1$ и $ERR\#1$, т.к. недостижимость ошибочной инструкции из этих вершин перед добавлением предикатов b_1' и b_2' уже доказана предикатами $b_1 \equiv x > y$ и $b_2 \equiv z > 0$. По сути это означает, что при уточнении абстракции можно полностью оставить без изменения всё АДД, кроме поддерева с корнем в вершине $L3\#1$.
- не пересчитывать значение предикатов b_1и b_2в вершинах $L3\#1, L4\#2, L5\#1, L5\#2, ERR\#1$ и $ERR\#2$, потому что они были получены для вершин $L2\#1$ и $L4\#1$, и для доказательства недостижимости ошибочной метки их значения существенно требуются только в этих вершинах.

Абстракция, которая строится с учетом предложенных оптимизаций, имеет таким образом не один фиксированный набор предикатов для всех вершин АДД, а свой набор предикатов для каждой вершины. Предикаты в таких наборах называют *локальными*, а соответствующий метод построения АДД – *ленивой абстракцией (*от англ. *lazy abstraction)*. Именно этот метод построения АДД реализован в инструментах CPAchecker и BLAST (который поэтому называется Berkeley *Lazy Abstraction* Software verification Tool).

8.2 Программы с циклами и крупноблочное кодирование

Для программ, содержащих циклы, метод CEGAR используется аналогично рассмотренному примеру (в силу того, что программа с циклами является простейшей, а метод CEGAR был рассмотрен для любой простейшей программы). Как было отмечено, конечность абстрактного дерева достижимости обеспечивается конечностью множества абстрактных состояний программы и применением операции покрытия вложенных состояний.

В примере 1 при построении АДД по заранее заданному набору предикатов (в разделе 5) выполнялось покрытие вершины $L4\#2\colon(L4,\top)$ другой ранее построенной вершиной $L4\#1\colon(L4,\top)$ с таким же абстрактным состоянием. В этом случае вложенность состояний была установлена тривиально. В общем случае для предикатной абстракции покрытие проверяется как импликация, т.е. состояние φ_1 покрывает состояние φ_2, если $\varphi_2 \rightarrow \varphi_1$. В случае с циклом, как правило, какая-либо вершина, соответствующая предыдущей итерации цикла будет покрывать вершину для того же места в программе на следующей его итерации, если в обеих вершинах будет выполнен один и тот же набор предикатов. Этот набор предикатов может, в частности, содержать инвариант цикла[3].

Рассмотрим пример простейшей программы, представленной на рис. 15. На рис. 16 для этой программы показан абстрактный граф достижимости, построенный для набора предикатов $\pi = \{b_1 \equiv z \geq 0, b_2 \equiv x > y, b_3 \equiv x \leq y\}$.

[3]Инвариантом в программировании называется логическое выражение, истинное после каждого прохода тела цикла и перед началом выполнения цикла и зависящее от переменных, изменяющихся в теле цикла.

```
void main() {
     int x = nondet;
     int y = nondet;
     int z = nondet;
L1:  z = 0;
LOOP:if(x > 0) {
L2:       if (y > 0) {
L3:            if (x > y){
L4:                 z = x - y;
L5:                 x = z;
               } else {
L6:                 z = y - x;
L7:                 y = z;
               }
L8:            goto LOOP;
          }
     }
L9:  if (z < 0)
ERR:      goto ERR;
L10:}
```

Рис. 15. Пример простейшей программы 2

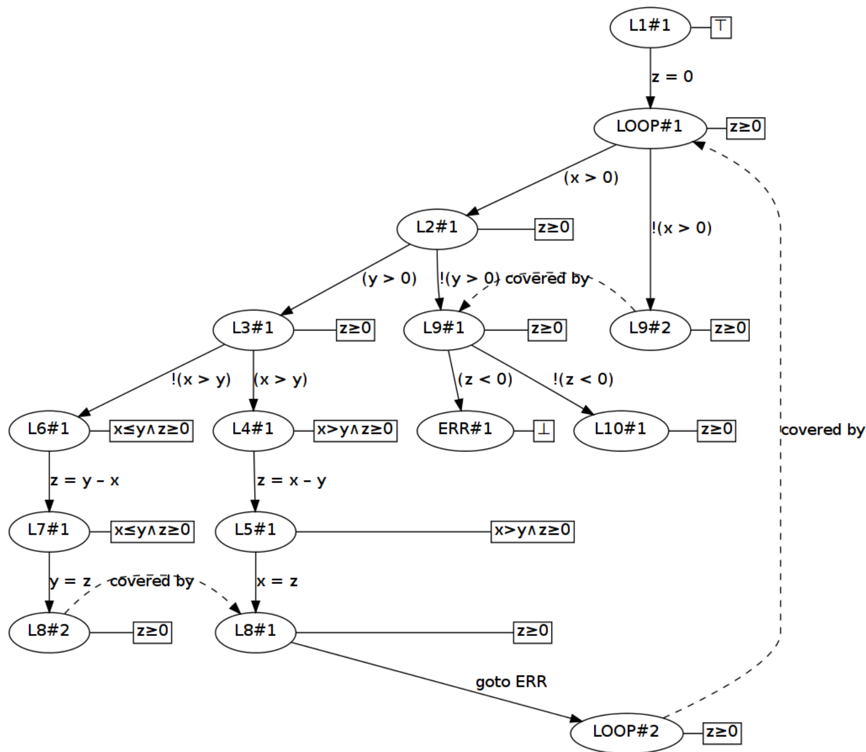


Рис. 16. Абстрактный граф достижимости для простейшей программы из примера 2

На этом примере видно, что по существу для обеспечения конечности абстрактного дерева достижимости и доказательства недостижимости ошибочной метки важны лишь состояния абстракции в вершинах, помеченных абстрактными состояниями$(LOOP, z \geq 0)$ и $(ERR, \perp)$. Также видно, что предикат $b_1 \equiv z \geq 0$ наиболее важен для доказательства недостижимости. Остальные вершины представляют по сути некоторые промежуточные состояния абстракции, а предикаты b_2 и b_3служат для представления некоторой промежуточной информации. Эти рассуждения наталкивают на мысль об оптимизации процесса построения предикатной абстракции за счет склеивания промежуточных вершин.

В инструменте CPAchecker при построении предикатной абстракции применяется крупноблочное кодирование, или LBE (кодирование большими блоками, Large Block Encoding) [42]. Его суть заключается в склеивании вершин на линейных участках графа достижимости и кодировании сильнейшего постусловия сразу для нескольких соответствующих последовательных переходов. В примере 2 можно склеить, к примеру,

вершины $L4\#1: (L4, x > y \wedge z \geq 0)$, $L5\#1: (L5, x > y \wedge z \geq 0)$ и $L8\#1(L8, z \geq 0)$. В таком случае сильнейшее постусловие $SP(z \geq 0, (x > y); z = x - y;\ x = z) = z_1 \geq 0 \wedge x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge x_2 = z_2$.

$$b_1: z_1 \geq 0 \wedge x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge x_2 = z_2 \wedge \neg(z_2 \geq 0) - UNSAT$$

В результате получим дугу $L3\#1: (L3, z \geq 0) \xrightarrow{x>y, z=x-y, x=z} L4 - L8\#1: (L8, z \geq 0)$. Метка $L4 - L8\#1$ дана новой вершине, полученной в результате склеивания. Аналогично можно склеить вершины $L6\#1: (L6, x \leq y \wedge z \geq 0)$, $L7\#1: (L7, x \leq y \wedge z \geq 0)$ и $L8\#2: (L8, z \geq 0)$. Пометим новую вершину, полученную в результате этой склейки меткой $L6 - L8\#1$. После этого предикаты $b_2 \equiv x \leq y$ и $b_3 \equiv x > y$ окажутся ненужными для доказательства недостижимости в данном примере. Последовательности инструкций $x > y$, $z = x - y$, $x = z$ и $!(x > y)$, $z = y - x$, $y = z$ называют блоками кодирования. Полученный граф достижимости показан на рис. 17.

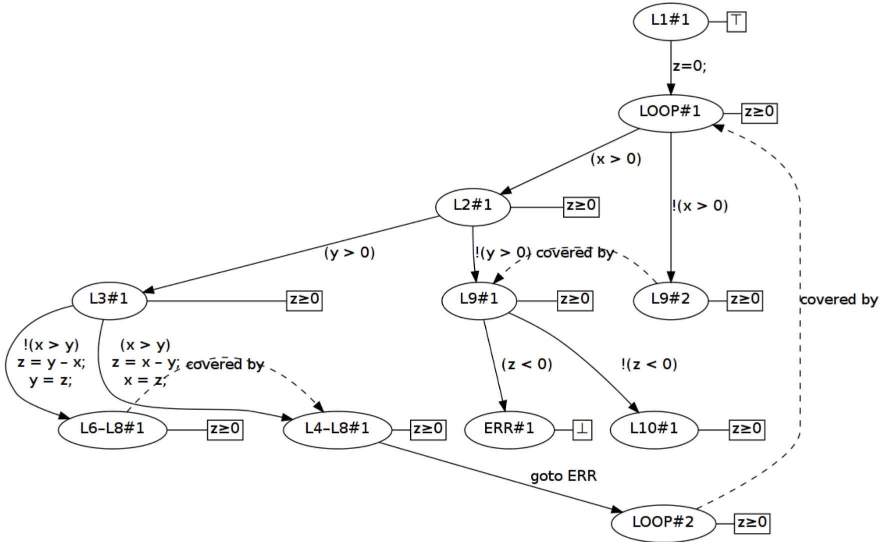


Рис. 17. Граф достижимости после склейки вершин на линейных участках

Склеивать можно не только вершины на линейных участках графа достижимости, но и вершины, соответствующие одному и тому же месту в программе, в котором происходит слияние потока управления. В нашем примере такими вершинами являются вершины $L6 - L8\#1$ и $L4 - L8\#1$. Склеивание таких вершин называется *слиянием* (от англ. *merge*). При слиянии необходимо также осуществить объединение двух альтернативных блоков кодирования, в нашем случае между вершинами $L3\#1$ и $L4 - L8\#1$, а также между $L3\#1$ и $L6 - L8\#1$. Для этого можно использовать дизъюнкцию. Пометим новую вершину $L4 - L8||L6 - L8\#1$ и посчитаем абстракцию в ней через сильнейшее постусловие:

$$SP(z \geq 0, x > y, z = x - y, x = z \;||!\,(x > y), z = y - x, y = z) =$$

$$(z_1 \geq 0 \land x_1 > y_1 \land z_2 = x_1 - y_1 \land x_2 = z_2) \lor (z_1 \geq 0 \land \neg(x_1 > y_1) \land z_2 = y_1 - x_1 \land y_2 = z_2)$$

$$b_1: ((z_1 \geq 0 \land x_1 > y_1 \land z_2 = x_1 - y_1 \land x_2 = z_2) \lor (z_1 \geq 0 \land \neg(x_1 > y_1) \land z_2 = y_1 - x_1 \land y_2 = z_2)) \land$$

$$\land \neg(z_2 \geq 0) - UNSAT$$

В результате получаем дугу $L3\#1: (L3, z \geq 0) \xrightarrow{x>y, z=x-y, x=z \;||!(x>y), z=y-x, y=z} L4 - L8||L6 - L8\#1(L8, z \geq 0)$.

Применяя последовательно аналогичные операции склейки и слияния вершин, получим абстрактный граф достижимости для крупноблочного кодирования с пересчётом состояния абстракции только в заголовке цикла и на ошибочной метке, показанный на рисунке 10 (для получения этого графа необходимо также предварительно дублировать вершину $L2\#1$).

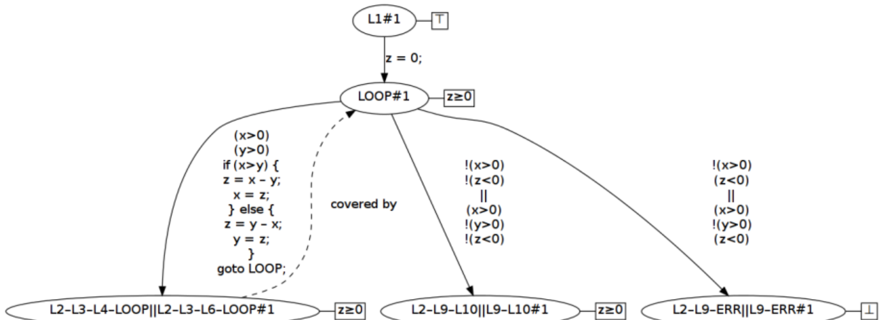


Рис.10. Граф достижимости для крупноблочного кодирования со слияниями

В инструменте CPAchecker размер блока кодирования является настраиваемым, как вручную, так и динамически во время построения абстракции. Подробнее об этом написано в статье [43]. Построение абстракции при крупноблочном кодировании в CPAchecker осуществляется аналогично рассмотренному ранее, при помощи вычисления абстрактных постусловий и решения задач о выполнимости формул. CPAchecker сразу строит абстракцию с заданным размером блока, не делая никаких последовательных преобразований АДД, то есть осуществляет склейку и слияние вершин «на лету». Размер блока кодирования в этом инструменте можно ограничивать максимальным количеством объединяемых в один переход инструкций или условием пересчёта абстракции (например, только на заголовке цикла или на заголовке цикла и операторах ветвления).

8.3 Поддержка указателей

В инструменте BLAST помимо типа данных `int` реализована поддержка некоторых выражений с указателями на основе анализа алиасов. В программировании словом *алиасинг* (от англ. *alias* – имя, прозвище) описывается ситуация, при которой какая-либо ячейка с данными в памяти программы оказывается доступна в исходном тексте (коде) этой программы под различными обозначениями (именами). Таким образом, изменение данных с использованием одного из таких обозначений неявно ведет к изменению значений, доступных по всем остальным обозначениям той же ячейки памяти. В результате алиасинг значительно осложняет как анализ и оптимизацию, так и верификацию соответствующих программ. Анализ алиасов предназначен для извлечения из кода программы полезной информации о присутствующем в ней алиасинге.

Например, пусть в программе имеются объявления:

```
int a;
int *p;
```

Тогда в точке программы, где выполнено условие $p == \&a$, разыменование $*\,p$ будет являться алиасом переменной a.

Другой пример: $*\,(q+1)$ может являться алиасом для $b[1]$, если $q == b$.

Если в результате анализа алиасов оказалось, что для некоторого выражения в программе найденное множество возможных значений указателя включает лишь один вариант (иными словами данное выражение всегда адресует один и тот же объект), то говорят, что такое выражение является *обязательным алиасом* (от англ. *must-alias*) этого объекта. В других случаях говорят о *возможных алиасах* (от англ. *may-aliases*).

Для методов статической верификации, которые сами по себе анализируют различные пути выполнения программы, бывает достаточно не чувствительного к потоку управления метода анализа возможных алиасов. В инструменте BLAST информация об алиасах используется при генерации ограничений, которые на состояние памяти программы накладывает операция присваивания. При анализе контрпримера или вычислении абстрактного постусловия для каждого возможного алиаса цели присваивания генерируется проверка, является ли его адрес равным адресу цели. В зависимости от результата этой проверки с помощью логических формул выражается либо обновление соответствующего выражения-алиаса, либо сохранение прежнего значения этого выражения. Выборка возможных алиасов для каждой цели присваивания, таким образом, производится на основе результатов работы нечувствительного к потоку анализа возможных алиасов. В качестве нечувствительного к потоку управления алгоритма анализа алиасов BLAST используется алгоритм на основе BDD, описанный в [44].

Пусть, например, в некоторой программе переменные a, pa и pb имеют тип `int`, `int *` и `int *` соответственно. Анализ алиасов показал, что выражение $*pb$ является возможным алиасом выражения $*pa$ и в момент генерации формулы перехода для операции присваивания выражениям $pa,\ pb,\ a,\ *pa,\ *pb$ соответствовали следующие индексы в SSA-представлении: $\{pa \to i, pb \to j, a \to k, *pa \to l, *pb \to m\}$. Тогда для операции присваивания `*pa = a` BLAST сгенерирует формулу перехода:

$$*pa_{l+1} = a_k \land ((pb_j = pa_i \land *pb_{m+1} = a_k) \lor (\neg(pb_j = pa_i) \land *pb_{m+1} = *pb_m))$$

Такой подход к использованию анализа возможных алиасов для генерации ограничений в формулах пути был расширен в [41] на анализ алиасов с участием структурных типов. В статье приводится алгоритм генерации ограничений, соответствующих присваиваниям и условиям в ветвлениях рассматриваемой программы, с учетом наличия структурных типов. В статье вводятся ограничения на точность анализа в случае рекурсивных структур (к примеру, связных списков) и сильно вложенных указателей на структуры (переменные, генерируемые при анализе, не могут отличаться от встречаемых в программе более, чем на фиксированное количество разыменований). Для целей присваивания обновляются не только непосредственно выражение в левой части и его алиасы (в случае равенства их адресов), но и их поля, если это структурные типы, и поля тех структур, на которые они указывают, если это указатели на структуру. Обновляемые таким образом значения генерируются рекурсивно не более чем на заданную глубину, которая измеряется количеством разыменований.

8.4 Анализ реальных программ на C

Реальные программы, в отличие от простейших

- состоят из большого числа функций, принимающих параметры и возвращающих значения;
- используют глобальные и локальные переменные различных типов,
- размещают составные объекты в статической и динамической памяти, в том числе структуры, массивы и объединения;
- могут содержать все конструкции (операторы и выражения) языка C, в том числе нелинейные и побитовые операции.

Кроме этого реальные инструменты статической верификации (в том числе BLAST и CPAchecker) позволяют использовать недетерминированные значения в любом месте программы, где допустим вызов функции без параметров.

Поэтому при верификации реальных программ инструменты используют ряд приёмов и упрощающих предположений.

BLAST и CPAchecker перед верификацией применяют к данной на вход программе некоторые упрощающие преобразования, которые, в частности, позволяют свести все виды циклов, короткую логику и тернарный оператор к использованию только операторов `if` и `goto`. В BLAST эти преобразования выполняются с использованием инструмента CIL [45].

Поддержка вызовов функций в BLAST и CPAchecker реализована с помощью встраивания тела функции в месте её вызова. При этом каждому формальному параметру функции ставится в соответствие переменная, которой присваивается значение соответствующего фактического параметра в месте вызова функции. Возвращаемое значение функции присваивается временной переменной непосредственно перед выходом из функции. При такой реализации BLAST и CPAchecker в случае рекурсивных функций поддерживают только небольшую конечную глубину рекурсивных вызовов.

Некоторые инструменты, использующие метод CEGAR, например, Yogi [46] реализуют возможности так называемого *обобщения* (от англ. *summarization*) вызовов функций. Такие инструменты в месте вызова функции сначала пытаются переиспользовать некоторую информацию о ней, собранную во время анализа тела функции. Например, Yogi анализирует вызовы функций в три этапа.

- Сначала он пытается использовать некоторое приближение множества переменных, значения которых могут изменяться в теле вызываемой функции. Информацию об этих переменных Yogi получает, в частности, из результатов анализа возможных алиасов. Yogi использует полученное приближение в предположении, что все переменные, возможно изменяемые вызываемой функцией, принимают произвольные значения (`nondet`).

- Если первое обобщение оказывается неудачным, что происходит в результате построения ложного контрпримера (проверку контрпримера Yogi выполняет с помощью интерпретации программы), инструмент пытается использовать следующее обобщение. Оно состоит из двух пар регионов – (φ_1, ψ_1) и (φ_2, ψ_2). Первая пара представляет собой *обобщение достижимости* (от англ. *must-summary*). Его смысл в том, что для каждого состояния, заданного φ_1 (для любого места вызова функции) существует выполнение вызываемой функции, приводящее в какое-либо состояние, заданное ψ_1. Вторая пара – это *обобщение недостижимости* (от англ. *not-may summary*), оно означает, что не существует выполнений вызываемой функции, приводящих из какого-либо состояния, заданного φ_2 в какое-либо состояние, заданное ψ_2. Предположим, что регион абстрактного состояния программы непосредственно перед вызовом функции равен α, а непосредственно после возврата из неё – β. Тогда если выполнены условия $\varphi_1 \to \alpha$ и $\psi_1 \cap \beta \neq \emptyset$, то это означает, что можно попытаться построить уточненный контрпример, включающий вызов функции, в два этапа. Сначала построить такой контрпример, чтобы перед вызовом функции был выполнен предикат φ_1, а после её вызова – предикат ψ_1. Затем построить контрпример выполнения тела функции, который существует по определению пары регионов (φ_1, ψ_1). Если же выполнены условия $\alpha \to \varphi_2$ и $\beta \to \psi_2$, то можно исключить построенный ложный контрпример, уточнив регион α соответствующего абстрактного состояния с помощью конъюнкции с регионом φ_2 (конъюнкция соответствует пересечению абстрактных состояний данных). По определению пары (φ_2, ψ_2) при условии $\beta \to \psi_2$ все состояния, заданные β окажутся недостижимыми из уточненного абстрактного состояния. Использование обобщений, таким образом, позволяет переиспользовать информацию о достижимости, полученную ранее при анализе тела функции, оптимизируя построение контрпримера и позволяя уточнять абстракцию без вызова интерполирующего SMT-решателя.
- Если использовать второе обобщение не удается, Yogi подставляет и анализирует тело вызываемой функции в месте вызова. При этом могут уточняться имеющиеся для этой функции пары регионов (φ_1, ψ_1) и (φ_2, ψ_2).

Для поддержки переменных различных типов в инструментах верификации используются два основных подхода – приближение значений всех переменных математическими целыми и вещественными числами, независимо от их типа, и точное кодирование значений переменных и операций над ними с помощью битовых векторов различного фиксированного размера. BLAST использует первый подход. В CPAchecker были реализованы оба подхода.

Решение и интерполяция логических формул в теории битовых векторов поддерживается некоторыми SMT-решателями, например, MathSAT 5 [47].

Использование битовых векторов позволяет также точно кодировать большинство нелинейных и побитовых операций над переменными различных типов. При использовании целых и вещественных чисел в BLAST и CPAchecker нелинейные и побитовые операции кодируются неинтерпретируемыми функциями, что снижает точность анализа.

Поддержка структур и указателей в BLAST реализована на основе анализа алиасов. CPAchecker поддерживает представление структур в виде битовых векторов с возможностью обращения к полям структуры, как к выделенным битовым подвекторам. Поддержка массивов в BLAST и CPAchecker на момент написания статьи не была реализована (используется неточная эвристика на основе неинтерпретируемых функций).

9 Использование решателей

Как было отмечено в части, описывающей процесс построения абстракции программы, для автоматического решения задачи о выполнимости логической формулы на практике используются специальные инструменты, называемые решателями.

В широком смысле *решающие процедуры* (от англ. *decision procedures*) или решатели, (от англ. *solvers*) – это алгоритмы, которые приняв на вход проблему разрешимости, то есть вопрос, сформулированный в рамках какой-либо формальной системы (аксиоматической теории), и требующий ответа «да» или «нет»; и выдают на выходе соответствующий корректный результат. В узком смысле в рамках данной статьи нас интересуют решатели для алгоритмически разрешимых задач, сформулированных в рамках теорий первого порядка, и используемых на практике в областях верификации, доказательства корректности, оптимизации и др. Возможности многих технологий в этих областях ограничены возможностями используемых решателей. Поэтому последние остаются объектом многих активных исследований во всём мире, как в академической среде, так и в промышленности.

Решающие процедуры можно разделять по поддерживаемым ими теориям. Решатели для логики высказываний, т.е. классической логики нулевого порядка, которые по сути являются инструментами для решения задачи выполнимости булевых формул, называются SAT-решателями. Решатели для формул классической логики первого порядка с равенством, заданных в рамках комбинации некоторых аксиоматических теорий, называются SMT-решателями. Такие решатели могут поддерживать только формулы нулевого порядка (без кванторов). Среди наиболее часто используемых на практике теорий – вещественная и целочисленная линейная арифметика, неинтерпретируемые функции, массивы и битовые векторы.

Практически все современные решатели для пропозициональной (логической) части формулы используют либо схему DPLL (Davis-Putnam-Logemann-Loveland), основанную на поиске с возвратом, для решения задачи в виде КНФ, либо основаны на *суперпозиционном исчислении* (от англ. *superposition calculus*), расширении резолютивного вывода [48]. Алгоритм DPLL описан, например, в книге [49]. Для взаимодействия с решающими процедурами и для подформул, заданных в рамках какой-либо теории, используются в основном техники комбинирования Нельсона-Оппена (Nelson-Oppen Combination Procedure), пропозиционального кодирования (от англ. propositional encodings) и отложенного комбинирования теорий (DTC, Delayed Theory Combination). Техники комбинирования решателей Нельсона-Оппена и пропозиционального кодирования приводится, например, в книге [49]. Отложенное комбинирование теорий предложено в статье [50] и используется в решателе MathSAT.

Также некоторые решатели могут в результате своей работы помимо основного ответа «выполнимо»/«невыполнимо», давать также некоторую дополнительную информацию, например, для выполнимой формулы – её модель, для невыполнимой – опровержение (например, в виде дерева вывода тождественной лжи), набор дизъюнктов, использованных для доказательства невыполнимости (англ. unsatisfiable core), для невыполнимой конъюнкции пары формул – интерполянт Крейга (Craig interpolant) [40].

Решатели, позволяющие находить интерполянты Крейга, называются интерполирующими процедурами, или интерполирующими решателями, или просто интерполяторами. В существующих инструментах интерполянты строятся из дерева вывода для доказательства невыполнимости. Некоторые методы построения интерполянтов из дерева вывода для комбинации определённого класса теорий описаны в статье [51]. Они требуют предварительного построения частичных интерполянтов для используемых теорий. Методы их построения для вещественной линейной арифметики предлагаются в статье [52], а для теории неинтерпретируемых функций – в статье [53]. На данных методах основывается интерполирующий решатель CSIsat [54] для логических формул без кванторов в рамках теорий вещественной линейной арифметики и неинтерпретируемых функций. Этот интерполирующий решатель используется инструментом верификации BLAST[4]. Другой интерполирующий решатель для формул без кванторов, MathSAT, поддерживающий также целочисленную линейную арифметику, использует свой метод построения и комбинирования частичных интерполянтов, описанный в статье [55] и диссертации [56]. MathSAT используется как основной интерполирующий решатель в инструменте CPAchecker [34]. SLAM2 использует для проверки выполнимости пути и извлечения предикатов SMT-решатель Z3 [57], о чём говорится, например, в статье [58].

Таким образом, практическая эффективность работы современных инструментов верификации (SLAM2, BLAST, CPAchecker) во многом

определяется эффективностью существующих на данный момент решающих процедур, особенно SMT-решателей с поддержкой теорий вещественной и целочисленной линейной арифметики и неинтерпретируемых функций, а также соответствующих интерполяторов.

10 Заключение

Мы подробно рассмотрели метод CEGAR для декартовой предикатной абстракции, применив его для доказательства недостижимости ошибочной метки в коде примера простейшей программы на языке C. На рассмотренном примере мы показали такие этапы работы инструментов статической верификации BLAST и CPAchecker, основанных на CEGAR, как построение графа потока управления программы, обход этого графа, построение абстрактного дерева достижимости, в том числе вычисление сильнейших постусловий и регионов абстрактных состояний, а также процесс уточнения абстракции по ложному контрпримеру с использованием интерполяции Крейга. В конце мы также рассказали о некоторых наиболее распространённых модификациях и оптимизациях, используемых при реализации подхода CEGAR в современных инструментах.

Инструменты, основанные на методе CEGAR, занимают первые места в международных соревнованиях по верификации Си-программ [59], [60]. Их текущее состояние позволяет успешно решать задачи верификации моделей аппаратных систем [61], драйверов устройств в операционных системах Windows [31, 58] и Linux [18, 35, 36, 62, 63, 64], а также систем, образующих так называемые product lines [65, 66].

Тем не менее, в реальном коде встречается немало сложностей, которые вызывают проблемы даже у самых современных инструментов, реализующих подход CEGAR. Открытые направления для развития включают в себя:

- поддержку нелинейной арифметики, возможно, с помощью комбинирования предикатной абстракции с другими видами анализа [67, 68];
- поддержку битовой арифметики, побитовых операций, учета возможных арифметических переполнений и преобразования типов;
- поддержку преобразования типов и объединений (union);
- анализ указателей, в том числе поддержку алиасинга, массивов, адресной арифметики, конструкций вида `container_of` [69] и др.;
- анализ сложных типов данных, таких как ссылочные структуры, хэш-таблицы и др.;
- анализ многопоточных программ.

Литература

[1]. Dershowitz N. Software horror stories. URL: http://www.cs.tau.ac.il/~nachumd/horror.html
[2]. Turing A. M. On Computable numbers, with an application to the Entscheidungsproblem // Proceedings of the London Mathematical Society. 1936. pp. 230—265.
[3]. Floyd R. Assigning Meanings to Programs // Mathematical Aspects of Computer Science. 1967. pp. 19—32.
[4]. Hoare C. An Axiomatic Basis for Computer Programming // Communications of the ACM. 1969. vol. 12. pp. 576—580.
[5]. Dijkstra E. A Discipline of Programming // Prentice-Hall, 1976.
[6]. Millo R. D., Lipton R., Perlis A. Social Processes and Proofs of Theorems and Programs // Communications of the ACM. 1979. vol. 22. pp.271—280.
[7]. Nelson G. Techniques for Program Verification // Tech. Rep. CSL81-10: Xerox Palo Alto Research Center, 1981.
[8]. Nelson G., Oppen D. Fast Decision Procedures Based on Congruence Closure // Journal of the ACM. 1980. vol. 27. pp. 356—364.
[9]. Shostak R. Deciding Combinations of Theories // Journal of the ACM. 1984. vol. 31. pp. 1—12.
[10]. Clarke E. M., Emerson E. A. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic // Logic of Programs. 1981. vol. 131. pp. 52—71.
[11]. Queille J., Sifakis J. Specification and Verification of Concurrent Systems in Cesar // Fifth International Symposium on Programming / Ed. by M. Dezani-Ciancaglini, U. Montanari. Lecture Notes in Computer Science. Springer-Verlag, 1981. pp. 337—351.
[12]. Vardi M., Wolper P. Reasoning about Infinite Computations // Information and Computation. 1994. vol. 115. pp. 1—37.
[13]. Pnueli A. The Temporal Logic of Programs // Proceedings of the 18th Annual Symposium on Foundations of Computer Science. IEEE Computer Society Press, 1977. pp. 46—57.
[14]. Emerson E. Temporal and Modal Logic // Handbook of Theoretical Computer Science / Ed. by J. van Leeuwen. Elsevier Science Publishers, 1990. vol. B. pp. 995—1072.
[15]. Khedker Uday P., Sanyal Amitabha, Karkare Bageshri. Data Flow Analysis: Theory and Practice // CRC Press (Taylor and Francis Group), 2009.
[16]. D'Silva V., Kroening D., Weissenbacher G. A Survey of Automated Techniques for Formal Software Verification // Computer-Aided Design of Integrated Circuits and Systems. 2008. vol. 27, no. 7. pp. 1165—1178. On IEEE Transactions.
[17]. Jhala R., Majumdar R. Software model cheking // ACM Computing Surveys. 2009.
[18]. Д. Бейер, А.К. Петренко. Верификация драйверов операционной системы Linux // Труды Института системного программирования РАН, том 23, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), 2012 г. стр. 405-412.
[19]. Vladimir Nesov. Automatically Finding Bugs in Open Source Programs // Third International Workshop on Foundations and Techniques for Open Source Software Certification. OpenCert 2009 vol. 20 2009. pp.19—29.
[20]. Dawson Engler, Benjamin Chelf, Andy Chou. Checking system rules using system-specific, programmer-written compiler extensions // Proceedings of the 4th conference

on Symposium on Operating System Design & Implementation vol. 4 OSDI'00. 2000. pp. 1—16.
[21]. Julia L. Lawall, Julien Brunel, Nicolas Palix, Rene Rydhof Hansen, Henrik Stuart, Gilles Muller. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code // DSN'09 – The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2009. pp. 43—52.
[22]. Арутюн Аветисян, Алексей Бородин. Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок // Труды Института системного программирования РАН, том 21, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). 2011. стр. 39-54.
[23]. В.Н. Игнатьев. Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования // Труды Института системного программирования РАН, том 22, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). 2012. стр. 169-188.
[24]. Biere A., Cimatti A, Clarke E., Strichman O., Zhu Y. Bounded model checking // Advances in Computers. 2003.
[25]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking // Journal of the ACM. 2003.
[26]. Thomas Donald, Moorby Phillip. The Verilog Hardware Description Language // Norwell, MA.: Kluwer Academic Publishers.
[27]. Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs // Tools and Algorithms for the Construction and Analysis of Systems. 2004.
[28]. Ivancic F., Yang Z., Ganai M.K., Gupta A., I. Shlyakhter, Ashar P. F-soft: Software verification platform // Computer Aided Verification. vol. 3576 of Lecture Notes in Computer Science. Springer, 2005. pp. 301—306.
[29]. Post H., Sinz C., Merz F., Gorges T., Kropf T. Linking functional requirements and software verification // 17th IEEE International Requirements Engineering Conferene. 2009. pp. 295—302.
[30]. Donaldson A. F., Kroening D., Ruemmer P. Automatic analysis of DMA races using model checking and k-induction // Formal Methods in System Design. 2011. vol. 39. pp. 83—113.
[31]. Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J. The Static Driver Verifier Research Platform // Computer Aided Verification. 2010.
[32]. D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar. The software model checker BLAST: Applications to software engineering // Int. J. Softw. Tools Technol. Transf., 2007. vol. 9, № 5, ISSN 1433-2779. Springer-Verlag, Berlin, Heidelberg. pp. 505—525.
[33]. E. Clarke, D. Kroening, N. Sharygina, K. Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C // Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), 2005. Lecture Notes in Computer Science, Springer Verlag. vol. 3440. ISBN 3-540-25333-5. pp.570—574.
[34]. D. Beyer, M. E. Keremoglu. CPAchecker: a tool for configurable software verification // Proceedings of the 23rd international conference on Computer aided verification. 2011. ISBN 978-3-642-22109-5. Springer-Verlag. pp. 184—190.
[35]. В.С. Мутилин, Е.М. Новиков, А.В. Страх, А.В. Хорошилов, П.Е. Швед. Архитектура Linux Driver Verification. // Труды Института системного программирования РАН, том 20, стр. 163—187, 2011.
[36]. A. Khoroshilov, V. Mutilin, E. Novikov, P. Shved, A. Strakh. Towards an Open Framework for C Verification Tools Benchmarking // Proceedings of the Eighth

International Andrei Ershov Memorial Conference “Perspectives of Systems Informatics” (PSI 2011), pp. 82—91, 2011.
[37]. S. Graf, H. Saïdi. Construction of Abstract State Graphs with PVS // Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel. pp. 72—83.
[38]. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G. Lazy abstraction // Proc. POPL. ACM, New York. pp. 58—70, 2002.
[39]. Cytron, R., Ferrante, J., Rosen, B.K.,Wegman,M.N., Zadek, F.K. Efficiently computing static single-assignment form and the program dependence graph // ACM Trans. Program. Languages Systems 13(4), pp. 451—490. 1991.
[40]. W. Craig. Linear reasoning // J. Symbolic Logic. 1957. vol. 22, pp. 250—268.
[41]. T. A. Henzinger, K. L. McMillan, R. Jhala, R. Majumdar. Abstractions from Proofs. // POPL 2004.
[42]. D. Beyer, A. Cimatti, A. Griggio, M.E. Keremoglu, R. Sebastiani. Software Model Checking via Large-Block Encoding // Proc. FMCAD, pp. 25—32. IEEE, 2009.
[43]. D. Beyer, M. E. Keremoglu, P. Wendler. Predicate abstraction with adjustable-block encoding // Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. Lugano, Switzerland. 2010. pp. 189—198.
[44]. M. Berndl, O. Lhotak, F. Qian, L. Hendren, N. Umanee. Points-to analysis using BDDs // Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03). ISBN 1-58113-662-5. San Diego, California, USA. pp.103—114.
[45]. George C. Necula, Scott McPeak, S. P. Rahul, Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. // in Proc. of Conference on Compiler Construction, 2002, pp. 213—228.
[46]. Aditya V. Nori and Sriram K. Rajamani. An Empirical Study of Optimizations in Yogi // ICSE '10: International Conference on Software Engineering, May 2010.
[47]. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, Roberto Sebastiani. The MathSAT5 SMT Solver. // Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, volume 7795, 2013, pp. 93—107.
[48]. Robert Nieuwenhuis, Alberto Rubio. Paramodulation-Based Theorem Proving // Handbook of Automated Reasoning I (7), Elsevier Science and MIT Press, 2001.
[49]. D. Kroening, O. Strichman. Decision Procedures. An Algorithmic Point of View // Springer, 2008.
[50]. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, R.Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination // CAV'05.
[51]. G. Yorsh, M. Musuvathi. A Combination Method for Generating Interpolants // CADE 2005.
[52]. A.Rybalchenko, V.Sofronie-Stokkermans. Constraint Solving for Interpolation // AVACS TR No.56, 2009.
[53]. K. L. McMillan. An Interpolating Theorem Prover // TCS'05.
[54]. D.Beyer, D.Zufferey, R.Majumdar. CSIsat: Interpolation for LA+EUF. Tool paper. 2008.
[55]. A.Cimatti, A.Griggio, R.Sebastiani. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories // ACM Transactions on Computational Logic. vol. 12, issue 1, October 2010.
[56]. Thi Thieu Hoa Le. A Novel Technique for Computing Craig Interpolants in Satisfiabilility modulo the Theory of Integer Linear Arithmetic. PhD thesis.

[57]. Leonardo De Moura, Nikolaj Bjørner. Z3: an efficient SMT solver // TACAS'08/ETAPS'08. Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems. pp. 337—340.
[58]. T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static driver verification with under 4% false alarms // Formal Methods in Computer-Aided Design (FMCAD), oct. 2010. pp. 35—42.
[59]. D. Beyer. Competition on Software Verification // C. Flanagan, B. König eds. Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 7214. ISBN 978-3-642-28755-8. Springer Berlin Heidelberg, 2012. pp. 504—524.
[60]. D. Beyer. Second Competition on Software Verification // N. Piterman, S. A. Smolka eds. Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 7795. ISBN 978-3-642-36741-0. Springer Berlin Heidelberg, 2013. pp. 594-609.
[61]. A. Cimatti, A. Micheli, I. Narasamdya, M. Roveri. Verifying SystemC: A Software Model Checking Approach. // in Proc. FMCAD, FMCAD Inc. 2010. pp. 51–59.
[62]. A. Galloway, G. Lüttgen, J. T. Mühlberg, R. I. Siminiceanu. Model-Checking the Linux Virtual File System // N. D. Jones, M. Müller-Olm (eds.). VMCAI 2009. LNCS, vol. 5403, pp. 74—88. Springer, Heidelberg. 2009.
[63]. J. T. Mühlberg, G. Lüttgen. Blasting Linux Code // L. Brim, B. R. Haverkort, M. Leucker, J. van de Pol (eds.). FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 211—226. Springer, Heidelberg (2007)
[64]. W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, F. Piessens. Sound Formal Verification of Linux’s USB BP Keyboard Driver // A. E. Goodloe, S. Person (eds.). NFM 2012. LNCS, vol. 7226, pp. 210—215. Springer, Heidelberg. 2012.
[65]. S. Apel, H. Speidel, P. Wendler, A. von Rhein, D. Beyer. Detection of feature interactions using feature-aware verification // 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2011, pp.372—375. 6-10 Nov. 2011.
[66]. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay. Symbolic model checking of software product lines // in Proceedings of the International Conference on Software Engineering (ICSE). ACM, 2011, pp. 321—330.
[67]. D. Beyer, T. A. Henzinger, G. Theoduloz. Program Analysis with Dynamic Precision Adjustment // Proc. ASE, pp. 29—38. IEEE (2008)
[68]. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival. Combination of Abstractions in the ASTREE Static Analyzer // M. Okada, I. Satoh (eds.). ASIAN 2006. LNCS, vol. 4435, pp. 272—300. Springer, Heidelberg (2008)
[69]. J. Corbet, A. Rubini, G. Kroah-Hartman. Linux Device Drivers, 3rd Edition, Chapter 3, section “The open Method”, pp. 58—59 // O'Reilly Media. 2005.

Introduction to CEGAR —Counter-Example Guided Abstraction Refinement

Khoroshilov A.V., Mandrykin M. U., MutilinV. S.
khoroshilov@ispras.ru, mandrykin@ispras.ru, mutilin@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Precision, completeness and scalability of static verification tools have dramatically improved over the last decade. In particular, automatic checking of moderate-sized software systems has been made possible due to development of CEGAR — Counter-Example Guided Abstraction Refinement. This approach is used in such tools as SLAM, BLAST, SATABS, and CPAchecker. The paper presents an extended review of predicate abstraction-based CEGAR. It provides an introduction to the general principles of CEGAR and describes some implementation details of CEGAR in BLAST and CPAchecker. In particular, the paper concerns deciding the reachability problem for C programs by means of symbolic predicate abstraction. The set of predicates for the abstraction is obtained by Craig interpolation of the logical formulas representing the counterexample traces being discovered during the analysis. This technique is explained by two examples analyzed step-by-step both in intuitive and in formal manner. The explanation proceeds from a number of greatly simplified programs employing a very restricted subset of C language features (e.g. using only a finite set of integer variables) to more complicated programs of arbitrary size with pointers, heap allocations and bit operations. In terms of considered abstract domains the paper describes the simplest fine-grained Cartesian abstraction and a coarse-grained Boolean abstraction with adjustable block encoding. The paper also includes small discussions on common issues arising from verification of real industrial C codebase and current capabilities of existing decision procedure implementations.

Keywords: static verification, predicate abstraction, model checking, counter-example guided abstraction refinement, Craig interpolation, large-block encoding.

References

[1]. Dershowitz N. Software horror stories. URL: http://www.cs.tau.ac.il/~nachumd/horror.html
[2]. Turing A. M. On Computable numbers, with an application to the Entscheidungsproblem. In Proc. London Mathematical Society, pp. 230—265, 1936.
[3]. Floyd R. Assigning Meanings to Programs. Mathematical Aspects of Computer Science, pp. 19—32, 1967.
[4]. Hoare C. An Axiomatic Basis for Computer Programming. Communications of the ACM (CACM), vol. 12, issue 10, pp. 576-580, 1969. doi: 10.1145/363235.363259
[5]. Dijkstra E. A Discipline of Programming. Prentice-Hall, 1976.
[6]. [6]. Millo R. D., Lipton R., Perlis A. Social Processes and Proofs of Theorems and Programs. Communications of the ACM (CACM),vol. 22, issue 5, pp. 271-280, 1979. doi: 10.1145/359104.359106
[7]. Nelson G. Techniques for Program Verification. Technical Report CSL81-10: Xerox Palo Alto Research Center, 1981.

[8]. Nelson G., Oppen D. Fast Decision Procedures Based on Congruence Closure. Journal of the ACM (JACM), vol. 27. pp. 356-364, 1980.
[9]. Shostak R. Deciding Combinations of Theories. LNCS, vol. 138, pp. 209-222, 1982. doi: 10.1007/BFb0000061
[10]. Clarke E. M., Emerson E. A. Synthesis of Synchronization Skeletons for Branching Time Temporal. LNCS, vol. 131, pp. 52-71, 1982. doi: 10.1007/BFb0025774
[11]. Queille J., Sifakis J. Specification and Verification of Concurrent Systems in Cesar. In Proc. of 5th International Symposium on Programming, LNCS, vol. 137, pp. 337-351, 1981. doi: 10.1007/3-540-11494-7_22
[12]. Vardi M., Wolper P. Reasoning about Infinite Computations. Information and Computation, vol. 115, issue 1, pp. 1–37, 1994. doi: 10.1006/inco.1994.1092
[13]. Pnueli A. The Temporal Logic of Programs. In Proc. 18th Annual Symposium on Foundations of Computer Science, pp. 46-57, 1977. doi: 10.1109/SFCS.1977.32
[14]. Emerson E. Temporal and Modal Logic. Handbook of Theoretical Computer Science. Elsevier Science Publishers, vol. B, pp. 995-1072, 1990.
[15]. Khedker U. P., Sanyal A., Karkare B. Data Flow Analysis: Theory and Practice. CRC Press (Taylor and Francis Group), 2009.
[16]. D'Silva V., Kroening D., Weissenbacher G. A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 27, issue 7, pp. 1165-1178, 2009. doi: 10.1109/TCAD.2008.923410
[17]. Jhala R., Majumdar R. Software model checking. ACM Computing Surveys (CSUR), vol. 41, issue 4, article 21, 2009. doi: 10.1145/1592434.1592438
[18]. Beyer D. Petrenko A. Linux Driver Verification. In Proc. Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies, LNCS, vol. 7610, pp. 1-6, 2012. doi: 10.1007/s10009-007-0044-z
[19]. Nesov V. Automatically Finding Bugs in Open Source Programs. Third International Workshop on Foundations and Techniques for Open Source Software Certification, OpenCert 2009, vol. 20, pp.19-29, 2009.
[20]. Engler D., Chelf B., Chou A., Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. In Proc. 4th conference on Symposium on Operating System Design & Implementation (OSDI), vol. 4, pp. 1-16, 2000.
[21]. Lawall J. L., Brunel J., Palix N., Rydhof H. R., Stuart H., Muller G. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In Proc. 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 43—52, 2009.
[22]. Avetisyan A., Borodin A. Mekhanizmy rasshireniya sistemy staticheskogo analiza Svace detektorami novykh vidov uyazvimostej i kriticheskikh oshibok [Mechanisms for extending the system of static analysis Svace by new types of detectors of vulnerabilities and critical errors]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 21, pp. 39-54, 2011 (in Russian).
[23]. Ignatyev V.N. Ispol'zovanie legkovesnogo staticheskogo analiza dlya proverki nastraivaemykh semanticheskikh ogranichenij yazyka programmirovaniya [Using static analysis for checking configurable semantic restrictions on a programming language]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, pp. 169-188, 2012 (in Russian).
[24]. Biere A., Cimatti A, Clarke E., Strichman O., Zhu Y. Bounded model checking. Advances in Computers, vol. 58, 2003.

[25]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. Journal of the ACM (JACM), vol. 50, issue 5, pp. 752-794, 2003. doi: 10.1145/876638.876643
[26]. Thomas D., Moorby P. The Verilog Hardware Description Language. Kluwer Academic Publishers, 1998.
[27]. Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 2988, pp. 168-176, 2004. doi: 10.1007/978-3-540-24730-2_15
[28]. Ivancic F., Yang Z., Ganai M.K., Gupta A., Shlyakhter I., Ashar P. F-soft: Software verification platform. In Proc. Computer Aided Verification (CAV), LNCS, vol. 3576, pp. 301-306, 2005. doi: 10.1007/11513988_31
[29]. Post H., Sinz C., Merz F., Gorges T., Kropf T. Linking functional requirements and software verification. In Proc. 17th IEEE International Requirements Engineering Conference, pp. 295-302, 2009. doi: 10.1109/RE.2009.43
[30]. Donaldson A. F., Kroening D., Ruemmer P. Automatic analysis of DMA races using model checking and k-induction. Formal Methods in System Design, vol. 39, pp. 83-113, 2011. doi: 10.1007/s10703-011-0124-2
[31]. Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J. The Static Driver Verifier Research Platform. Computer Aided Verification (CAV), LNCS, vol. 6174, pp. 119-122, 2010. doi: 10.1007/978-3-642-14295-6_11
[32]. Beyer D., Henzinger T., Jhala R., Majumdar R. The Software Model Checker Blast: Applications to Software Engineering. Int. Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007. doi: 10.1007/s10009-007-0044-z
[33]. Clarke E., Kroening D., Sharygina N., Yorav K. SATABS: SAT-based Predicate Abstraction for ANSI-C. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 3440, pp. 570-574, 2005. doi: 10.1007/978-3-540-31980-1_40
[34]. Beyer D., Keremoglu M.E. CPAchecker: A Tool for Configurable Software Verification. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6806, pp. 184–190, 2011. 10.1007/978-3-642-22110-1_16
[35]. Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).
[36]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an Open Framework for C Verification Tools Benchmarking. In Proc. Perspectives of Systems Informatics (PSI), LNCS, vol. 7162, pp. 82-91, 2012. doi: 10.1007/978-3-642-29709-0_17
[37]. Graf S., Saidi H. Construction of Abstract State Graphs with PVS. In Proc. Computer Aided Verification (CAV), LNCS, vol. 1254, pp. 72-83, 1997. doi: 10.1007/3-540-63166-6_10
[38]. Henzinger T.A., Jhala, R., Majumdar, R., Sutre, G. Lazy abstraction. In Proc. 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), Pages 58-70, 2002. doi: 10.1145/503272
[39]. Cytron R., Ferrante J., Rosen B.K.,Wegman M.N., Zadek F.K. Efficiently computing static single-assignment form and the program dependence graph. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 13, issue 4, pp. 451-490, 1991. doi: 10.1145/115372.115320
[40]. Craig W. Linear reasoning. Symbolic Logic, vol. 22, pp. 250–268, 1957.

[41]. Henzinger T.A., Jhala R., Majumdar R., McMillan K.L. Abstractions from proofs. In Proc. 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), pp. 232-244, 2004. doi: 10.1145/964001.964021
[42]. Beyer D., Cimatti A., Griggio A., Keremoglu M.E., Sebastiani R. Software Model Checking via Large-Block Encoding. In Proc. Formal Methods in Computer-Aided Design (FMCAD), pp. 25–32, 2009. doi: 10.1109/FMCAD.2009.5351147
[43]. Beyer D., Keremoglu M. E., Wendler P. Predicate abstraction with adjustable-block encoding. In Proc. Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 189-198, 2010.
[44]. Berndl M., Lhotak O., Qian F., Hendren L., Umanee N. Points-to analysis using BDDs. Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI), pp. 103 – 114, 2003. doi: 10.1145/781131.781144
[45]. George C. Necula, Scott McPeak, S. P. Rahul, Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In Proc. Conference on Compiler Construction, LNCS, vol. 2304, pp. 213-228, 2002. doi: 10.1007/3-540-45937-5_16
[46]. Nori A.V., Rajamani S.K. An Empirical Study of Optimizations in Yogi. In Proc. 32nd ACM/IEEE International Conference on Software Engineering (ICSE), vol. 1, pp. 355-364, 2010. doi: 10.1145/1806799.1806852
[47]. Cimatti A., Griggio A., Joost S. B., Sebastiani R. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7795, pp. 93—107, 2013.doi: 10.1007/978-3-642-36742-7_7
[48]. Nieuwenhuis R., Rubio A. Paramodulation-Based Theorem Proving. Handbook of Automated Reasoning, Elsevier Science and MIT Press, 2001.
[49]. Kroening D., Strichman O. Decision Procedures: An Algorithmic Point of View. Journal of Automated Reasoning, vol. 51, issue 4, pp. 453-456, 2008. doi: 10.1007/s10817-013-9295-4
[50]. Bozzano M., Bruttomesso R., Cimatti A., Junttila T., Ranise S., Van Rossum P., Sebastiani R. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In Proc. Computer Aided Verification (CAV), LNCS, vol. 3576, pp. 335-349, 2005. doi: 10.1007/11513988_34
[51]. Yorsh G., Musuvathi M. A Combination Method for Generating Interpolants. In Proc. Conference on Automated Deduction (CADE), LNCS, vol. 3632, pp. 353-368, 2005. doi: 10.1007/11532231_26
[52]. Rybalchenko A., Sofronie-Stokkermans V. Constraint Solving for Interpolation. In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, vol. 4349, pp. 346-362, 2007. doi: 10.1007/978-3-540-69738-1_25
[53]. McMillan K. L. An Interpolating Theorem Prover. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), vol. 2988, pp 16-30, 2004. doi : 10.1007/978-3-540-24730-2_2
[54]. Beyer D., Zufferey D., Majumdar R. CSIsat: Interpolation for LA+EUF. In Proc. Computer Aided Verification (CAV), LNCS, vol. 5123, pp. 304-308, 2008. doi: 10.1007/978-3-540-70545-1_29
[55]. Cimatti A., Griggio A., Sebastiani R. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. ACM Transactions on Computational Logic (TOCL), vol. 12, issue 1, 2010. doi: 10.1145/1838552.1838559
[56]. Thi Thieu Hoa Le. A Novel Technique for Computing Craig Interpolants in Satisfiability modulo the Theory of Integer Linear Arithmetic. PhD thesis, 2010.

[57]. De Moura L., Bjorner N. Z3: an efficient SMT solver. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 4963, pp. 337-340, 2008. doi: 10.1007/978-3-540-78800-3_24
[58]. Ball T., Bounimova E., Kumar R., Levin V. SLAM2: Static Driver Verification with Under 4% False Alarms. In Proc. Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 35-42, 2010.
[59]. Beyer D. Competition on Software Verification. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 504-524, 2012. doi: 10.1007/978-3-642-28756-5_38
[60]. Beyer D. Second Competition on Software Verification. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp 504-524, 2012. doi: 10.1007/978-3-642-28756-5_38
[61]. Cimatti A., Micheli A., Narasamdya I., Roveri M. Verifying SystemC: A Software Model Checking Approach. In Proc. Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 51–59, 2010.
[62]. Galloway A., Lüttgen G., Mühlberg J.T., Siminiceanu R.I. Model-Checking the Linux Virtual File System. In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 5403, pp. 74—88, 2009. doi: 10.1007/978-3-540-93900-9_10
[63]. Mühlberg J.T., Lüttgen G. Blasting Linux Code. In Proc. Formal Methods: Applications and Technology, LNCS, vol. 4346, pp. 211—226, 2007. doi: 10.1007/978-3-540-70952-7_14
[64]. Penninckx W., Mühlberg J. T., Smans J., Jacobs B., Piessens F. Sound Formal Verification of Linux’s USB BP Keyboard Driver. NASA Formal Methods, LNCS, vol. 7226, pp. 210-215, 2012. doi: 10.1007/978-3-642-28891-3_21
[65]. Apel S., Speidel H., Wendler P., von Rhein A., Beyer D. Detection of feature interactions using feature-aware verification. In Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 372-375, 2011. doi: 10.1109/ASE.2011.6100075
[66]. Classen A., Heymans P., Schobbens P.-Y., Legay A. Symbolic model checking of software product lines. In Proc. International Conference on Software Engineering (ICSE), pp. 321-330, 2011. doi: 10.1145/1985793.1985838
[67]. Beyer D., Henzinger T. A., Theoduloz G. Program Analysis with Dynamic Precision Adjustment. In Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 29-38, 2008. doi: 10.1109/ASE.2008.13
[68]. Cousot P., Cousot R., Feret J., Mauborgne L., Miné A., Monniaux D., Rival X. Combination of Abstractions in the ASTREE Static Analyzer. In Proc. Advances in Computer Science (ASIAN), LNCS, vol. 4435, pp. 272-300, 2007. doi: 10.1007/978-3-540-77505-8_23
[69]. Corbet J., Rubini A., Kroah-Hartman G. Linux Device Drivers. O'Reilly Media, pp. 58—59, 2005.

Построение спецификаций программных интерфейсов в открытой системе покомпонентной верификации ядра Linux[1]

Новиков Е.М.
novikov@ispras.ru

Аннотация. Одним из наиболее перспективных методов поиска ошибок в программах в настоящее время является статическая верификация. Для успешного применения существующих инструментов к ядру операционной системы Linux приходится проводить верификацию покомпонентно. При этом инструментам необходимо предоставлять модель окружения, отражающую реальное окружение компонентов достаточно точно. Разработка полной модели окружения для компонентов ядра Linux является очень трудоемкой задачей, поскольку программных интерфейсов в ядре очень много и они не являются стабильными. В статье предлагается новый подход к построению спецификаций программных интерфейсов, который позволяет достаточно эффективно применять инструменты статической верификации для проверки выполнения правил использования программных интерфейсов в условиях неполноты модели окружения.

Ключевые слова: ядро Linux; компонент ядра; драйвер устройств; правило использования программных интерфейсов; спецификация; модель окружения; статическая верификация; аспектно-ориентированное программирование; язык программирования Си.

1. Введение

На основе изменений, сделанных в стабильных версиях ядра операционной системы (ОС) Linux за год разработки, был проведен анализ ошибок в компонентах ядра [1]. Результаты данного анализа показали, что среди ошибок, которые не связаны с нарушениями спецификаций устройств, сетевых протоколов, алгоритмов выделения памяти и т.п., около половины всех ошибок в компонентах происходят вследствие нарушения правил

---

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 11.519.11.4006).

использования программных интерфейсов. Для поиска подобных ошибок в настоящее время преимущественно применяются такие подходы, как экспертиза кода [2-4] и динамическое тестирование [5-8]. При использовании данных методов проверки осуществляются с учетом достаточно точных знаний о реальном окружении компонентов ядра ОС Linux, что позволяет выявлять ошибки с высоким уровнем достоверности.

Иначе дело обстоит с различными подходами статического анализа кода. Данные подходы продемонстрировали существенный прогресс в области поиска ошибок за несколько последних десятилетий. В настоящее время на практике, в том числе при поиске ошибок в ядре ОС Linux, очень широко применяются инструменты легковесного статического анализа кода [9-13]. Поскольку данные инструменты в первую очередь нацелены на небольшие время работы и процент ложных сообщений об ошибках, они либо не проводят межпроцедурный анализ, либо делают это достаточно ограниченным образом. Поэтому, даже несмотря на то, что инструментам при анализе доступен весь исходный код ядра ОС Linux, их знания об окружении компонентов ядра являются достаточно поверхностными. Последнее приводит к тому, что инструменты принципиально пропускают ошибки и делают ложные срабатывания.

Наряду с методами легковесного статического анализа активно развивались подходы к статической верификации, нацеленные на обнаружение всех возможных ошибок искомого вида [14-22]. Ввиду текущих ограничений инструментов статической верификации на размер и сложность анализируемого кода, при их применении к ядру ОС Linux приходится ставить задачу верификации покомпонентно. При этом для того, чтобы число ложных срабатываний не было очень большим, для верифицируемого компонента необходимо достаточно точно моделировать его окружение. Модель окружения должна строиться таким образом, чтобы имитировать реальное взаимодействие компонента с так называемой *сердцевиной ядра*, которое заключается в следующем:

- программные интерфейсы компонентов ядра (*функции-обработчики*) регистрируются в сердцевине ядра при вызове стандартной функции инициализации компонентов, а затем вызываются из сердцевины в ответ на системные вызовы со стороны пользовательских приложений и на прерывания со стороны соответствующей аппаратуры;
- программные интерфейсы сердцевины ядра ОС Linux предоставляются компонентам посредством заголовочных файлов аналогично стандартным Си-библиотекам.

Для того, чтобы применить различные инструменты статической верификации для проверки правил использования программных интерфейсов в исходном коде компонентов ядра ОС Linux была разработана открытая система верификации Linux Driver Verification (LDV) [23-25]. Перед проведением непосредственно верификации LDV готовит исходный код компонентов.

Сначала на основе файлов сборки ядра извлекается информация о составе и опциях компиляции и компоновки компонентов. Затем для каждого компонента в отдельности генерируется модель окружения на основе конфигурации, которая описывает несколько типов компонентов и предоставляет шаблоны для остальных. На заключительном этапе подготовки осуществляется постановка задачи верификации для проверки некоторого правила использования программных интерфейсов сердцевины ядра.

Правила использования программных интерфейсов сердцевины ядра в открытой системе верификации LDV задаются с помощью спецификаций. Спецификации включают в себя модельное состояние (набор глобальных переменных) и модельные реализации программных интерфейсов (набор функций). Модельное состояние представляет собой отображение реального состояния ядра ОС Linux. Модельные реализации программных интерфейсов выполняют изменения модельного состояния в соответствии с семантикой моделируемых интерфейсов, а также проверки, которые требуется для правил. При необходимости в спецификации задается привязка точек использования программных интерфейсов сердцевины ядра в компонентах к модельным реализациям интерфейсов с помощью конструкций аспектно-ориентированного программирования [26, 27].

Практическое применение открытой системы верификации LDV для поиска ошибок нескольких видов в коде компонентов ядра ОС Linux с помощью инструментов статической верификации BLAST и CPAchecker позволило обнаружить несколько десятков критических ошибок. Также в ходе экспериментов было выявлено, что из-за неполноты модели окружения (с точки зрения вызовов функций-обработчиков компонентов и программных интерфейсов сердцевины ядра) у инструментов статической верификации достаточно часто происходили ложные срабатывания. Анализ ложных срабатываний инструментов статической верификации является по сути ручным. Данный анализ требует значительных трудозатрат по времени и привлечения дорогостоящих экспертов ядра ОС Linux. Также стоит заметить, что в случае ложных срабатываний инструменты статической верификации не могут автоматически ни обнаружить в коде другие ошибки искомого вида, ни доказать, что их нет.

До сих пор основные усилия для решения данной проблемы были направлены на улучшение полноты модели окружения компонентов. Например, для применения системы верификации SDV [21] разработчики драйверов ОС Microsoft Windows должны предварительно вручную проаннотировать их функции-обработчики. Поскольку программные интерфейсы ядра данной ОС являются стабильными, разработчики SDV сумели описать достаточно хорошую модель для них. Разработка полной модели окружения для компонентов ядра ОС Linux является очень трудоемким процессом ввиду существенно большего многообразия программных интерфейсов ядра, а также их непрерывного изменения по мере развития ядра. Существующие системы верификации компонентов ядра ОС Linux не предложили

автоматизированных решений. В системе верификации Avinux требуется вручную задавать последовательности вызовов функций-обработчиков [22]. В системе верификации DDVerify необходимо полностью описывать модель окружения [20]. Разработчики данной системы построили модель для нескольких типов драйверов устройств, но большие трудозатраты не позволили им развивать данное направление.

Для того чтобы эффективно применять инструменты статической верификации для проверки выполнения правил использования программных интерфейсов в исходном коде компонентов ядра ОС Linux в статье предложен новый метод построения спецификаций программных интерфейсов, который описан в разделе 2. Раздел 3 посвящен инструментарию, реализующему предложенный подход. В разделе 4 представлены результаты практического применения разработанного инструментария. На основании данных результатов делаются выводы о сильных и слабых сторонах предложенного подхода, а также об области его применения. В заключении подводятся итоги работы и рассматриваются направления дальнейшего развития.

2. Метод построения спецификаций программных интерфейсов в условиях неполноты модели окружения

Рассмотрим два примера, которые демонстрируют, как неполнота модели окружения приводит к ложным срабатываниям у инструмента статической верификации BLAST. Для данных примеров проверялась спецификация правила, которое накладывает следующие ограничения на корректное использование в одном потоке функций захвата (*mutex_lock*) и освобождения (*mutex_unlock*) мьютексов[2]:

- запрещается повторно захватывать уже захваченный мьютекс;
- запрещается повторно освобождать уже освобожденный мьютекс;
- запрещается освобождать незахваченный мьютекс;
- запрещается оставлять захваченные мьютексы после завершения работы компонента.

[2] Мьютекс – это примитив синхронизации, который позволяет обеспечить взаимоисключающий доступ к разделяемым данным.

```
  * struct mmc_host *mmc;
  * sdmmc_request(mmc, mpq);
676   struct realtek_pci_sdmmc *host = mmc_priv(mmc);
677   struct rtsx_pcr *pcr = host->pcr;
687   mutex_lock(&pcr->pcr_mutex);
697   mutex_lock(&host->host_mutex);
```

Рисунок 1. *Код драйвера PCI-Express SD/MMC карт drivers/mmc/host/rtsx_pci_sdmmc.c («*» помечена модель окружения, сгенерированная LDV).*

2.1. Пример ложного срабатывания вследствие неполноты модели вызовов функций-обработчиков

В ядро ОС Linux версии 3.8 был добавлен драйвер PCI-Express SD/MMC карт, который состоит из одного Си-файла *drivers/mmc/host/rtsx_pci_sdmmc.c*[3]. Для данного драйвера система верификации LDV генерирует модель вызовов функций-обработчиков неполным образом. На Рис. 1 наглядным образом продемонстрирован один из допустимых данной моделью путей в коде драйвера (опущены несущественные детали). При вызове функции-обработчика *sdmmc_request* ей в качестве параметра передается **mmc** – указатель на структуру *mmc_host*. В строке 676 с помощью функции *mmc_priv* по указателю **mmc** получается поле **host** данной структуры. В строке 677 получается поле **pcr** структуры *realtek_pci_sdmmc*, на которую указывает **host**. Затем в строках 687 и 697 захватываются мьютексы **pcr_mutex** из структуры *rtsx_pcr*, на которую указывает **pcr**, и **host_mutex** из структуры *realtek_pci_sdmmc*, на которую указывает **host**.

На данном пути у инструмента статической верификации BLAST происходит ложное срабатывание. Причиной этого является то, что инструмент считает, что мьютексы **pcr_mutex** и **host_mutex**, получаемые на основе указателя **mmc**, могут совпадать (правильнее говорить, что данные идентификаторы могут обозначать одну и ту же область памяти, то есть могут совпадать адреса **pcr_mutex** и **host_mutex**), поскольку на момент вызова функции-обработчика *sdmmc_request* память под структуру *mmc_host*, на которую указывает **mmc**, не была выделена и инициализирована. Выделение и инициализация памяти

[3] http://lxr.free-electrons.com/source/drivers/mmc/host/rtsx_pci_sdmmc.c?v=3.8.

происходит в функции-обработчике *rtsx_pci_sdmmc_drv_probe*, которая также вызывается имеющейся моделью окружения, но неупорядоченно с *sdmmc_request*.

```
  * struct spi_device *spi;
  * lis3l02dq_probe(spi);
681   struct lis3l02dq_state *st;
682   struct iio_dev *indio_dev;
684   indio_dev = iio_device_alloc(sizeof *st);
689   st = iio_priv(indio_dev);
690   spi_set_drvdata(spi, indio_dev);
695   mutex_init(&st->buf_lock);
  * lis3l02dq_remove(spi);
787   struct iio_dev *indio_dev = spi_get_drvdata(spi);
793   lis3l02dq_stop_device(indio_dev);
765     mutex_lock(&indio_dev->mlock);
766     ...lis3l02dq_spi_write_reg_8(indio_dev, ...);
 89       struct lis3l02dq_state *st = iio_priv(indio_dev);
 91       mutex_lock(&st->buf_lock);
```

Рисунок 2. *Код драйвера линейного акселерометра drivers/staging/iio/accel/lis3l02dq_core.c («*» помечена модель окружения, сгенерированная LDV; волнистой линией подчеркнуты функции, реализации и модели которых не предоставляются для анализа).*

2.2. Пример ложного срабатывания вследствие неполноты модели программных интерфейсов сердцевины ядра

Для драйвера линейного акселерометра, состоящего из одного Си-файла *drivers/staging/iio/accel/lis3l02dq_core.c*[4], система верификации LDV генерирует полным образом ту часть модели окружения, которая имитирует вызовы функций-обработчиков. На Рис. 2 показано, что модель окружения вызывает функцию-обработчик *lis3l02dq_probe*, передавая ей в качестве параметра **spi** – указатель на структуру *spi_device*. В данной функции **indio_dev** присваивается указатель на структуру *iio_dev*, который возвращается функцией *iio_device_alloc* (строка 684). После этого из

[4] http://lxr.free-electrons.com/source/drivers/staging/iio/accel/lis3l02dq_core.c?v=3.8.

indio_dev с помощью функции *iio_priv* получается поле **st** данной структуры (строка 689). Далее посредством функции *spi_set_drvdata* **indio_dev** связывается с **spi** (строка 690), а затем инициализируется мьютекс **buf_lock** из структуры *lis3l02dq_state*, на которую указывает **st** (строка 695).

При вызове функции-обработчика *lis3l02dq_remove*, которой в качестве параметра передается тот же **spi**, что и *lis3l02dq_probe*, в строке 787 из данного **spi** с помощью функции *spi_get_drvdata* получается указатель на структуру *iio_dev* **indio_dev**. Данный **indio_dev** передается в качестве параметра функции *lis3l02dq_stop_device* (строка 793). В строке 765 захватывается мьютекс **mlock** из структуры *iio_dev*, на которую указывает **indio_dev**. Затем **indio_dev** передается функции *lis3l02dq_spi_write_reg_8* в качестве параметра. В этой функции из **indio_dev** с помощью функции *iio_priv* получается поле структуры *iio_dev* **st** (строка 89) и захватывается мьютекс **buf_lock** из структуры *lis3l02dq_state*, на которую указывает **st** (строка 91).

Ложное срабатывание у инструмента статической верификации BLAST обусловлено тем, что инструмент считает, что мьютексы **mlock** и **buf_lock**, получаемые на основе указателя **spi**, могут совпадать, поскольку при анализе BLAST не предоставляются ни реализации, ни модели для функций сердцевины ядра *iio_device_alloc*, *spi_set_drvdata* и *spi_get_drvdata*. В функции *iio_device_alloc* выделяется память под структуру *iio_dev* и, в частности, инициализируется мьютекс **mlock** из данной структуры. Функции *spi_set_drvdata* и *spi_get_drvdata* соответственно помещают и извлекают указатель на структуру *iio_dev* **indio_dev** из структуры *spi_device*, на которую указывает **spi**.

Аналогичные проблемы происходили в ходе практического применения системы верификации LDV при проверке большинства спецификаций правил использования программных интерфейсов сердцевины ядра ОС Linux, поскольку неполная модель окружения не позволяла различать определенные объекты[5]. Например, это правила, связанные с выделением и освобождением ресурсов, с использованием различных механизмов синхронизации, с инициализацией объектов перед их использованием и т.д.

[5] В данной статье под объектами понимаются не классические объекты из объектно-ориентированного программирования, а те объекты, которые интересны с точки зрения специфицируемых программных интерфейсов. Например, для функций захвата и освобождения мьютексов такими объектами являются мьютексы. Различными считаются объекты, занимающие разные области памяти или, иными словами, представимые различными указателями.

2.3. Подход к построению спецификаций программных интерфейсов

С целью уменьшения числа ложных срабатываний инструментов статической верификации, возникающих при проверке выполнения требований спецификаций для правил использования программных интерфейсов в исходном коде компонентов ядра ОС Linux в условиях неполноты модели окружения, в этой статье предложен новый подход к построению спецификаций программных интерфейсов. В основу данного подхода была положена идея различать объекты с помощью анализа тех выражений, в которых они участвуют. Для этого было предложено определить функцию Ψ, которая должна на вход принимать выражение, в котором может использоваться явным или неявным образом некоторый объект интересный с точки зрения специфицируемых программных интерфейсов. На выходе Ψ должна возвращать информацию о том, какому объекту или объектам соответствует рассматриваемое выражение. Для всех выявленных с помощью данной функции объектов в подходе было предложено строить спецификацию таким образом, чтобы хранить информацию об их текущем состоянии в уникальных переменных модельного состояния, которые должны изменяться и проверяться независимо друг от друга в модельных реализациях программных интерфейсов.

Функция Ψ должна удовлетворять следующим условиям:

1. Для двух выражений, которым соответствует один и тот же объект, Ψ должна возвращать одно и то же значение. Данное условие необходимо для того, чтобы гарантировать надежность метода или, иными словами, обнаружение всех возможных ошибок искомого вида.
2. Для двух выражений, которым соответствуют разные объекты, Ψ должна вернуть разные значения. Это условие говорит о нацеленности данной функции на различение объектов и, соответственно, на уменьшение числа ложных срабатываний.

Легко предложить такую функцию, которая удовлетворяет только первому условию, – это функция Ψ', которая для любого выражения возвращает одно и то же значение. Забегая вперед, можно отметить, что результаты экспериментов с Ψ', представленные в подразделе 4.4, показали большое число ложных срабатываний. Также легко предложить функцию Ψ'', которая удовлетворяет только второму условию, – это функция, которая для любого входа всегда возвращает разные значения.

Функция Ψ, которая удовлетворяет одновременно двум условиям в полной мере, должна учитывать, например, алиасы и арифметику с указателями. Для этого Ψ должна уметь решать задачу, которая отчасти сопоставима с задачей инструментов статической верификации. Это означает, что в условиях неполноты модели окружения реализация функции Ψ неизбежно столкнется с

теми же проблемами, что и статическая верификация (см. подразделы 2.1 и 2.2).

Более простое определение функции Ψ можно предложить только при условии выполнения определенных правил кодирования в проекте. Например, если в проекте не используются операции преобразования типов, затрагивающие рассматриваемые объекты, а также другие операции, которые могут привести к совпадению объектов с разными именами, то в качестве Ψ можно рассмотреть функцию, которая возвращает имена объектов для соответствующих выражений[6]. С помощью такой функций для примера из подраздела 2.1 можно выявить мьютексы **pcr_mutex** и **host_mutex**, для которых в подходе предложено использовать различное модельное состояние, например, переменные **ldv_mutex_pcr_mutex** и **ldv_mutex_host_mutex** соответственно. Для мьютексов **mlock** и **buf_lock** (подраздел 2.2) – **ldv_mutex_mlock** и **ldv_mutex_buf_lock** соответственно. Благодаря этому в дальнейшем при проведении верификации становится возможным избежать ложных срабатываний, которые происходили ранее из-за того, что неполная модель окружения не позволяла различить данные мьютексы.

```
 1 struct A {
 2   int x;
 3 } global;
 4 struct B {
 5   int y;
 6 };
 7 int main()
 8 {
 9   struct B *local1;
10   struct A *local2 = (struct A *)malloc(sizeof(struct A));
11   int *z;
12   /* Преобразование типа для адреса глобальной переменной. */
13   local1 = (struct B *)&global;
14   assert(&global.x == &local1->y);  /* Совпадают. */
15   /* Преобразование типа для адресов из кучи. */
16   local1 = (struct B *)local2;
17   assert(&local2->x == &local1->y);  /* Совпадают. */
18   /* Использование алиасов. */
19   z = &global.x;
20   assert(&global.x == z); /* Совпадают. */
21   return 0;
22 }
```

Рисунок 3. Пример программы, в которой не выполняются правила кодирования, позволяющие различать объекты по их именам.

[6] Стоит отметить, что при рассматриваемых условиях данная функция полностью отвечает только первому условию, поскольку интересные с точки зрения специфицируемых программных интерфейсов объекты могут иметь одинаковые имена, но при этом быть вложены в разные структуры (см. подраздел 4.3).

В общем случае рассмотренные правила кодирования могут не соблюдаться. На Рис. 3 приведены три примера, для которых при использовании данного подхода к построению спецификаций программных интерфейсов будут различаться объекты с различными именами, **x**, **y** и **z**, притом, что адреса **x** и **y** совпадают с **z** (строки 14, 17 и 20). Подобное происходит при преобразованиях типа для адреса глобальной переменной (строка 13) и для адреса из кучи (строка 16), а также при использовании алиасов (строка 19). Для компонентов ядра ОС Linux рассмотренные правила кодирования выполняются с высокой степенью достоверности, поэтому подход применим для покомпонентной верификации ядра.

3. Инструментарий для построения спецификаций программных интерфейсов

Первоначально предложенный подход к построению спецификаций программных интерфейсов был реализован в инструменте статической верификации BLAST. В данной реализации функции Ψ передавались на вход те выражения, которые являлись фактическими аргументами специфицируемых программных интерфейсов сердцевины ядра ОС Linux (см. примеры передачи мьютексов функции *mutex_lock* в подразделах 2.1 и 2.2). С целью указания информации о данных интерфейсах инструменту BLAST на вход подавалась дополнительная конфигурация. На основе фактических аргументов специфицируемых программных интерфейсов функция Ψ вычисляла имена интересных объектов. В дальнейшем в данной статье результат работы устроенной таким образом функции Ψ будет называться *подписью аргумента* (по аналогии с повседневной жизнью, где подпись практически однозначно идентифицирует человека).

Для создания соответствующих выявленным объектам переменных модельного состояния и модельных реализаций программных интерфейсов, которые изменяют и проверяют данные переменные независимо друг от друга, имена этих переменных и модельных реализаций программных интерфейсов помечались в спецификации с помощью специального суффикса. Для всех уникальных вычисленных подписей аргументов спецификация дублировалась, причем суффикс заменялся на соответствующую подпись аргумента, что гарантировало уникальность переменных модельного состояния и независимость их изменения и проверки для различных объектов.

Применение данной реализации на практике позволило существенно сократить число ложных срабатываний, благодаря чему повысилась эффективность выявления ошибок. При этом первоначальная реализация подхода обладала несколькими существенными недостатками. Во-первых, она не поддерживала привязку точек использования программных интерфейсов сердцевины ядра в компонентах к модельным реализациям интерфейсов с помощью конструкций аспектно-ориентированного программирования, что

требовалось для большинства спецификаций правил. Во-вторых, в реализации были ошибки, из-за чего подписи аргументов не всегда вычислялись корректно. В-третьих, что являлось наиболее существенным, реализация подхода на основе инструмента BLAST не позволяла воспользоваться ей при применении других инструментов статической верификации.

Для преодоления указанных недостатков была разработана новая реализация подхода. В этой реализации было предложено вычислять подписи аргументов специфицируемых программных интерфейсов с помощью запросов по исходному коду компонентов ядра ОС Linux [28]. Для создания уникальных переменных модельного состояния и модельных реализаций программных интерфейсов, которые изменяют и проверяют данные переменные независимо друг от друга, было предложено задавать спецификации в виде шаблонов [29,30].

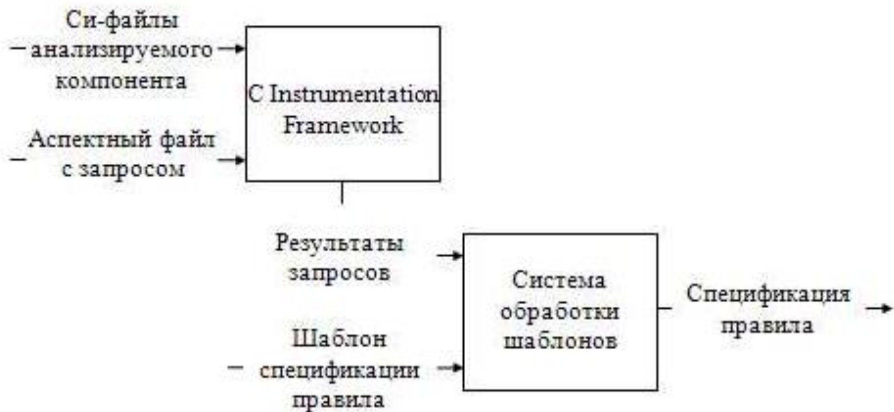


Рисунок 4. Архитектура инструментария для построения спецификаций правил использования программных интерфейсов.

Схема работы нового инструментария, реализующего предложенный метод, показана на Рис. 4. На первом шаге аспектный файл с запросом по исходному коду применяется к Си-файлам анализируемого компонента ядра ОС Linux с помощью C Instrumentation Framework (CIF) [26-28]. Затем получившиеся результаты запросов объединяются (например, при генерации подписей аргументов эта процедура заключается в исключении дубликатов) и используются для построения итоговой спецификации правила на основе шаблона спецификации. На практике аспектный файл с запросом также строится на основе шаблона спецификации правила, что позволяет избежать ненужного дублирования кода.

Новый инструментарий был реализован в компонентах системы верификации LDV:

- Был разработан новый компонент LDV, который выполняет запросы по исходному коду компонентов ядра ОС Linux с помощью CIF,

обрабатывает полученные результаты и готовит на их основе и на основе шаблона финальную спецификацию правила.

- Был доработан существующий компонент Rule Instumentor, который на основе заданной спецификации осуществляет инструментирование исходного кода анализируемого компонента ядра ОС Linux с помощью CIF. Благодаря этому Rule Instrumentor стал получать информацию об аспектном файле с запросом, шаблоне спецификации правила и различных дополнительных настройках на основе базы данных спецификаций правил, поддерживаемых LDV, а также вызывать новый компонент соответствующим образом.

4. Практическое применение разработанного инструментария

В данном разделе будут использоваться следующие обозначения:

- *32_** – спецификации правила, описывающего корректное использование функций захвата и освобождения мьютексов;
- *39_** – спецификации правила, описывающего корректное использование функций захвата и освобождения спин блокировок;
- *32_1a* – спецификация, которая использует подход аспектно-ориентированное программирование для описания привязок модельных реализаций программных интерфейсов к использованию интерфейсов в компонентах ядра ОС Linux;
- *32_7* – спецификация, которая использует первоначальную реализацию подхода, встроенную в инструмент статической верификации BLAST;
- *32_7a* – расширенная версия спецификации *32_1a* в виде шаблона;
- *39_7* – аналог *32_7* для спин блокировок;
- *39_7a* – аналог *32_7a* для спин блокировок;
- *SIMPLE_ID* – механизм генерации подписей аргументов, который принимает во внимание только имя объекта (использовался по умолчанию для спецификаций 32_7a и 39_7a, а также единственный механизм, поддерживаемый для 32_7 и 39_7);
- *COMPLEX_ID* – механизм генерации подписей аргументов, который помимо имени объекта дополнительно использует имя структурного типа, если объект является полем переменной или параметра соответствующего структурного типа или указателя на него;
- *Safe* – вердикт верификации, который говорит о том, что в анализируемой программе нет нарушений проверяемой спецификации;

- *Unsafe* – вердикт верификации, который говорит о том, что в анализируемой программе обнаружено нарушение проверяемой спецификации, которое может быть либо ошибкой, либо ложным срабатыванием;
- *Unknown* – вердикт верификации, который говорит о том, что инструмент статической верификации не смог вынести вердикт *Safe* или *Unsafe* по той или иной причине (например, нехватка памяти или ошибка в парсере инструмента).

Далее с целью демонстрации эффективности предложенного подхода и его реализации в подразделах 4.1-6 приведен анализ результатов проверки спецификаций 32_* и 39_* для компонентов ядра ОС Linux с помощью инструмента статической верификации BLAST. Также подход был успешно использован для ряда других спецификаций, а новая реализация подхода позволила использовать для проверки инструмент статической верификации CPAchecker, но полученные результаты не приводятся в данной статье для краткости. В подразделе 4.7 демонстрируется альтернативный вариант использования подхода к построению спецификаций программных интерфейсов, а в подразделе 4.8 делается обобщение области применения предложенного подхода.

4.1. Сравнение спецификаций 39_7 и 39_7a

Сравнение результатов верификации для спецификаций 39_7 и 39_7a на всех драйверах устройств ядра Linux версии 3.5 с ограничением по памяти 6 Гб показало 324 перехода на 3630 запусков (около 9%):

- 114 Unknown → Safe. Анализ вердиктов Unknown демонстрирует следующее распределение по проблемам: около 80% из-за ошибок в BLAST, по 10% из-за нехватки памяти и времени.
- 40 Unsafe → Safe. Все вердикты Unsafe были обусловлены ошибками реализации подхода в инструменте BLAST.
- 23 Unsafe → Unknown. 19 Unsafe были обусловлены ошибками реализации подхода в инструменте BLAST. Для 4 оставшихся Unsafe соответствующие вердикты Unknown произошли из-за ошибок в BLAST.
- 145 Safe → Unknown. Анализ вердиктов Unknown демонстрирует следующее распределение по проблемам: около 70% из-за ошибок в CIF, по 10% из-за ошибок в CIL (парсер BLAST) и из-за нехватки памяти, 7% и 3% из-за ошибок в BLAST и генераторе модели окружения соответственно.
- 1 Safe → Unsafe. Unsafe не находился раньше по непонятной причине. Найденный Unsafe является ложным срабатываниям, т.к. разные спин блокировки сначала передаются в одну и ту же функцию (*diva_os_enter_spin_lock*) и только потом захватываются с помощью

одного из специфицируемых программных интерфейсов (при этом объекты называются одинаково).

Данное сравнение продемонстрировало, что новая реализация подхода работает в целом лучше, в частности, с ее помощью были преодолены проблемы реализации подхода в инструменте BLAST. Все негативные переходы происходили из-за ошибок в инструментах CIF, BLAST, CIL и генераторе модели окружения.

4.2. Сравнение спецификаций 32_7 и 32_7a

При проведении экспериментов для спецификации 32_7 была сделана специальная настройка системы верификации LDV для того, чтобы уменьшить число проблем, вызванных ошибками реализации подхода в инструменте статической верификации BLAST (использовать данную настройку на постоянной основе нецелесообразно). Также на тот момент были исправлены большинство ошибок в CIF.

Сравнение результатов верификации спецификаций 32_7 и 32_7a на всех драйверах устройств ядра Linux версии 3.5 с ограничением по памяти 6 Гб показало 236 переходов на 3630 запусков (около 7%):

- 134 Unknown → Safe. Анализ вердиктов Unknown демонстрирует следующее распределение по проблемам: около 90% из-за ошибок в BLAST, 10% из-за нехватки памяти.
- 6 Unsafe → Unknown. Четыре Unknown были обусловлены ошибками в BLAST, два были вызваны нехваткой времени.
- 46 Safe → Unknown. Анализ вердиктов Unknown демонстрирует следующее распределение по проблемам: 40% из-за ошибок в CIF, 20% из-за ошибок в CIL, 20% из-за нехватки памяти, по 10% из-за ошибок в BLAST и генераторе модели окружения.

Данное сравнение продемонстрировало, что новая реализация подхода позволяет выявить существенно больше Safe вердиктов при незначительном уменьшении числа Unsafe вердиктов. Все негативные переходы происходили из-за ошибок в инструментах BLAST, CIF, CIL и генераторе модели окружения.

4.3. Сравнение механизмов генерации подписей аргументов SIMPLE_ID с COMPLEX_ID

Сравнение результатов верификации, полученных для спецификации 32_7a с SIMPLE_ID и COMPLEX_ID на всех драйверах устройств ядра Linux версии 3.5 с ограничением по памяти 6 Гб, показало 4 перехода:

- 3 Unsafe → Safe. Все вердикты Unsafe были обусловлены тем, что захватывались мьютексы с одинаковыми именами из разных структур.
- 1 Unsafe → Unknown. Вердикт Unsafe был обусловлен тем, что захватывались мьютексы с одинаковыми именами из разных структур. Вердикт Unknown был вызван нехваткой памяти – в данном случае это лучше, чем получить ложное срабатывание.

Сравнение результатов верификации, полученных для спецификации 39_7a с SIMPLE_ID и COMPLEX_ID на всех драйверах устройств ядра Linux версии 3.5 с ограничением по памяти 6 Гб, показало 10 переходов:

- 7 Unsafe → Safe. Все вердикты Unsafe были обусловлены тем, что захватывались спин блокировки с одинаковыми именами из разных структур.
- 2 Unknown → Safe. Оба вердикта Unknown были связаны с ошибками в инструменте BLAST. Для 39_7 вердикт для одного из данных драйверов был Unsafe, так как захватывались мьютексы с одинаковыми именами из разных структур; для второго – тоже ошибка в инструменте BLAST.
- 1 Unknown → Unsafe. Вердикт Unknown был связан с ошибкой в инструменте BLAST. Для 39_7 вердикт для данного драйвера также был Unsafe.

Таким образом, в ряде случаев COMPLEX_ID позволяет получить корректные вердикты вместо ложных срабатываний, получаемых для SIMPLE_ID из-за того, что захватываются объекты с одинаковыми именами из разных структур. Значит, механизм генерации подписей аргументов COMPLEX_ID целесообразно использовать по умолчанию.

4.4. Сравнение спецификаций 32_1a и 32_7a

Сравнение спецификаций 32_1a и 32_7a (с использованием механизма генерации подписей COMPLEX_ID) на всех компонентах ядра Linux версии 3.8-rc1 с ограничением по памяти 15 Гб показало 188 переходов на 5372 запуска (около 3%):

- 115 Unsafe → Safe. Переходы обусловлены тем, что подход позволил различить мьютексы с разными именами из разных структур.
- 47 Unsafe → Unknown. Переходы обусловлены тем, что подход позволил различить мьютексы с разными именами из разных структур.
- 14 Unknown → Safe. Анализ вердиктов Unknown демонстрирует следующее распределение по проблемам: примерно по 40% из-за ошибок в BLAST и из-за нехватки времени, 20% из-за нехватки памяти.

- 1 Unknown → Unsafe. Вердикт Unknown был из-за ошибки в инструменте BLAST. Найденный Unsafe оказался истинной ошибкой[7].
- 2 Safe → Unknown. 1 Unknown возник из-за нехватки памяти, 1 - из-за ошибки в BLAST.

Таким образом, предложенные подход и его реализация позволяют сократить число ложных срабатываний, возникших вследствие неполноты модели окружения, на 162 (что составляет около 73% от общего числа Unsafe), а в ряде случаев даже помогают преодолеть некоторые ошибки в других инструментах.

4.5. Оценка границы применимости предложенного подхода

Для определения границы применимости предложенного подхода к построению спецификаций в условиях неполноты модели окружения рассмотрим причины оставшихся 59 Unsafe, полученных для спецификации 32_7a с COMPLEX_ID на всех драйверах устройств ядра Linux версии 3.8-rc1 с ограничением по памяти 15 Гб:

```
  * struct usb_interface *intf;
  * synusb_pre_reset(intf);
493   struct synusb *synusb = usb_get_intfdata(intf);
494   struct input_dev *input_dev = synusb->input;
496   mutex_lock(&input_dev->mutex);
  * synusb_reset_resume(intf)
520   return synusb_resume(intf);
475     struct synusb *synusb = usb_get_intfdata(intf);
476     struct input_dev *input_dev = synusb->input;
479     mutex_lock(&input_dev->mutex);
```

Рисунок 5. Код драйвера USB устройств Synaptics drivers/input/mouse/synaptics_usb.c («» помечена модель окружения, сгенерированная LDV).*

- 23 Unsafe остались из-за неполноты и некорректности модели окружения. Например, для драйвера USB устройств Synaptics, который состоит из одного Си-файла *drivers/input/mouse/synaptics_usb.c*[8], система верификации LDV

[7] https://lkml.org/lkml/2013/3/29/313.

[8] http://lxr.free-electrons.com/source/drivers/input/mouse/synaptics_usb.c?v=3.8.

сгенерировала модель окружения таким образом, что предложенный подход не позволил предотвратить ложное срабатывание. На Рис. 5 показано, что модель окружения вызывает функцию-обработчик *synusb_pre_reset*, передавая ей в качестве параметра **intf**. В данной функции из **intf** с помощью функции *usb_get_intfdata* получается **synusb** (строка 493), а из **synusb** получается его поле **input_dev** (строка 494). Затем в строке 496 захватывается мьютекс **mutex** из **input_dev**. Затем вместо функции-обработчика *synusb_post_reset*, которая освобождает данный мьютекс, модель окружения вызывает *synusb_reset_resume*, которая пытается повторно захватить его. Для решения подобных проблем сложно предложить что-нибудь, кроме уточнения конфигурации, на основе которых генерируется модель окружения.

- 16 Unsafe связаны с неточным анализом указателей в инструменте статической верификации BLAST. Например, для файловой системы NCP, которая состоит из нескольких Си-файлов, в том числе *fs/ncpfs/dir.c*, *fs/ncpfs/ncplib_kernel.c* и *fs/ncpfs/sock.c*[9], BLAST не отслеживает изменение поля структурной переменной, передаваемой в качестве параметра вызываемых функций (Рис. 6). В строке 866 захватывается мьютекс **mutex** из **server**, после чего в строке 869 полю **lock** из **server** присваивается 1. Позже, в строке 874, инструмент считает это поле равным нулю и не выполняет освобождение мьютекса **mutex** из **server**. В итоге это приводит к ложному срабатыванию. Ложные срабатывания из-за неточного анализа указателей в инструменте статической верификации BLAST не обусловлены ни спецификацией правила, ни моделью окружения, поэтому они явным образом не влияют на границу применимости предложенного подхода.

[9] http://lxr.free-electrons.com/source/fs/ncpfs/dir.c?v=3.8, http://lxr.free-electrons.com/source/fs/ncpfs/ncplib_kernel.c?v=3.8, http://lxr.free-electrons.com/source/fs/ncpfs/sock.c?v=3.8.

```
  * ncp_readdir(...);
551   ncp_read_volume_list(...);
706     ncp_get_volume_info_with_number(...);
209       ncp_init_request_s(server, 44);
 96         ncp_lock_server(server);
866           mutex_lock(&server->mutex);
869           server->lock = 1;
234       ncp_unlock_server(server);
874         if (!server->lock) {
875           printk(KERN_WARNING ...);
876           return;
877         }
```

Рисунок 6. Код файловой системы NCP fs/ncpfs/dir.c, fs/ncpfs/ncplib_kernel.c и fs/ncpfs/sock.c («» помечена модель окружения, сгенерированная LDV).*

```
 116 static struct mousedev *mousedev_mix;
   * mousedev_init();
1092   mousedev_mix = mousedev_create(...);
   * mousedev_connect(...);
 970   struct mousedev *mousedev;
 973   mousedev = mousedev_create(...);
 977   ... mixdev_add_device(mousedev);
 931     ...mutex_lock_interruptible(&mousedev_mix->mutex);
 936     ...mousedev_open_device(mousedev);
 427       ... mutex_lock_interruptible(&mousedev->mutex);
```

Рисунок 7. Код драйвера мыши IntelliMouse Explorer PS/2 drivers/input/mousedev.c («» помечена модель окружения, сгенерированная LDV).*

- 13 Unsafe остались из-за того, что разные объекты имели одинаковые имена и получались из переменных и параметров с одним и тем же структурным типом. Например, для драйвера мыши IntelliMouse Explorer PS/2, который состоит из одного Си-файла *drivers/input/mousedev.c*[10], в строках 931 и 427 захватываются мьютекс **mutex** из **mousedev_mix** и **mutex** из **mousedev** соответственно,

[10] http://lxr.free-electrons.com/source/drivers/input/mousedev.c?v=3.8.

причем и **mousedev_mix**, и **mousedev** являются указателями на один и тот же структурный тип **struct mousedev** (Рис. 7). В данном случае **mousedev_mix** и **mousedev** указывают на разные объекты (они создаются независимым друг от друга образом в строках 1092 и 973), но предложенный подход не позволяет различить соответствующие мьютексы. В строке 538, которая не попала в рассматриваемый путь **mousedev** присваивается **mousedev_mix**. Поэтому захват в одном потоке мьютексов **mutex** из **mousedev_mix** и **mousedev**, которые указывает на одну и ту же область памяти, мог бы привести к зависанию системы. Данные Unsafe вердикты говорят о том, что предположение об отсутствии использовании алиасов для интересных объектов не всегда выполняется. Но поскольку это не может привести к пропуску ошибок и незначительно увеличивает число ложных срабатываний, этим можно пренебречь.

- 7 Unsafe являются реальными ошибками. Для большей части данных ошибок были сделаны патчи, которые были одобрены разработчиками ядра ОС Linux[11]. Достаточно много ошибок, связанных с некорректным захватом и освобождением мьютексов были обнаружены с помощью системы верификации LDV для более старых версий ядра.

Таким образом, использование предложенного подхода не позволило предотвратить ложные срабатывания в 52 случаях (около 23% от числа всех Unsafe), что является достаточно хорошим показателем на фоне преодоленных 162 ложных срабатываний (около 73% от общего числа Unsafe). Для устранения большинства оставшихся ложных срабатываний необходимо дорабатывать модель окружения и инструменты статической верификации.

4.6. Оценка эффективности работы инструментария

Оценка накладных расходов, возникших при применении разработанного инструментария для построения спецификаций программных интерфейсов, была произведена на основании общего времени работы системы верификации LDV на ряде запусков в различных конфигурациях. Значение имеет именно такая комплексная оценка времени, поскольку сравнивать время подготовки исходного кода к верификации нецелесообразно по нескольким причинам. Во-первых, использование разработанного инструментария требует строго больше времени, чем его неиспользование. Тоже справедливо в отношении времени работы алгоритма генерации подписей COMPLEX_ID по сравнению с SIMPLE_ID. Во-вторых, точно неизвестно, сколько по времени

[11] https://lkml.org/lkml/2013/2/19/468, https://lkml.org/lkml/2012/12/21/319, https://lkml.org/lkml/2013/3/29/313.

работает первоначальная реализация подхода, встроенная в инструмент статической верификации BLAST. Параметры конфигураций, а также результирующее время работы приведены в Табл. 1.

№	Версия ядра ОС Linux	Компоненты ядра	Спецификация правила	Ограничение по памяти (Гб)	Алгоритм генерации подписей	Общее время работы LDV (чч:мм)
1	3.5	drivers/media	32_7	6	SIMPLE_ID	2:37
2	3.5	drivers/media	32_7a	6	SIMPLE_ID	2:44
3	3.5	drivers	39_7	6	SIMPLE_ID	17:58
4	3.5	drivers	39_7a	6	SIMPLE_ID	16:51
5	3.5	drivers	39_7a	6	COMPLEX_ID	17:08
6	3.8-rc1	все	32_1a	15	-	34:16
7	3.8-rc1	все	32_7a	15	COMPLEX_ID	39:23

Таблица 1. *Общее время работы системы верификации LDV для различных конфигураций.*

По данной таблице видно следующее:

- На небольшом подмножестве драйверов устройств (*drivers/media*) для проверки спецификации 32_7a требуется незначительно больше времени, чем для 32_7 (номера запусков 1 и 2). Зато на всех драйверах устройств для проверки спецификации 39_7a требуется примерно на 1 час меньше времени, чем для 39_7 (номера запусков 3 и 4).
- На всех драйверах устройств для проверки спецификации 39_7a при использовании алгоритма генерации подписей COMPLEX_ID требуется незначительно больше времени, чем для SIMPLE_ID (номера запусков 4 и 5).
- На всех компонентах ядра ОС Linux для проверки спецификации 32_7a требуется на 5 с небольшим часов больше, чем для 32_1a (номера запусков 6 и 7). Данное увеличение затрат по времени происходит вследствие того, что время для вынесения вердикта Unsafe достаточно сильно меньше, чем время, требуемое на доказательство Safe. Для 32_7a находится более чем на 160 Unsafe меньше, чем для 32_1a (раздел 4.4).

4.7. Применение предложенного подхода при разработке спецификаций для новых правил

Помимо рассмотренной ранее возможности применения предложенного подхода к построению спецификаций программных интерфейсов в условиях неполноты модели окружения, метод позволил разработать спецификации для трех новых правил, которые требовали инициализировать объекты ядра ОС Linux до их использования. Дело в том, что зачастую инициализация объектов

в компонентах ядра ОС Linux осуществляется в глобальной области видимости, например, в драйвере *drivers/char/virtio_console.c*[12] в строке 84 с помощью макрофункции *DEFINE_SPINLOCK(pdrvdata_lock)* объявляется и инициализируется спин блокировка, а в строке 85 с помощью макрофункции *DECLARE_COMPLETION(early_console_added)* объявляется и инициализируется *completion*. Выполнить инструментирование данных конструкций таким же образом, как, например, вызовы функций, нельзя, поскольку в глобальной области видимости нельзя использовать неконстантные выражения.

С помощью предложенного подхода стало возможным построить спецификацию данных правил. Например, для рассматриваемого драйвера на основе его исходного кода сначала извлекается информация о том, какие спин блокировку (**pdrvdata_lock**) и *completion* (**early_console_added**) он объявляет и инициализирует, а затем эти данные используются для получения итоговой спецификации на основе шаблона.

4.8. Обобщение области применения предложенного подхода

Изначально подход был нацелен на преодоление ложных срабатываний у инструментов статической верификации, возникающих из-за неполноты модели окружения. Как это было показано ранее в данном разделе, с этой задачей подход справился достаточно хорошо. В предыдущем подразделе был продемонстрирован альтернативный вариант использования подхода для разработки спецификаций правил использования программных интерфейсов сердцевины ядра ОС Linux.

Область применения предложенного подхода можно обобщить следующим образом. Посредством аспектно-ориентированного программирования возможно выполнять инструментирование исходного кода компонента ядра с целью привязки точек использования программных интерфейсов сердцевины ядра в компонентах к модельным реализациям интерфейсов [26, 27]. Предложенный подход, который во многом опирается на аспектно-ориентированное программирование, расширяет его возможности. Он позволяет выполнять инструментирование некоторых точек программы на основе информации, полученной для других, связанных с данными некоторым образом точек. Это открывает новые возможности использования аспектно-ориентированного программирования, в частности для спецификации программных интерфейсов.

---

[12] http://lxr.free-electrons.com/source/drivers/char/virtio_console.c?v=3.8.

5. Заключение

В статье был предложен подход к построению спецификаций программных интерфейсов, нацеленный на уменьшение числа ложных срабатываний инструментов статической верификации, которые возникают при проверке выполнения требований спецификаций для правил использования программных интерфейсов в исходном коде компонентов ядра ОС Linux в условиях неполноты модели окружения. Была разработана новая реализация данного подхода на основе C Instrumentation Framework, которая показала в целом более хорошие результаты, чем его первоначальная реализация на основе инструмента статической верификации BLAST. Новый инструментарий позволил использовать для проверки спецификаций инструмент статической верификации CPAchecker, а также применять подход для спецификаций, для которых требуется привязка модельных реализаций программных интерфейсов к местам использования интерфейсов в коде компонентов на основе конструкций аспектно-ориентированного программирования.

Результаты практического применения новой реализации данного подхода в составе системы верификации Linux Driver Verification продемонстрировали, что подход справился с данной задачей достаточно успешно: он позволил сократить число ложных срабатываний более чем на 73%. В статье было показано, что для преодоления оставшихся ложных срабатываний необходимо дорабатывать модель окружения и инструменты статической верификации.

Помимо ядра ОС Linux предложенный подход к построению спецификаций программных интерфейсов в условиях неполноты модели окружения может быть применен при покомпонентной верификации в других проектах, в которых соблюдаются определенные правила кодирования. Более того, подход расширяет существующие возможности аспектно-ориентированного программирования, поскольку он позволяет выполнять инструментирование некоторых точек программы на основе информации, полученной для других, связанных с данными некоторым образом точек. В статье приводятся примеры, как с помощью новых возможностей были разработаны спецификации для нескольких правил использования программных интерфейсов.

Новый инструментарий был реализован в компонентах Linux Driver Verification. Для удобства использования в будущем планируется перенести соответствующую функциональность в C Instrumentation Framework и расширить его интерфейс. Также планируется расширить возможности реализации по выявлению объектов, интересных с точки зрения правил использования программных интерфейсов.

Литература

[1]. В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов. Анализ типовых ошибок в драйверах операционной системы Linux. Труды Института системного программирования РАН, т. 22, стр. 349-374, 2012.

[2]. M.E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, 15(3), 182-211, 1976.

[3]. B. Boehm, V. Basili. Software Defect Reduction Top 10 List. IEEE Computer, 34(1), 135-137, January, 2001.

[4]. E.S. Raymond. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, Sebastopol, CA, USA, 2001.

[5]. P. Larson. Testing Linux with the Linux Test Project. Proceedings of the Ottawa Linux Symposium, Ottawa, Ontario, Canada, June 26–29, 2002.

[6]. Novell System Test Kit for Linux. https://www.suse.com/partners/ihv/pdf/SystemTestKit-7.1-Linux-10_22_12.pdf.

[7]. Oracle Linux Tests. https://oss.oracle.com/projects/olt/dist/documentation/OLT_TestCoverage.pdf.

[8]. А.В. Цыварев, В.А. Мартиросян. Тестирование драйверов файловых систем в ОС Linux. Труды Института системного программирования РАН, т. 23, стр. 413-426, 2012.

[9]. D. Engler, B. Chelf, A. Chou, S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. Proceedings of the 4th conference on Symposium on Operating System Design & Implementation, San Diego, California, pp.1-16, October 22-25, 2000.

[10]. Coverity Scan: 2011 Open Source Integrity Report. http://www.coverity.com/library/pdf/coverity-scan-2011-open-source-integrity-report.pdf.

[11]. H. Stuart. Hunting bugs with Coccinelle. Masters Thesis, University of Copenhagen, August, 2008.

[12]. Sparse - a Semantic Parser for C. https://sparse.wiki.kernel.org/index.php/Main_Page.

[13]. Dan Carpenter. Killing Bugs in C with Smatch. Linux Plumbers Conference, Santa Rosa, California, September 7-9, 2011.

[14]. P. Shved, M. Mandrykin, V. Mutilin. Predicate Analysis with Blast 2.7. Proceedings of TACAS, vol. 7214, pp. 525–527, 2012.

[15]. S. Löwe, P. Wendler. CPAchecker with Adjustable Predicate Analysis. Proceedings of TACAS, vol. 7214, pp. 528–530, 2012.

[16]. C. Sinz, F. Merz, S. Falke. LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation. Proceedings of TACAS, vol. 7214, pp. 542–544, 2012.

[17]. D. Beyer. Competition on software verification. Proceedings of TACAS, vol. 7214, pp. 504–524, 2012.

[18]. Results of the 2013 2nd International Competition on Software Verification. http://sv-comp.sosy-lab.org/2013/results/index.php.

[19]. D. Engler, M. Musuvathi. Static analysis versus model checking for bug finding. In VMCAI, pp. 191-210, 2004.

[20]. T. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher. Model checking concurrent Linux device drivers. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 501-504, 2007.

[21]. T. Ball, E. Bounimova, V. Levin, R. Kumar, J. Lichtenberg. The Static Driver Verifier Research Platform. CAV 2010, 2010.

[22]. H. Post, W. Küchlin. Integration of static analysis for Linux device driver verification. The 6th Intl. Conf. on Integrated Formal Methods, IFM 2007, 2007.
[23]. В.С. Мутилин, Е.М. Новиков, А.В. Страх, А.В. Хорошилов, П.Е. Швед. Архитектура Linux Driver Verification. Труды Института системного программирования РАН, т. 20, стр. 163-187, 2011.
[24]. A. Khoroshilov, V. Mutilin, E. Novikov, P. Shved, A. Strakh. Towards an Open Framework for C Verification Tools Benchmarking. Proceedings of the Eighth International Andrei Ershov Memorial Conference «Perspectives of Systems Informatics» (PSI 2011), pp. 82-91, 2011.
[25]. Открытая система верификации Linux Driver Verification. http://forge.ispras.ru/projects/ldv.
[26]. E. Novikov. One Approach to Aspect-Oriented Programming Implementation for the C programming language. roceedings of the 5th Spring/Summer Young Researchers' Colloquium on Software Engineering, Yekaterinburg, pp. 74-81, 12-13 May, 2011.
[27]. Реализация аспектно-ориентированного программирования для языка Си C Instrumentation Framework. http://forge.ispras.ru/projects/cif.
[28]. Е.М. Новиков, А.В. Хорошилов. Использование аспектно-ориентированного программирования для выполнения запросов по исходному коду программ. Труды Института системного программирования РАН, т. 23, стр. 371-386, 2012.
[29]. D. Chamberlain, D. Cross, A. Wardley. Perl Template Toolkit. O'Reilly Media, pp. 592, December, 2003.
[30]. N. Gunton. Creating Modular Web Pages With EmbPerl. March 13, 2001, http://www.perl.com/pub/2001/03/embperl.html.

Building Programming Interface Specifications in the Open System of Componentwise Verification of the Linux Kernel

Novikov E.M.
novikov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Nowadays static verification is one of the most promising methods for finding bugs in programs. To apply successfully existing tools for the Linux kernel one needs to perform componentwise verification. Such verification needs an environment model that reflects a real environment of components rather accurately. Development of the complete environment model for Linux kernel components is a very time-consuming task since there are too many programming interfaces in the kernel and they are not stable. The given paper suggests a new approach for building programming interface specifications. This approach allows one to apply static verification tools efficiently (with small number of false alarms but without missed bugs) for checking rules of programming interfaces usage under conditions of the incomplete environment model, if component developers obey a specific coding style. Two implementations of the suggested approach were developed. The first one extended the BLAST static verification tool while the second one utilized C Instrumentation Framework – an aspect-oriented programming implementation for the C programming language. The latter allowed to use various static verification tools, like BLAST, CPAchecker, CBMC, etc., for checking specifications. In practice that implementation helped to reduce the number of false alarms on more than 70% for some rules of programming interfaces usage. The paper shows that to avoid left false alarms one needs to develop more precise environment models and to use more accurate static verification tools. Besides the Linux kernel the suggested approach can be applied for componentwise verification in projects where developers obey the specific coding style.

Keywords: Linux kernel; kernel component; device driver; rule of programming interfaces usage; specification; environment model; static verification; aspect-oriented programming; C programming language.

References

[1]. Mutilin V.S., Novikov E.M., Khoroshilov A.V. Analiz tipovykh oshibok v drajverakh operatsionnoj sistemy Linux [Analysis of typical faults in Linux operating system drivers]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, pp. 349-374, 2012 (in Russian).

[2]. Fagan M.E. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, vol. 38, issue 2/3, pp. 258-287, 1999. doi: 10.1147/sj.382.0258

[3]. Boehm B., Basili V. Software Defect Reduction Top 10 List. Computer, vol. 34, issue 1, pp. 135-137, 2001. doi: 10.1109/2.962984

[4]. Raymond E.S. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly Media, 1999.

[5]. Larson P. Testing Linux with the Linux Test Project. In Proc. Ottawa Linux Symposium, 2002.
[6]. Novell System Test Kit for Linux. https://www.suse.com/partners/ihv/pdf/SystemTestKit-7.1-Linux-10_22_12.pdf.
[7]. Oracle Linux Tests. https://oss.oracle.com/projects/olt/dist/documentation/OLT_TestCoverage.pdf.
[8]. TSyvarev A.V., Martirosyan V.A. Testirovanie drajverov fajlovykh sistem v OS Linux [Testing of Linux File System Drivers].Trudy ISP RAN [The Proceedings of ISP RAS], vol. 23, pp. 413-426, 2012 (in Russian).
[9]. Engler D., Chelf B., Chou A., Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. In Proc. 4th conference on Symposium on Operating System Design & Implementation (OSDI), vol. 4, pp. 1-16, 2000.
[10]. Coverity Scan: 2011 Open Source Integrity Report. http://www.coverity.com/library/pdf/coverity-scan-2011-open-source-integrity-report.pdf.
[11]. Stuart H. Hunting bugs with Coccinelle. University of Copenhagen, Masters Thesis, 2008.
[12]. Sparse - a Semantic Parser for C. https://sparse.wiki.kernel.org/index.php/Main_Page.
[13]. Shved P., Mandrykin M., Mutilin V. Predicate Analysis with Blast 2.7. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 525–527, 2012. doi: 10.1007/978-3-642-28756-5_39
[14]. Shved P., Mandrykin M., Mutilin V. Predicate Analysis with Blast 2.7. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 525–527, 2012. doi: 10.1007/978-3-642-28756-5_39
[15]. Löwe S., Wendler P. CPAchecker with Adjustable Predicate Analysis. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 528–530, 2012. doi: 10.1007/978-3-642-28756-5_40
[16]. Sinz C., Merz F., Falke S. LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 542–544, 2012. doi: 10.1007/978-3-642-28756-5_44
[17]. Beyer D. Competition on Software Verification. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 504-524, 2012. doi: 10.1007/978-3-642-28756-5_38
[18]. Results of the 2013 2nd International Competition on Software Verification. http://sv-comp.sosy-lab.org/2013/results/index.php.
[19]. Engler D., Musuvathi M. Static analysis versus model checking for bug finding. In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, vol. 2937, pp. 191-210, 2004. doi: 10.1007/978-3-540-24622-0_17
[20]. Witkowski T., Blanc N., Kroening D., Weissenbacher G. Model checking concurrent Linux device drivers. In Proc. 22nd IEEE/ACM international conference on Automated Software Engineering (ASE), pp. 501-504, 2007. doi: 10.1145/1321631.1321719
[21]. Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J. The Static Driver Verifier Research Platform. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6174, pp. 119–122, 2010. 10.1007/978-3-642-14295-6_11
[22]. Post H., Küchlin W. Integrated static analysis for Linux device driver verification. In Proc. Integrated Formal Methods (IFM), LNCS, vol. 4591, pp. 518-537, 2007. doi: 10.1007/978-3-540-73210-5_27

[23]. Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).
[24]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an Open Framework for C Verification Tools Benchmarking. In Proc. Perspectives of Systems Informatics (PSI), LNCS, vol 7162, pp. 82-91, 2012. doi: 10.1007/978-3-642-29709-0_17
[25]. Open verification system Linux Driver Verification. http://linuxtesting.ru/ldv.
[26]. Novikov E. One Approach to Aspect-Oriented Programming Implementation for the C programming language. In Proc. 5th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), pp. 74-81, 2011.
[27]. C Instrumentation Framework: aspect-oriented programming implementation for the C programming language. http://forge.ispras.ru/projects/cif.
[28]. Novikov E.M., Khoroshilov A.V. Ispol'zovanie aspektno-orientirovannogo programmirovaniya dlya vypolneniya zaprosov po iskhodnomu kodu programm [Using Aspect-Oriented Programming for Querying Source Code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 23, pp. 371-386, 2012 (in Russian).
[29]. Chamberlain D., Cross D., Wardley A. Perl Template Toolkit. O'Reilly Media, 2003.
[30]. Gunton N. Creating Modular Web Pages With EmbPerl. 2001, http://www.perl.com/pub/2001/03/embperl.html.

Автоматизация регрессионного тестирования при помощи анализа трасс событий[1]

Владимир Федотов
Институт системного программирования РАН
e-mail: vfl@ispras.ru

Аннотация. В настоящей работе представлен process mining алгоритм, предназначеный для применения в инструментированной распределенной системе. Подразумевается, что система инструментирована таким образом, что взаимодействия между модулями системы проходят через единую шину, связывающую все компоненты системы. Записанные при обработке на шине события анализируются представленным в работе алгоритмом, результатом работы которого является модель взаимодействий между модулями системы. Полученная модель используется для разработки покрывающего регрессионного тестового набора.

Ключевые слова: регрессионное тестирование; распределенные системы; process mining

1 Введение

Одним из трендов развития современных информационных систем является переход к большей распределенности компонентов этих систем. Код этих компонентов выполняется не просто на разных процессорах, но и на разных машинах. Эти компоненты не имеют общей памяти и, все чаще, не имеют общего хранилища данных. Взаимодействия между ними построены на обмене сообщениями поверх различных протоколов, среди которых чаще других используется HTTP.

Причины, по которым архитектура систем стремится к распределенности, разнообразны. Это и популярность Agile разработки с ее короткими итерациями, которые легче подстраивать под небольшие модули системы. Это и дешевизна современных средств виртуализации. Это и существенно возросшая скорость обмена данными через

---

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 11.519.11.4024).

локальные сети и интернет. Это и современное стремление бизнеса к глобализации, которое ведет к тому, что в один бизнес-процесс могут быть вовлечены IT-системы нескольких компаний.

Другой тенденцией, без сомнения связанной с первой, является уменьшение функциональной значимости каждого компонента в отдельности. Зачастую один компонент выполняет единственную, достаточно тривиальную, функцию – например, маршрутизацию входящих запросов на другие компоненты системы по определенным правилам.

С точки зрения разработки, такие компоненты легче поддерживать, так как их код относительно тривиален и не превышает объемом 1 kloc. Главное же преимущество такого подхода состоит в масштабируемости, которая имеет первостепенную важность для современной информационной системы. Так, система, состоящая из набора модулей, не связанных общим хранилищем данных, масштабируется линейно.

Внедрение подобной архитектуры приводит к смещению акцентов при тестировании как отдельных модулей, так и всей системы вцелом. С одной стороны юнит-тестирование теряет значимость, так как каждый компонент в отдельность тривиален. С другой стороны, интеграционное и системное тестирование приобретают особую важность, так как именно проблемы взаимодействия компонентов являются наиболее критическими для работы распределенной системы.

Большая часть взаимодействий происходит поверх стандартных интернет-протоколов, таких как HTTP, которые не предполагают никакой типизации передаваемых объектов. Таким образом, все ошибки типов и форматов данных обнаруживаются лишь в run-time и лишь на этапе интеграционного тестирования.

Другим следствием распределенности системы является то, что документация также становится все более «распределенной». Описание фактически единого бизнес-процесса либо разбивается на спецификации множества независимых модулей, либо отсутствует вовсе. Первой жертвой этой проблемы становится регрессионное тестирование, для обеспечения которого необходима полная и актуальная документация.

Негативно влияет на качество регрессионного тестирования и сама природа распределенной системы. Обмен сообщениями между модулями подразумевает, что многие логические зависимости в системе являются семантическими зависимостями между передаваемыми данными. Так, например, строки 'aaaa' и 'bbbb' могут представлять различные классы экививалентности для вызываающей системы в зависимости от того, какие атрибуты связаны с ними в вызываемой системе.

Определить эти классы эквивалентности по спецификации вызывающей системы невозможно. Более того, функциональные спецификации зачастую не содержат описания семантических связей данных или

описывают их как «корректные» и «некорректные», не раскрывая смысл этих понятий.

2 Data-mining в целях тестирования

Прямым следствием отсутствия полной документации и сложности поиска семантических зависимостей является то, что для создания полноценных регрессионных тестовых наборов тестировщикам все чаще приходится прибегать к реверс-инжинирингу требований, выполняемому на предыдущей версии системы.

Впрочем, подобный ad-hoc метод также несет в себе значительные риски, так как не предоставляет никаких гарантий полноты полученного набора требований.

Более перспективным методом является run-time анализ поведения существующей системы в продуктивном окружении. Такой анализ возможен благодаря тому, что большинство коммерческих систем снабжены средствами мониторинга и отчетности (так называемый business intelligence), записывающими обрабатываемые системой события.

Хорошие результаты также приносит анализ системных журналов. Однако в связи с их объемом и неупорядоченностью данных такой анализ требует применения специальных средств и методов.

В действительности, анализ данных, или data mining является черезвычайно популярной темой исследований. Методы data mining работают как в уже упомянутых средствах business intelligence, так и применяются ad hoc для анализа неподготовленных данных.

Цели анализа могут быть различными: поиск статистических зависимостей, маркетинговые исследования, анализ производительности системы [2], [5]. В некоторых случаях методы data mining применимы даже для анализа эффективности дизайна нового сайта (сколько кликов сделал пользователь, прежде чем нашел желаемое) и поиска угроз безопасности системы [7].

Методы data minig также применимы в тестировании, прежде всего именно для run-time анализа системы. Так, в [6] описан перспективный подход к построению регрессионных тестовых наборов и поиску классов эквивалентности данных при помощи анализа трасс событий в системе.

В целом, такие подходы, позволяющие восстановить модель принятия решений системой, выделились в отдельный класс, получивший название «process mining» [8].

Для нас в первую очередь интересно применение методов process mining для восстановления модели системы, применимой для регрессионного тестирования. Перспективный подход описан в [1] как дельта-анализ двух моделей системы (например, действующей и доработанной версий) и тестирование соответствия модели фактической реализации системы и эталонной модели системы.

Впрочем, подход описывает лишь общие принципы такого тестирования и не подходит для внедрения в виде законченной методологии тестирования.

Отдельно стоит рассмотреть алгоритмы, используемые в подходах process mining. На настоящий момент они разбиваются на общие и специфические. Специфические алгоритмы эффективны на узком классе данных. Общие алгоритмы эффективны на всех классах данных, однако гораздо менее эффективны, чем специфические алогритмы, на конкретном классе данных.

Таким образом, применение process mining для строго определенной задачи подразумевает либо разработку нового спецического алгоритма, либо поиск наиболее подходящего из существующих.

Критерием эффективности такого алгоритма является сравнение с одним из общих алгоритмов. В качестве эталонного алгоритма принято выбирать α -алгоритм, описанный в [8].

В настоящей работе мы описываем алгоритм process mining, применимый для получения регрессионных тестовых наборов, и основные методы его применения.

3 Process-mining событий в ESB

Для применения mining-подходов первоочередное значение имеет способ, которым собираются события для анализа. Мы применяем подход к инструментации тестового окружения при помощи открытых ESB-платформ, описанный в [3] и [4]. Инструментированное подобным образом окружение позволяет собирать трассу сообщений, представленных HTTP-запросами и ответами компонентов системы.

3.1 Свойства сообщений

В нашем подходе мы абстрагируемся от синхронности или асинхронности анализируемых взаимодействий, поэтому для выделения зависимостей (корелляции) между сообщениями все проходящие через шину события дискретизируются.

На практике это означает, что все попавшие на шину события собираются в единую очередь. Обработчик очереди запускается через равные промежутки времени, представляющие собой период дискретизации. При обработке очередного сообщения к нему добавляется текущее значение счетчика периодов или, проще говоря, шаг, на котором оно было обработано.

Таким образом, обработанное сообщение записывается в трассу с указанием 1) шага обработки; 2) компонента-отправителя; 3) компонента-получателя. Также записываются поля сообщения и их значения. Подразумевается, что сообщение представимо в виде XML-

документа (1). Таким образом, поля сообщения являются листьями документа, значения полей – текстовыми значениями листьев.

$$m = \{t, src, dst, F, V\} \quad (1)$$

Дискретизация времени обработки сообщений влечет за собой интересные следствия. В частности, существенно упрощается анализ зависимостей между сообщениями. Так, сообщения, где $m_1.t = m_2.t$, независимы, если $m_1.src \neq m_2.src$. В противном случае, это означает, что отправитель выполяет два параллельных запроса.

3.2 Корелляция сообщений

Применяемый нами подход основан на поиске зависимых сообщений в трассе событий. Так, например, зависимыми могут быть пара запрос-ответ или запрос, порождающий последующий запрос. В различных BPEL-инструментах принято использовать так называемый идентификатор корелляции, который передается по цепочке от предыдущего сообщения следующему, однозначно идентифицируя, таким образом, зависимость. Разумеется, в общем случае рассчитывать на наличие такого идентификатора не приходится.

Для определения отношения корелляции сообщений мы используем сравнение основных параметров сообщения, как показано в (2). Таким образом, сообщения зависимы, только если отправитель последующего сообщения явлется получателем предыдущего, и шаг обработки различается строго на 1.

$$m_1 \to m_2 \Leftarrow m_2.src = m_1.dst \wedge m_2.t - m_1.t = 1 \quad (2)$$

Отношение (2) является верным (свидетельствует о корелляции) во всех случаях, где дискретизация сообщений выполнена корректно, то есть зависимые сообщения обработаны на 1) разных шагах обработки; 2) последовательных шагах обработки.

3.3 Интеракции

Говоря о корелляции сообщений, логично рассматривать цепочку кореллирующих сообщений, как единую сущность. В BPEL каждая такая цепочка является результатом выполения процесса для каждого конкретного входящего запроса или тестового воздействия. Само существование и длина цепочек сообщений обусловлены распределенностью исследуемых систем. Структура цепочек может различаться в зависимости от используемых в системе транспортных

протоколов. Так, например, при использовании синхронного HTTP взаимодействие в системе из трех модулей будет представлено цепочкой «вложенных» взаимодействий:

$$Client \rightarrow A \rightarrow B \rightarrow C \rightarrow B \rightarrow A \rightarrow Client\ .$$

В данном случае синхронность подразумевает, что модуль А удерживает HTTP-сессию с клиентом открытой до тех пор, пока остальные модули системы не завершат работу. Цепочку, подобную этой, мы называем интеракцией. Каждая интеракция состоит из попарно кореллирующих сообщений: $i = m_1 \rightarrow m_2 \rightarrow \ldots \rightarrow m_n$

Таким образом, трасса, собраная в ходе работы системы, представляет собой набор интеракций. Каждая интеракция состоит из сообщения-стимула и последующих, кореллирующих друг с другом сообщений (3). Вырожденная интеракция состоит только из сообщения-стимула, представляя собой случай, когда никакой реакции не последовало.

$$i = \{m_0, M\} \qquad (3)$$

Конечным сообщением в интеракции является сообщение, на которое не последовала реакция на следующем шаге обработки. Конечное сообщение может совпадать с начальным или не быть единственным в случаях, когда один из модулей системы генерирует несколько параллельных запросов.

В общем виде, интеракция является деревом, корень которого обозначает сообщение-стимул, а листья являются конечными сообщениями. Интеракция, в которой все листовые сообщения являются конечными, помечается как закрытая.

3.4 Свойства интеракций

Мы рассматриваем интеракции как прообраз тестов. Так, m_0 является тестовым воздействием, а M – ожидаемой реакцией. Можно сказать, что отношение $M_x ?My, m_0^x = m_0^y$ является автоматическим тестовым оракулом.

Разумеется, каждая трасса может содержать произвольное количество идентичных интеракций. Поэтому использование для тестов всей трассы нецелесообразно. Мы анализируем идентичность интеракций и составляющих их сообщений на нескольких уровнях для того, чтобы отсечь дублирующие друг друга интеракции.

Для определения степени равенства двух сообщений мы используем три различных по строгости отношения. Так, отношение эквивалентности подразумевает равенство всех атрибутов сообщений (4). Отношение равенства подразумевает равенство всех адресных атрибутов

сообщений, а также их структуры (5), данные в сообщениях могут различаться. Наконец, отношение схожести подразумевает только равенство всех адресных атрибутов сообщений (6). Отношение экивалентности является наиболее сильным, отношение схожести наиболее слабым: $m_x = m_y \Rightarrow m_x ; m_y \Rightarrow m_x \approx m_y$

$$m_x = m_y \Leftarrow m_x.t = m_y.t \wedge m_x.src = m_y.src \wedge m_x.dst = m_y.dst \wedge (\forall f_x : f_x = f_y \wedge v(f_x) = v(f_y)) \quad (4)$$

$$m_x ; m_y \Leftarrow m_x.t = m_y.t \wedge m_x.src = m_y.src \wedge m_x.dst = m_y.dst \wedge (\forall f_x : f_x = f_y) \quad (5)$$

$$m_x \approx m_y \Leftarrow m_x.t = m_y.t \wedge m_x.src = m_y.src \wedge m_x.dst = m_y.dst \quad (6)$$

Из отношения равенства вложенных сообщений следует отношение равенства интеракций. Так как интеракция может содержать сообщения, состоящие в разных степенях равенства, отношение равенства интеракций соответствует наиболее слабому из отношений равенства вложенных сообщений. Правила (7, 8, 9) определения равенства двух интеракций представлены для интеракций из двух сообщений и аналогичны для интеракций из большего числа сообщений.

$$i_x = i_y \Leftarrow \forall m_x : m_x = m_y \quad (7)$$

$$i_x ; i_y \Leftarrow \forall m_x : m_x ; m_y \vee i_x = i_y \quad (8)$$

$$i_x \approx i_y \Leftarrow \forall m_x : m_x \approx m_y \vee i_x = i_y \vee i_x ; i_y \quad (9)$$

3.5 Алгоритм формирования модели интеракций

Нулевым шагом алгоритма является формирование интеракций. Формирование интеракций происходит непосредственно при обработке очередного сообщения. Если сообщение кореллирует с одной из существующих интеракций, эта интеракция дополняется новым сообщением. Если же нет, сообщение инициирует создание новой интеракции. Ситуация, при которой сообщение кореллирует более чем с одной незакрытой интеракцией, невозможна. Завершение сбора трассы

означает, что новые интеракции перестают добавляться в трассу, все незакрытые интеракции помечаются как закрытые.

Далее из трассы удаляются все дублирующиеся интеракции, такие что $i_1 = i_2 \vee i_1;i_2$

Результирующий набор интеракций трансформируется в направленный ациклический граф следующим образом: для каждой последующей интеракции каждое последующее сообщение является новым узлом графа, если граф не содержит другого узла, для которого выполнялось бы условие $m_x;m_y$. Для каждого узла достижимыми являются узлы, представленные коррелирующими сообщениями.

Трасса также может содержать интеракции, все сообщения в которых не кореллируют друг с другом. Такие интеракции являются независимыми и относятся к разным моделям. Количество сформированных моделей по итогам обработки трассы равно количеству попарно независимых интеракций в ней.

3.6 Расширяемость моделей

Модели, полученные в результате обработки трассы, расширяемы. Это означает, что модель, полученная при обработке одной трассы, может быть дополнена моделью, полученной при обработке другой трассы. Модель M_1 расширяема моделью M_2, если в модели M_1 существует хотя бы одно сообщение, для которого выполняется отношение равенства для любого другого сообщения из модели M_2 (10).

$$\exists m_x(M_1) : m_x;m_y(M_2) \qquad (10)$$

3.7 Генерация тестов

Конечной целью майнинга событий в системе является генерация тестового набора, покрывающего модель взаимодействий в системе. Такой тестовый набор может использоваться в дальнейшем, как регрессионный или как промежуточный, для взаимосвязи различных этапов тестирования за счет расширяемости получаемых моделей.

Тестовый набор, сгенерированный на основе модели, состоит из набор тестовых воздействий, представляющих собой все начальные узлы модели, и оракула, представленного остальными узлами модели. В данном случае мы исходим из того, что в ходе выполнения полученных тестов будет сгенерирована новая модель взаимодействий, сравнение которой с эталонной (предыдущей) моделью дает представление о различиях в поведении системы.

В целом, критерием корректности считается следующее: тест T является пройденным, если в результате тестового воздействия m_0 порождается интеракция I, такая что: $\forall m_x(T)\exists m_y(I): m_x; m_y$

3.8 Предположения и ограничения

Основные ограничения подхода следуют из используемого отношения зависимости (2).

Подразумевается, что анализируемая система реализует event-driven архитектуру. То есть сообщения не могут возникать в ней спонтанно, например, вследствие срабатывания таймера. Каждое сообщение в системе появляется лишь после 1) обработки предыдущего сообщения; 2) пользовательского воздействия.

Описанное отношение зависимости (2) имеет смысл только для систем взаимодействия, которые построены на обмене сообщениями. Оно не имеет смысла для систем, обменивающихся файлами или транзакциями. Так, например, подход неприменим для ETL-систем.

Подразумевается, что все зависимые сообщения обрабатываются в рамках периода дискретизации системы. Таким образом, если время обработки запроса превышает период дискретизации, то ответное сообщение не будет считаться результатом обработки запроса. Впрочем, в реальных системах, при периоде дискретизации равному половине HTTP-таймаута, влияние этого ограничения на наш взгляд минимально.

4 Заключение

В настоящей работе был представлен специфический алгоритм process mining, который предназначен для применения в инструментированной распределенной системе. Подразумевается, что система инструментирована таким образом, что взаимодействия между модулями системы проходят через единую шину, где дискретизируются и записываются. Записанные таким образом события анализируются представленным в работе алгоритмом, результатом работы которого является модель взаимодействий между модулями системы. Полученная модель используется для разработки покрывающего регрессионного тестового набора.

В последующих работах будет представлено сравнение результатов разработанного алгоритма и generic алгоритма, в качестве которого используется α-алгоритм. Также мы планируем подготовить описание метода применения разработанного алгоритма для внедрения 1) в процесс регрессионного тестирования распределенной системы; 2) в стандартный, построенный в соответствии с V-моделью процесс тестирования. Там же будут представлены примеры применения и результаты апробации подхода на реальных проектах.

В долгосрочной перспективе мы планируем разработать полноценный набор средств для инструментации распределенной системы. Предложенный в этой работе подход будет дополнен методом выделения классов эквивалентности данных и их внедрения в тестовую модель. Также мы планируем разработать набор эвристик для поиска скрытых состояний в распределенной системе. Такие эвристики позволят выявлять и внедрять в модель события, изменяющие состояние того или иного объекта в системе.

Список литературы

[1] W. M. P. Van Der Aalst. Business alignment: using process mining as a tool for Delta analysis and conformance testing. *Requirements Engineering*, 10(3):198–211, August 2005.

[2] A Adriansyah and JCAM Buijs. Mining Process Performance from Event Logs. *Business Process Management Workshops*, 2013.

[3] Vladimir Fedotov. Run-time monitoring for model-based testing of distributed systems. In *SYRCoSE 2012*, 2012.

[4] VN Fedotov. Service-oriented approach to integration testing in distributed systems. *SYRCoSE 2010*, page 58, 2010.

[5] Rattikorn Hewett. Mining software defect data to support software testing management. *Applied Intelligence*, 34(2):245–257, September 2009.

[6] Mark Last, Menahem Friedman, and Abraham Kandel. The data mining approach to automated software testing. *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03*, page 388, 2003.

[7] Jianxiong Luo and SM Bridges. Mining fuzzy association rules and fuzzy frequency episodes for intrusion detection. *International Journal of Intelligent Systems*, 15(I):1–36, 2000.

[8] W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, September 2004.

Automated event trace analysis for regression testing

Vladimir Fedotov

Abstract. Modern IT systems become more and more distributed, both in literal sense - as business logic spreads across applications running in several different network domains, and in a metaphorical sense - as expert knowledge about system's inner working spreads across different outsource companies and hundreds of documents. In this paper we are discussing regression testing as one of the issues related to that trend.

When it comes to testing, quality of specifications is a major issue. They are either unmanageably big and outdated, or very much fragmented - each specifying its own module, but lacking perspective on how system works as a whole. In our practice, it often comes down to reverse engineering to discover what exactly different parts of the system are expecting from each other, but such ad hoc techniques are not welcome in the domain of testing.

Different approach relies on analyzing system's events trace that is provided either by business activity monitoring tools, transaction logs or, simply, log files. There is a lot of different data mining techniques already developed; we are mostly interested in process mining as a way of discovering system's decision model.

A key factor of any process mining application is the algorithm of discovering events relation. It can be either generic algorithm like Alpha-algorithm or a specific algorithm tailored for a particular set of possible events. In this paper we provide an outline for a specific algorithm that we use to discover events relation in a special environment.

The environment we are using is specifically designed for the goal and has central part in the process. It is used for mediating interactions happening between system's modules in run-time as a discrete process thus providing us with a very straightforward way of determining events causality.

Keywords: regression testing; distributed systems; process mining

References

[1]. W. van der Aalst. Business alignment: using process mining as a tool for Delta analysis and conformance testing. Requirements Engineering, 10(3):198–211, August 2005.

[2]. A. Adriansyah and J. Buijs. Mining Process Performance from Event Logs. Business Process Management Workshops, 2013.

[3]. V.N. Fedotov. Run-time monitoring for model-based testing of distributed systems. SYRCoSE 2012.

[4]. V.N. Fedotov. Service-oriented approach to integration testing in distributed systems. SYRCoSE 2010.

[5]. R. Hewett. Mining software defect data to support software testing management. Applied Intelligence, 34(2):245–257, September 2009.

[6]. M. Last, M. Friedman, A. Kandel. The data mining approach to automated software testing. Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03, page 388, 2003.

[7]. J. Luo, S.M. Bridges. Mining fuzzy association rules and fuzzy frequency episodes for intrusion detection. International Journal of Intelligent Systems, 15(I):1–36, 2000.
[8]. W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering, 16(9):1128–1142, September 2004.

Распределенные горизонтально масштабируемые решения для управления данными

С. Д. Кузнецов, А. В. Посконин
kuzloc@ispras.ru, aposk@yandex.ru

Аннотация. В современном мире всё острее встает проблема работы с огромными объемами данных и большими нагрузками. Крупные Web-приложения, социальные сети, различные научные исследования, бизнес-аналитика, а также множество других областей, так или иначе, сталкиваются с проблемами управления и анализа данных большого объема («big data»). Кроме анализа уже накопленного объема данных, возникают задачи манипулирования данными под большой нагрузкой, характерные, например, для Web-приложений. В таких проектах большое количество пользователей одновременно читают и пишут информацию, что требует от системы управления данными не только большой пропускной способности и низких задержек, но и масштабируемости, надежности и определённых гарантий согласованности данных. Несмотря на большую популярность, опыт применения и универсальность, традиционные SQL-ориентированные СУБД зачастую не могут удовлетворить требования современных приложений, что привело к появлению большого числа специализированных распределённых систем, способных лучше справляться с возникающими задачами. В данной статье предлагается обзор некоторых современных решений, обеспечивающих масштабируемость при работе с большими объемами данных под высокими нагрузками.

Ключевые слова: MapReduce; NoSQL; нереляционные модели данных; масштабируемость; согласованность данных; NewSQL.

1. Введение

В современном мире постоянно возникают задачи, связанные с обработкой больших объемов данных и высокой нагрузкой. Важным примером могут служить Web-приложения, такие как социальные сети, торговые платформы, крупные новостные порталы и т.д. Все они обладают своей спецификой, но их общей особенностью является работа с огромными объемами данных и большим количеством пользователей, которые могут читать или изменять эти данные. Кроме того, объемы данных в таком случае постоянно растут, равно

как и аудитория, что приводит к вопросу о том, как масштабировать приложение (и используемые решения для работы с данными). Обычно рассматривают масштабируемость в двух направлениях: вертикальную (наращивание вычислительной мощности одного сервера) и горизонтальную (наращивание числа серверов). Вертикальная масштабируемость, как известно, имеет свой предел, а финансовые затраты нелинейно зависят от производительности сервера. Таким образом, приемлемым вариантом оказывается только горизонтальное масштабирование, то есть увеличение числа серверов при возрастающих нагрузках и объемах данных.

К сожалению, горизонтальное масштабирование системы управления данными является достаточно сложной задачей. Традиционные SQL-ориентированные СУБД изначально создавались для работы на одной машине и потому плохо приспособлены для работы в кластере или облаке, что привело к разработке новых распределённых систем и подходов, решающих проблему горизонтальной масштабируемости. Несмотря на то, что все они направлены на решение проблемы масштабируемости, каждое решение имеет те или иные достоинства и недостатки и может эффективно справляться, таким образом, лишь с определённым классом задач. Мысль о том, что универсальных систем управления данными не существует, подчеркивает общую тенденцию к переходу от повсеместного использования «универсальных» SQL-ориентированных СУБД к выбору системы исходя из решаемой задачи (см., например, [1]).

Данная работа предлагает обзор некоторых популярных в настоящее время систем управления данными и подходов, обеспечивающих горизонтальную масштабируемость и производительность, необходимую современным приложениям. Раздел 2 посвящен особенностям работы с данными в распределённых архитектурах, являющихся основой большинства современных систем. Раздел 3 содержит обзор технологии распределённых вычислений MapReduce, нашедшей широкое применение в области анализа и обработки больших данных и реализованной во многих современных системах. Раздел 4 посвящен NoSQL-системам, отступающим от реляционной модели данных и традиционной транзакционной семантики. Наконец, раздел 5 описывает некоторые современные решения с поддержкой SQL и ACID-транзакций, способные составить конкуренцию NoSQL-системам в ряде задач.

Прежде чем перейти к описанию особенностей распределённых архитектур, необходимо дать пояснение относительно терминологии. В данной работе активно используется термин «система управления данными» (data management system) как наиболее общий и охватывающий как СУБД, так и файловые системы, облачные хранилища и системы типа «ключ-значение». В силу большого разнообразия моделей данных и подходов использование данного термина представляется более удачным, чем термина «СУБД».

2. Принципы организации распределённых систем управления данными

Распределённая архитектура позволяет достичь не только горизонтальной масштабируемости (в идеале – линейного роста производительности при добавлении новых узлов), но и увеличить надежность системы с помощью хранения нескольких копий данных. Децентрализация и отсутствие общих ресурсов позволяют избежать узких мест и единых точек отказа. В распределённых системах управления данными используются два основных приёма для организации хранения данных (практически всегда применяемых совместно):

Разделение данных (шардинг, sharding) – подход, при котором каждый узел системы содержит свою часть данных и выполняет операции над ними. Этот метод является основным средством обеспечения горизонтальной масштабируемости, однако теперь операции над несколькими объектами могут вовлечь в работу несколько узлов, что требует активной передачи данных по сети. Кроме того, при увеличении числа узлов, хранящих данные, увеличивается вероятность сбоев, что требует наличия избыточности – применения репликации (см. далее). Шардинг также порождает такие задачи, как распределение данных по узлам, балансировка нагрузки, оптимизация сетевых взаимодействий и т.д.

Репликация (replication) – подход, при котором одни и те же данные хранятся на нескольких узлах в сети. Репликация помогает повысить надежность системы и справляться как со сбоями отдельных узлов, так и с потерей целого кластера. Кроме того, она позволяет масштабировать операции чтения (несколько реже - записи). Серьезной задачей является поддержка согласованного состояния копий данных (реплик): при синхронном обновлении реплик увеличивается время ответа системы, а при асинхронном - возникает промежуток времени, когда реплики находятся в несогласованном состоянии.

Существуют две основные схемы репликации (поддерживающие как синхронные, так и асинхронные варианты):

Ведущий-ведомый (master-slave) – при такой схеме репликации операции модификации данных обрабатывает только ведущий узел (master), а сделанные изменения синхронно или асинхронно передаются на ведомого (slave). Чтения могут осуществляться как с ведущего узла (который гарантированно содержит последнюю версию данных), так и с ведомого (данные могут быть несколько устаревшими при асинхронной репликации и «отставать» от ведущего).

Ведущий-ведущий (master-master, multi-master) – при этой схеме все узлы могут обрабатывать операции записи и передавать обновления остальным. В этом случае реализовать синхронную репликацию достаточно сложно, к тому

же резко возрастают задержки, связанные с сетевыми взаимодействиями. При асинхронном обновлении возникает другая проблема – могут появиться конфликтующие версии данных, которые требуют наличия механизма для определения и разрешения конфликтов (автоматически или на уровне приложения).

2.1 Модели согласованности

Применение репликации в распределённой системе порождает задачу поддержания идентичного состояния копий данных на разных узлах (и, соответственно, видимых разными клиентами). Для обозначения гарантий согласованности данных, которые предоставляются системой, используют термин «модель согласованности» (consistency model). Следует отметить, что слово «согласованность» в этом контексте отличается от свойства согласованности из определения ACID-транзакций. Модель согласованности определяет физическую согласованность состояния данных на разных узлах системы, в то время как в случае ACID имеется в виду логическая согласованность (в рамках ограничений целостности, определённых для базы данных). Ниже приводятся примеры моделей, часто применяемых в распределённых системах управления данными (см., например, [2]):

Согласованность в «конечном счёте» (eventual consistency) гарантирует, что при отсутствии новых обновлений в течение некоторого времени все копии данных (реплики) станут согласованными.

Монотонные чтения (monotonic reads) являются усилением согласованности «в конечном счёте» и гарантируют, что если какое-либо значение было прочитано клиентом, то последующие чтения никогда не вернут предыдущие значения.

Чтение своих записей (read your writes) также усиливает согласованность «в конечном счёте», давая гарантию того, что клиент всегда увидит данные, которые он до этого записал. Эта модель может также комбинироваться с монотонными чтениями.

Мгновенная согласованность (immediate consistency) означает, что как только операция модификации данных успешно завершена, все клиенты мгновенно увидят это изменение. Такую модель согласованности поддерживают, например, системы с синхронной репликацией.

Многие NoSQL-системы обеспечивают лишь согласованность «в конечном счёте» или её вариации. При этом возникает период времени, в течение которого данные находятся в несогласованном состоянии, называемый «окном несогласованности» (inconsistency window). Его величина зависит (при отсутствии сбоев) от скорости распространения обновлений, загруженности системы и количества узлов, содержащих реплики данных. Приложения, использующие системы управления данными, допускающие несогласованность, должны уметь справляться с появлением несколько

«устаревших» данных (в [3] проверяется, насколько часто это может происходить на практике при использовании различных облачных NoSQL-систем). Кроме того, при одновременных обновлениях реплик данных возникают конфликтующие версии. Для обнаружения конфликтов применяются такие подходы, как временные метки и векторные часы (см., например, [4]). Для разрешения конфликтов существуют различные стратегии, но обычно наиболее точное решение о том, какую версию выбрать, может сделать только само приложение (о различных стратегиях разрешения конфликтов см., например, [5]).

В некоторых системах уровень поддержки согласованности можно варьировать, используя различные конфигурации, настройки и/или операции. Рассмотрим, как этого можно добиться на практике с помощью кворума. Пусть данные реплицируются по N-узлам; обозначим через R – число узлов, к которым обращается клиент при чтении данных, а через W – число узлов, от которых ожидается подтверждение успешной записи. Изменяя эти параметры, можно добиться различного поведения распределённой системы:

При $R + W > N$ множества реплик для записи и чтения пересекаются, а значит, чтение всегда возвратит результат успешно завершившейся операции записи (мгновенная согласованность).

При $R + W \leq N$ невозможно гарантировать возврат последней версии данных, гарантируется лишь согласованность «в конечном счёте».

При $R = N$, $W = 1$ поддерживается мгновенная согласованность, а операции записи работают быстро за счёт более медленных и дорогих чтений (с N реплик). При $W = N$, $R = 1$ получается обратная ситуация.

Следует отметить, что чем ближе значения R и W к общему числу реплик N, тем больше вероятность, что операция может завершиться ошибкой из-за выхода каких-либо узлов из строя.

Вариации согласованности «в конечном счёте» (например, наличие монотонных чтений и чтений своих записей) во многом зависят от реализации взаимодействия клиентов с системой (привязывается ли клиент к конкретному серверу, используется ли узел-маршрутизатор, поддерживаются ли версии объектов и т.д.). Более подробно о нюансах моделей согласованности читатель может узнать, например, из [2] и [6].

2.2 Теорема CAP

Для обоснования компромиссов, выбираемых распределёнными системами, часто приводится эмпирическое утверждение, известное как теорема CAP, или теорема Брюэра (Eric Brewer) [7]. Это утверждение гласит, что распределенная система не может гарантировать одновременного выполнения следующих трёх свойств:

1) *Согласованность (Consistency)* - все узлы в каждый момент времени имеют согласованные данные (все пользователи в любой момент времени видят одинаковые данные).

2) *Доступность (Availability)* - при выходе из строя каких-либо узлов оставшиеся узлы должны продолжать функционировать.

3) *Устойчивость к разделению (Partition Tolerance)* - если из-за сбоя сети система распадается на группы узлов, не связанные между собой, то каждая группа должна продолжать функционировать.

Несмотря на то, что данное утверждение само по себе недостаточно формализовано, оно было уточнено и доказано для некоторых частных случаев [8]. Однако желание обеспечить устойчивость к разделению сети и высокую доступность системы – не единственные причины для ослабления согласованности данных. Как отмечает Дэниэл Абади (Daniel Abadi) в [9], формулировка теоремы CAP не учитывает такого важного свойства системы, как величина задержки при ответе на запрос пользователя. Усиление модели согласованности ведёт в общем случае к более высоким задержкам (достаточно вспомнить, например, случай синхронной репликации, когда обновления должны распространиться по большинству узлов или даже по всем узлам для того, чтобы операция изменения данных считалась завершенной). Особенно важно минимизировать передачу данных по сети между узлами в случае большой географической распределённости кластеров.

3. MapReduce

Модель распределённых вычислений MapReduce была впервые представлена компанией Google в 2004 году [10]. Технология MapReduce предназначена для организации распределённых вычислений на больших кластерах недорогих (и потому потенциально ненадежных) машин. Основной процесс обработки данных состоит из двух шагов: Map и Reduce. Каждый из этих двух шагов описывается пользовательской функцией, при этом сложности и детали исполнения программы на кластере машин скрываются от пользователя или разработчика.

Функция Map производит предварительную обработку данных – на вход функции подается ключ и связанные с ним данные, а в результате её работы генерируются промежуточные пары (ключ, значение). Каждый узел кластера выполняет функцию Map на своей назначенной порции данных (в идеале, находящейся на том же узле, чтобы минимизировать пересылку данных по сети). Кроме того, на одном узле могут одновременно выполняться несколько Map-задач (это число зависит от параметров узла). Промежуточным этапом между Map и Reduce является этап перемешивания (shuffle), в результате которого на вход каждой исполняемой функции Reduce поступает один из промежуточных ключей и список значений, соответствующих этому ключу (на одном узле могут одновременно исполняться несколько Reduce-задач).

Функция Reduce производит свертку и возвращает итоговый список значений (чаще всего ноль значений либо единственное значение, например, сумму). Таким образом, результаты применения функции Reduce к каждому промежуточному ключу и списку значений, ассоциированных с этим ключом, формируют окончательный результат.

К результатам исполнения Map на одном узле может быть применена промежуточная функция Combiner для уменьшения объема данных. Часто это функция является той же, что используется на этапе Reduce, только действует она не сразу на все промежуточные ключи и значения, а лишь на те, которые получены на каждом узле. Таким образом, Combiner не может заменить Reduce, но может существенно сократить объем данных, передаваемых на этапе перемешивания.

MapReduce позволяет решать множество задач, связанных с анализом и обработкой больших объемов данных, за приемлемое время благодаря высокому параллелизму. Кроме того, подход MapReduce устойчив к сбоям узлов и позволяет динамически распределять Map- и Reduce-подзадачи по узлам кластера, принимая во внимание фактическое распределение данных по узлам кластера. В настоящее время существуют реализации MapReduce как в виде отдельных библиотек и программных каркасов (например, Apache Hadoop [11]), так и встроенные в различные системы управления данными (например, MongoDB [12], CouchDB [13], Riak [14]). Далее будет кратко рассмотрена одна из самых популярных реализаций MapReduce – Apache Hadoop.

3.1 Apache Hadoop

Apache Hadoop [11] является открытой реализацией MapReduce на Java, следующей принципам, предложенным Google в [10]. Hadoop включает в себя четыре модуля:

1. Hadoop Common (библиотеки и утилиты для поддержки других модулей Hadoop)
2. Hadoop Distributed File System, HDFS (реализация распределённой файловой системы)
3. Hadoop YARN (программный каркас для управления заданиями и ресурсами кластера)
4. Hadoop MapReduce (реализация MapReduce, основанная на YARN)

Далее будут кратко рассмотрены HDFS и MapReduce, которые при совместном использовании обеспечивают эффективную параллельную обработку данных. К сожалению, рассмотреть все нюансы реализации и функционирования Apache Hadoop невозможно в рамках статьи; более подробную информацию о работе с Apache Hadoop читатель может найти, например, в [15].

3.1.1 Hadoop Distributed File System (HDFS)

HDFS является распределённой файловой системой, разработанной согласно принципам организации Google File System (GFS). HDFS предназначена для надежного хранения файлов большого объема на кластере машин путём разбиения файлов на достаточно крупные блоки (по умолчанию – 64 мегабайта). При этом на разных машинах хранится несколько копий одного и того же блока (по умолчанию – 3 копии). HDFS позволяет осуществлять запись только в конец файла, чтение же разрешено и с произвольных позиций. Организация файловой системы иерархическая: поддерживается корневой каталог и подкаталоги, которые могут содержать файлы и другие каталоги.

С точки зрения физической организации в HDFS выделяется так называемый узел имен (name node) и узлы данных (data node). Узел имен хранит метаданные файловой системы, такие как имена файлов и каталогов, информацию о распределении блоков по узлам данных и т.д. Чтение и запись данных осуществляется без непосредственного участия узла имен, таким образом, он не является узким местом всей системы (кроме случаев хранения большого числа маленьких файлов, что является неэффективным). Предусмотрены также механизмы резервного копирования и восстановления узла имен в случае сбоев.

Таким образом, HDFS представляет собой распределённую файловую систему для надежного хранения больших неизменяемых файлов на кластере недорогих машин. Основное преимущество HDFS – возможность использовать информацию о распределении блоков по узлам, что позволяет, например, эффективно планировать и назначать Map-задачи для MapReduce, запуская их на тех узлах, где физически расположены данные. Стоит отметить, что Apache Hadoop также поддерживает и ряд других файловых систем, а HDFS может применяться не только для поддержки MapReduce.

3.1.2 Hadoop MapReduce

Hadoop MapReduce позволяет эффективно выполнять MapReduce-задания на кластере машин. Исходные данные обычно хранятся в HDFS, что позволяет эффективно использовать сведения о расположении блоков. Итоговый результат также обычно сохраняется в HDFS из соображений надежности, а промежуточные результаты Map-обработки эффективнее хранить в локальных файловых системах узлов. Шаги обработки могут быть описаны как на Java, так и на любом другом языке, поддерживающем работу со стандартными потоками ввода-вывода ОС. Hadoop MapReduce может работать с различными форматами входных данных, которые, кроме того, определяют осмысленные способы разбиения входных данных на порции для Map-подзадач. Источниками входных данных могут служить не только файлы из HDFS, но и, например, базы данных; можно также реализовать поддержку собственного

формата входных данных. Аналогичным образом дело обстоит и с генерируемыми данными.

Один узел в кластере выполняет роль координирующего узла (job tracker), остальные узлы непосредственно выполняют подзадачи общей MapReduce-задачи (такой узел называется «task tracker» и содержит несколько слотов для подзадач). Координирующий узел следит за общим ходом выполнения задания, назначает Map- и Reduce-подзадачи по узлам, используя сведения о расположении данных и загруженности узлов, перезапускает подзадачи на других узлах в случае сбоев и т.д.

3.1.3 Резюме

HDFS позволяет хранить данные большого объема на кластере машин, однако эта файловая система не предназначена для работы с постоянно изменяемыми данными. Тем не менее, существует ряд проектов (например, Apache HBase [16] и Apache Cassandra [17], также рассматриваемые в данной работе), использующих HDFS и позволяющих эффективно работать с такими данными. Кроме того, эти данные могут служить входом для MapReduce.

Hadoop MapReduce позволяет эффективно распараллеливать анализ данных большого объема, однако не всегда бывает удобно описывать Map- и Reduce-функции на языках программирования. Для преодоления этой проблемы может быть использован, например, проект Apache Pig [18], поддерживающий высокоуровневый язык запросов (Pig Latin), который затем транслируется в последовательность MapReduce-программ для выполнения на кластере. Pig Latin также при необходимости может быть расширен с помощью пользовательских функций, написанных на Java, Python и других языках.

Apache Hadoop является мощным и очень популярным инструментом для работы с данными большого объема, применяемым как самостоятельно, так и в качестве основы для множества других проектов, таких как Pig, Hive, HBase, Cassandra, Mahout и др. [11]

4. NoSQL-системы

Термин «NoSQL» был впервые применен в 1998 году Карло Строцци (Carlo Strozzi) в качестве названия для его небольшой реляционной СУБД, которая не использовала язык SQL для манипулирования данными [19]. С 2009 года термин «NoSQL» стал использоваться уже для обозначения растущего числа распределенных систем управления данными, которые отказывались от поддержки ACID-транзакций (Atomicity, Consistency, Isolation, Durability – Атомарность, Согласованность, Изолированность, Постоянство хранения) – одного из ключевых принципов работы с реляционными базами данных [20]. По мнению Строцци, современные системы категории NoSQL точнее было бы называть нереляционными («NoREL») [19]. В настоящее время термин

«NoSQL» обычно расшифровывается как «Not Only SQL» , то есть «Не Только SQL» [21].

Главной предпосылкой к появлению систем, относимых к NoSQL, послужила растущая потребность в горизонтальной масштабируемости приложений, то есть в возможности наращивать производительность путём добавления новых вычислительных узлов к уже работающим. Таким образом, большинство NoSQL-систем изначально проектировались и создавались для работы в распределенной среде - кластере или облаке, где применение традиционных SQL-ориентированных систем связано с определёнными трудностями (см., например, [22], [23]). Основной причиной отказа от поддержки транзакционной семантики послужила сложность эффективной реализации транзакций в распределённой среде: в общем случае приходится использовать двухфазный протокол фиксации транзакций, который требует пересылки большого количества сообщений по сети (см., например, [24]). Хотя NoSQL-системы обычно не поддерживают ACID-транзакции в полном объеме, в ряде случаев поддерживаются, например, атомарные операции для чтения и модификации, оптимистические блокировки и другие инструменты, помогающие упростить разработку приложений в условиях параллельного доступа к данным.

В данном разделе будут рассматриваться системы, относимые в публикациях к категории NoSQL. К сожалению, рассмотреть все существующие NoSQL-решения не представляется возможным (на момент написания работы список [21] насчитывал порядка 150 систем), а многие системы достаточно сложны и обладают богатой функциональностью, поэтому в соответствующих разделах будут приведены только ключевые особенности конкретных решений, такие как модель данных, возможности масштабирования и т.д. Кроме того, NoSQL-движение является достаточно молодым, поэтому многие проекты находятся в стадии активной разработки и претерпевают значительные изменения от версии к версии. Наиболее актуальную и полную информацию о рассматриваемых системах читатель может получить, используя ссылки, приведённые в тексте.

4.1 Модели данных и классификация

Важным отличием систем категории NoSQL от реляционных баз данных являются их нереляционные модели данных и способы осуществления запросов. В целом модели данных, лежащие в основе NoSQL, значительно проще, чем классическая реляционная модель, что в ряде случаев облегчает работу с ними. Обычно (хотя и с некоторыми вариациями) NoSQL-системы делят (см., например, [25], [26], [27]), исходя из модели данных, на следующие основные классы:

1) Системы «ключ-значение» (Key-Value Stores)

2) Документные СУБД (Document Stores)

3) Системы типа Google BigTable (Extensible Record Stores / Wide Column Stores / Column Families)

Иногда под термином «NoSQL» понимают также вообще все системы, не являющиеся реляционными (SQL-ориентированными), однако чаще всего имеются в виду именно представители приведённых трёх классов систем. Это деление является достаточно общим, так как внутри каждой группы системы значительно различаются и в плане поддержки согласованности данных, и в нюансах работы с ними (например, атомарность операций, использование блокировок или мультиверсионного доступа (MVCC, Multi-Version Concurrency Control) [28]), однако оно отражает основные аспекты и область применения соответствующих NoSQL-систем. Кроме того, многие системы имеют черты более чем одного класса, и потому иногда их трудно классифицировать по такому принципу. Далее будет дана общая характеристика каждого из классов и рассмотрены примеры некоторых конкретных систем, которые могут быть к ним отнесены.

4.2 Системы «ключ-значение»

NoSQL-системы типа «ключ-значение» хранят данные (неструктурированные или структурированные) и позволяют иметь доступ к ним при помощи единственного уникального ключа. Работа с данными обычно осуществляется с помощью простых операций вставки, удаления и поиска по ключу. Вторичные ключи и индексы в таких системах не поддерживаются. При этом может поддерживаться некоторая структура данных, позволяющая менять отдельные поля объекта, но не позволяющая строить по ним запросы (в этом заключается основное отличие систем типа «ключ-значение» от документных СУБД [25]). В этом отношении системы «ключ-значение» похожи на популярную распределённую систему кэширования в оперативной памяти Memcached [29], но предоставляют постоянное хранение данных и ряд дополнительных возможностей. Существуют NoSQL-системы, обладающие обратной совместимостью с Memcached, что позволяет использовать тот же интерфейс и те же клиентские библиотеки, облегчая переход с Memcached (например, MemcacheDB [30], Couchbase Server [31] и др.). Далее будут подробнее рассмотрены некоторые системы класса «ключ-значение».

4.2.1 Project Voldemort

Project Voldemort [32] – система управления данными типа «ключ-значение» с открытым исходным кодом, реализованная на Java и активно используемая в социальной сети LinkedIn [33]. Помимо скалярных значений, Project Voldemort допускает также списки значений и записи с именованными полями, ассоциируемые с одним ключом. Самим значением и ключом может являться любая сущность, для которой определена сериализация (правила

преобразования объекта в последовательность байтов и обратно). Вся работа с данными осуществляется с использованием трёх операций: put, get и delete. Project Voldemort использует механизм MVCC при модификации данных.

Project Voldemort поддерживает шардинг и репликацию, а также несколько способов физической организации системы. Для распределения ключей по логическому кольцу узлов используется метод консистентного хэширования (consistent hashing, см., например, [34]). Шардинг осуществляется прозрачно для приложения, узлы могут быть добавлены или удалены в процессе работы, восстановление при сбоях также происходит автоматически. Project Voldemort использует асинхронную репликацию и поддерживает согласованность «в конечном счёте», конфликты разрешаются при чтении (read-repair). Также используется механизм направленной передачи (hinted handoff), который позволяет сохранить данные даже при выходе хранящих их узлов из строя, используя другие узлы. Project Voldemort может хранить данные в оперативной памяти и обеспечивать постоянство хранения с помощью одного из поддерживаемых механизмов, например, Berkeley DB [35]. Более подробно архитектура Project Voldemort описывается в [36].

4.2.2 DynamoDB

DynamoDB [37] – облачный сервис, представленный компанией Amazon в начале 2012 года [38]. Эта система является последователем таких технологий Amazon, как Dynamo [39] и SimpleDB. DynamoDB предоставляет пользователям быструю и масштабируемую систему управления данными, репликация и шардинг в которой осуществляются автоматически. Высокая производительность системы достигается за счёт использования твердотельных накопителей (SSD) и ряда других оптимизаций. Использование облачного сервиса снимает с пользователей необходимость в установке и администрировании серверов, позволяя оплачивать лишь потребляемые ресурсы, такие как трафик и объем хранимой информации.

Модель данных DynamoDB является достаточно гибкой и богатой для систем типа «ключ-значение». Данные хранятся в так называемых таблицах, обладающих первичным ключом (простым или составным) и набором атрибутов (заранее зафиксированной схемы нет, поддерживаются скалярные типы данных и множества). Для работы с данными используются операции поиска, вставки и удаления по первичному ключу, условные операции (например, обновить, если выполнено условие), атомарные модификации (например, увеличение значения атрибута на единицу) и поиск по неключевым атрибутам путём полного сканирования таблицы. Последняя операция делает эту систему еще ближе к документным СУБД, однако эффективного способа запрашивать данные по неключевым атрибутам DynamoDB не имеет. Желаемый уровень согласованности может быть указан при чтении данных («eventually consistent reads» или «strongly consistent reads»). Полностью согласованные чтения гарантированно возвращают

последнюю версию данных, однако это сказывается на производительности. Клиентские библиотеки для работы с DynamoDB доступны для большого числа языков программирования, включая Java, .NET, PHP, Perl, Python, Ruby и др.

4.2.3 Redis

Redis [40] – система управления данными типа «ключ-значение» с открытым исходным кодом, написанная на C и также поддерживающая достаточно богатую для таких систем модель данных. Значения могут содержать не только строки, но и множества, списки и другие структуры данных. Помимо обычных операций получения, сохранения и удаления данных по ключу, Redis поддерживает атомарные операции, такие как увеличение числа на единицу, добавление элемента в список и т. д. Кроме того, поддерживаются транзакции, содержащие группы операций и обладающие свойствами изолированности и атомарности, а также оптимистические блокировки.

Redis работает в оперативной памяти, за счёт чего достигается высокая производительность. Для обеспечения долговременного хранения могут применяться снимки данных в определённые моменты времени или постоянная запись операций модификации данных на диск; также возможна работа вообще без использования дисков.

Горизонтальная масштабируемость может быть достигнута с помощью разделения данных (логика реализуется на стороне клиента). Также поддерживается асинхронная репликация со схемой «ведущий-ведомый».

Клиентские библиотеки для работы с Redis доступны для большинства языков программирования, а сама эта система используется в таких крупных проектах, как Twitter, Instagram, Digg, Github, StackOverflow, Flickr и других [41].

4.2.4 Riak

Riak [14] – мощная система управления данными типа «ключ-значение» с открытым исходным кодом, написанная на Erlang. На архитектуру Riak оказала значительное влияние система Amazon Dynamo [39].

Организация кластера Riak похожа на Project Voldemort: также используется консистентное хэширование (consistent hashing) и направленная передача (hinted handoff). Такая архитектура обеспечивает надежность, децентрализацию и легкое добавление новых физических узлов. На каждом физическом узле (node) может работать несколько виртуальных узлов (vnode), что помогает при балансировке нагрузки; репликация выполняется асинхронно, количество реплик может быть гибко настроено (N). Riak позволяет указывать число реплик для чтения (R) и записи (W) при соответствующих операциях и, таким образом, варьировать уровень согласованности, однако атомарных операций не предусмотрено. Riak

использует вариант MVCC для реализации параллельного доступа. Конфликты обновлений могут разрешаться либо по принципу «последний выигрывает», либо на уровне приложения (в этом случае приложению возвращаются конфликтующие версии объектов). Поддерживаются триггеры (называемые «commit hooks»), позволяющие запускать функции на JavaScript или Erlang перед или после модификации объекта.

Ключи в Riak организуются в корзины («bucket»), что позволяет задавать такие параметры, как число реплик, список триггеров и метод разрешения конфликтов, на уровне корзины. Таким образом, объект однозначно идентифицируется парой (корзина, ключ). Объекты в Riak представляются в виде JSON [42] и могут иметь несколько полей данных. Кроме того, объект содержит метаданные, в том числе поддерживаются ссылки на другие объекты. Riak поддерживает вторичные индексы (возможность приписать объекту пару атрибут-значение для последующих запросов по ним) и MapReduce для более сложных запросов. Функциональность Riak делает эту систему близкой к документным СУБД, однако в Riak не поддерживаются запросы к полям объектов (единственный способ – MapReduce), поэтому обычно эту систему относят к классу систем «ключ-значение».

Подсистема хранения данных в Riak является подключаемым модулем, что позволяет использовать различные реализации, подходящие для разных задач. Взаимодействовать с Riak пользователь может с помощью REST-интерфейса или с помощью программного интерфейса, доступного для большинства языков программирования. Riak используется в достаточно большом количестве проектов [43].

4.2.5 Aerospike

Aerospike (ранее эта система называлась Citrusleaf) [44] интересна тем, что, являясь NoSQL-системой типа «ключ-значение», поддерживает оптимистические блокировки, атомарные операции, синхронную репликацию и мгновенную согласованность. При сбоях кластер Aerospike может работать в режиме устойчивости к разделению («partition tolerant mode», система продолжает работу в разделенном состоянии) или в режиме согласованности («high consistency mode», часть кластера может быть отключена, чтобы не допустить несогласованности данных) [45].

Система Aerospike оптимизирована для работы в оперативной памяти и на твердотельных носителях (Flash, SSD), при этом ключи всегда находятся в оперативной памяти. Для оптимизации сетевых взаимодействий применяются клиентские библиотеки, использующие сведения о состоянии кластера. Модель данных в Aerospike на самом верхнем уровне содержит пространства имён (namespaces), внутри которых содержатся множества (sets), которые хранят записи (records), обладающие уникальным ключом и типизированными атрибутами (bins). Подробно архитектура Aerospike описывается в [46].

4.2.6 Резюме

На момент написания данной работы список NoSQL-систем класса «ключ-значение» в [21] насчитывал более 30 систем, включая такие системы, как Scalaris, Tokyo Cabinet, GenieDB, LevelDB и др. Все эти системы обладают своими особенностями и нюансами и подходят для различных задач. При выборе конкретной системы приходится принимать во внимание множество факторов, таких как желаемый уровень согласованности данных, наличие атомарных операций, легкость масштабирования и администрирования, надежность, наличие клиентских библиотек для используемого языка программирования и т.д. В целом, достоинствами систем типа «ключ-значение» являются хорошая горизонтальная масштабируемость, простота и производительность, однако часто их моделей данных оказывается недостаточно для построения серьезных приложений, где возникает потребность, например, в поиске по сочетанию атрибутов, поддержка вложенных объектов, индексы на полях объектов и т.д. В этом случае нужно обратить внимание на системы с более сложной моделью данных, например, документные СУБД.

4.3 Документные СУБД

Документные СУБД предоставляют больше возможностей, чем системы типа «ключ-значение». Единицей хранения данных в таких системах является документ – некоторый объект, обладающий произвольным набором атрибутов (полей), который может быть представлен, например, в JSON [42]. Документные системы поддерживают поиск по полям документов, индексы, часто допускаются вложенные документы и массивы, а заранее предопределённой схемы данных, как правило, нет. В отличие от систем типа «ключ-значение», документные СУБД позволяют запрашивать коллекции документов на основании нескольких ограничений на атрибуты, могут осуществлять агрегатные запросы, сортировку результатов, поддерживают индексы на полях документов и т.д. Документные системы обычно не поддерживают семантику ACID, однако между собой они значительно различаются в плане поддержки согласованности данных, наличии атомарных операций, в способах контроля параллельного доступа к документам, а также во многих других аспектах.

4.3.1 MongoDB

MongoDB [12] – документная СУБД с открытым исходным кодом, написанная на C++ и разрабатываемая компанией 10gen. MongoDB обладает достаточно богатой функциональностью и является одной из самых популярных NoSQL-систем на данный момент [47].

MongoDB позволяет оперировать JSON-документами (хранимыми и передаваемыми в виде BSON – более компактного двоичного представления JSON), объединяемыми в коллекции, которые, в свою очередь, объединяются в базы данных. Каждый документ в коллекции должен содержать уникальный идентификатор (сгенерированный автоматически или пользователем), который не может изменяться после создания документа. Кроме идентификатора, документ также может содержать произвольный набор полей, которые могут содержать массивы и вложенные документы. Заранее предопределенной схемы данных нет: документы в одной коллекции могут содержать разные наборы полей.

Для работы с документами предусмотрены операции поиска, вставки, удаления и обновления документов. Для поиска документов в коллекции используется метод запросов по образцу, поддерживаются сортировка, проекция, просмотр результатов запроса с помощью курсора. Кроме того, поддерживается MapReduce, а также Aggregation Framework – способ формирования запроса из последовательных шагов, таких как агрегация, проекция, сортировка и т.д., что позволяет выполнять достаточно сложные аналитические запросы. Обновление документа может производиться либо полной заменой документа (с сохранением идентификатора), либо изменением полей существующего документа (в том числе добавление элемента в массив, увеличение числа и т.д.); при этом операция модификации одного документа всегда является атомарной (если обновление затрагивает несколько документов, то это уже не так). Кроме того, поддерживается атомарная операция «findAndModify», которая находит и изменяет документ, возвращая старую или новую версию. MongoDB использует блокировки для синхронизации параллельного доступа в пределах одного узла.

Для ускорения поиска документов поддерживается создание индексов на одном или нескольких полях документов в коллекции (реализованные с помощью B-деревьев), а также двумерные пространственные индексы. Существует возможность «подсказать» оптимизатору запросов, какой индекс использовать (hint), и проанализировать план выполнения (explain).

Масштабируемость в MongoDB достигается за счёт разделения документов из коллекции по узлам на основании выбранного ключа (shard key). Поддерживается асинхронная репликация в режиме «главный-подчиненный»: операции записи обрабатываются только главным узлом, а чтения могут осуществляться как с главного узла, так и с одного из подчиненных. Клиент может работать в разных режимах: неблокирующем (не дожидаясь подтверждения) или блокирующем (ожидая подтверждения от заданного количества узлов). Таким образом, MongoDB поддерживает различные модели согласованности в зависимости от того, разрешены ли чтения с вторичных узлов и от скольких узлов ожидаются подтверждения при записи. MongoDB может быть использована также и в качестве распределённой файловой системы благодаря функционалу GridFS.

Клиентские библиотеки для работы с MongoDB доступны для большого числа языков программирования, кроме того, поддерживается REST-интерфейс. Эта система используется в большом числе крупных компаний и проектов, среди которых SourceForge, Foursquare, The Guardian, Forbes, The New York Times и др. [48].

4.3.2 CouchDB

CouchDB [13] – проект Apache Software Foundation с открытым исходным кодом, реализованный на Erlang. CouchDB является распределённой документной СУБД, оперирующей JSON-документами.

Заранее предопределённой схемы данных в CouchDB, как и в MongoDB, не предусмотрено – документы могут содержать различные наборы полей (поддерживаются скалярные поля, массивы, вложенные документы и т.д.) и имеют уникальный идентификатор, а также номер ревизии (revision); документы организуются в базы данных. Отчёты и запросы к базам данных строятся с использованием MapReduce-представлений (views) – специальных функций на JavaScript, позволяющих задавать вид возвращаемых данных и выполнять агрегацию; эти функции помещаются в специальные документы (design documents). Запросы к представлениям позволяют задавать ограничения на возвращаемые данные, осуществлять сортировку, ограничивать количество возвращаемых результатов и т.д. Для представлений строятся индексы на основе B-деревьев, обновляемые при модификациях данных. Операции модификации на уровне документа обладают свойствами ACID, а читатели никогда не блокируются благодаря использованию MVCC; CouchDB поддерживает оптимистические блокировки при работе с документами. После каждой операции, изменяющей данные, происходит немедленный сброс данных на диск. Данные дописываются в конец файла, старые ревизии документов сохраняются, поэтому требуется периодически проводить сжатие (compaction) базы данных (в процессе сжатия система остается доступной для чтения и записи). Также поддерживаются JavaScript-функции для валидации данных и проверки прав доступа к ним при обновлениях.

CouchDB может масштабироваться только за счет репликации, выполняемой асинхронно. Поддерживается как схема репликации «ведущий-ведомый», так и «ведущий-ведущий». Кроме того, существует механизм условной (filtered) репликации, когда реплицируются только определённые документы. Каждый клиент видит согласованное состояние базы данных, однако эти состояния могут различаться для разных клиентов (усиленный вариант согласованности «в конечном счёте»). Когда один и тот же документ изменяется на разных узлах, возникает конфликт. При обнаружении конфликта одна из версий документа автоматически становится «победителем», а «проигравшая» версия сохраняется и может быть использована для разрешения конфликта.

Существует также проект CouchDB Lounge [49], предоставляющий возможности шардинга для CouchDB.

Работа с CouchDB осуществляется через REST-интерфейс, а клиентские библиотеки доступны для большого числа различных языков, в том числе Java, .NET, Python, PHP, Ruby и др. CouchDB используется в достаточно большом количестве проектов [50].

4.3.3 Couchbase Server

Couchbase Server [31] – проект, являющийся слиянием проектов Membase (система типа «ключ-значение», совместимая с Memcached) и системы CouchDB, рассмотренной ранее.

Couchbase Server может быть использован как в качестве системы управления данными типа «ключ-значение», совместимой с протоколом Memcached, так и в качестве документной СУБД, работающей с JSON-документами через REST-интерфейс. Документы могут содержать произвольный набор полей, имеют уникальный идентификатор и хранятся в «корзинах» (data bucket). Запросы осуществляют с помощью MapReduce-представлений (views) на JavaScript аналогично CouchDB. Представления строятся инкрементально и асинхронно, поэтому по умолчанию моделью согласованности является согласованность «в конечном счёте», однако на уровне операции можно указать, чтобы данные индексировались сразу же. Подсистема хранения данных также функционирует аналогично CouchDB – данные записываются в конец файла, требуется периодическое сжатие базы данных.

Важной особенностью Couchbase Server по сравнению с CouchDB является поддержка автоматического и прозрачного для приложения шардинга. Кроме того, Couchbase поддерживает два различных вида репликации – внутри кластера (intra-cluster) и межкластерную (inter-cluster, XDCR – Cross Datacenter Replication). Первый вид репликации осуществляется в пределах кластера, где узлы содержат как свои собственные данные, так и реплики других узлов, и поддерживает мгновенную согласованность на уровне документа – репликация в стиле Membase. Второй вид репликации предназначен для географически распределённых кластеров, соединённых с помощью WAN, и выполняется асинхронно, обеспечивая согласованность в «конечном счёте» между кластерами; разрешение конфликтов в этом случае осуществляется аналогично CouchDB – в кластерах выбирается один и тот же «победитель».

Couchbase Server является новой и активно развивающейся системой с богатыми возможностями. Наиболее актуальную и подробную документацию по Couchbase Server читатель может найти в [51].

4.3.4 Резюме

Документные СУБД имеют гибкую модель данных, которая в ряде случаев является более удобной, чем фиксированная схема, и лучше сочетается с

объектно-ориентированным программированием, сокращая прослойку между языком программирования и СУБД. Системы этого класса могут легко масштабироваться, хотя делают это немного по-разному. Способы построения запросов также различаются, но в целом поддерживаются достаточно сложные выборки, включающие ограничения на значения полей, агрегацию, сортировку и т.д. К вопросу согласованности данных документные системы также подходят по-разному, обычно позволяя в определенной степени варьировать их в зависимости от конфигурации и потребностей приложения. Кроме того, могут быть реализованы дополнительные механизмы, например, оптимистические блокировки, атомарные операции и т.д., что позволяет усилить гарантии согласованности данных для тех приложений, где это необходимо. Документные СУБД обычно поддерживают постоянство хранения данных, используя запись на жесткие диски или SSD-накопители, а надежность обеспечивается с помощью журналирования и репликации; индексы и часто используемые документы при этом обычно хранятся в оперативной памяти для быстрого доступа. Транзакционная семантика на уровне нескольких документов в таких системах обычно не поддерживается, единственной возможностью является реализация на уровне приложения. Тем не менее документные СУБД по функциональности постепенно приближаются к традиционным SQL-ориентированным СУБД.

На момент написания работы список документных СУБД в [21] насчитывал порядка 20 систем, включая такие системы, как Terrastore, RethinkDB, RavenDB и др.

4.4 Системы типа Google BigTable

Разработка Google BigTable [52] была начата в 2004 году для поддержки различных сервисов Google, таких как Google Earth, Google Maps, Google Analytics и др. BigTable базируется на Google File System (GFS, используется для хранения данных и журнала), Chubby (используется для координации и хранения некоторых метаданных) и других разработках компании и не распространяется за пределами Google, но возможность её использования предоставляется в рамках Google App Engine. BigTable проектировалась таким образом, чтобы легко масштабироваться на сотни и тысячи узлов и работать с петабайтами данных.

Таблица BigTable представляет собой отображение ключа ряда (row key), ключа столбца (column key) и временной метки (timestamp) в значение в виде строки. Ключ ряда и ключ столбца также являются обычными строками. Ключи рядов упорядочены в лексикографическом порядке, а столбцы объединены в семейства столбцов (column family), которые должны быть определены до использования, после чего в каждое семейство столбцы могут быть добавлены динамически. Семейства столбцов обычно хранят однотипные данные, и их число невелико (не более сотни), в то время как

столбцов в семействе может быть неограниченное количество. Каждая ячейка таблицы может содержать несколько версий данных, помеченных временными метками и упорядоченными по ним, текущее значение имеет наибольшую временную метку, поддерживается автоматическое удаление старых версий; ячейки также могут вообще не содержать данных. Таким образом, каждая строка таблицы (ряд) может содержать произвольное число атрибутов (столбцов), входящих в заранее определённые семейства.

Ряды таблицы разделяются по диапазонам ключей, формируя относительно небольшие по размеру сегменты («tablets» в терминологии BigTable), являющиеся единицами распределения при балансировке нагрузки. Кластер BigTable содержит один главный сервер (master) и сервера, непосредственно хранящие сегменты. Главный сервер отвечает за распределение сегментов по узлам, балансировку нагрузки, операции со схемой и т.д. Семейства столбцов являются единицами контроля прав доступа и параметров хранения. Данные хранятся по столбцам, а семейства столбцов, доступ к которым обычно осуществляется вместе, могут быть выделены в группы локальности (locality groups), что позволяет оптимизировать чтения. На уровне группы можно указать, например, чтобы данные её семейств столбцов постоянно находились в оперативной памяти и читались из неё, а также настроить сжатие данных. BigTable поддерживает асинхронную репликацию между кластерами, гарантируя при этом согласованность «в конечном счёте».

В BigTable предусмотрены операции для создания и удаления таблиц и семейств столбцов, изменения метаданных (например, прав доступа), записи и удаления значений, чтения определенных строк, просмотра подмножеств данных (например, столбцов из определённого семейства). Также поддерживаются атомарные операции над строками таблицы и исполнение скриптов для обработки данных. Клиентские библиотеки кэшируют метаданные о расположении сегментов и большую часть времени обращаются непосредственно к узлам, хранящим данные. Также BigTable может быть использована с MapReduce.

Подробности реализации BigTable читатель может найти в [52]. Успех BigTable положил начало новому семейству систем, применяющих схожие подходы для обеспечения масштабируемости и высокой производительности.

4.4.1 HBase

HBase [16] – проект с открытым исходным кодом на Java, разрабатываемый Apache Software Foundation. HBase следует принципам BigTable и использует Apache Hadoop. Вместо GFS в HBase используется HDFS (Hadoop Distributed File System), но могут быть использованы и другие файловые системы. Также поддерживается Hadoop MapReduce, а роль сервиса Chubby в HBase выполняет Apache ZooKeeper.

Архитектура и функциональность HBase во многом соответствует BigTable (описанной в [52]), хотя имеются и некоторые отличия. Например, HBase

поддерживает несколько главных (master) серверов, чтобы повысить надежность системы. В HBase не поддерживается концепция групп локальности (locality groups), вся конфигурация выполняется на уровне семейств столбцов. Как и BigTable, HBase не поддерживает семантику ACID в полном объеме, однако определённые свойства, усиливающие гарантии согласованности, обеспечиваются (см. [53]). HBase не поддерживает вторичные индексы: записи могут быть запрошены только с помощью первичного ключа или сканирования таблицы. Индексы, тем не менее, могут быть построены вручную с помощью дополнительных таблиц.

Работа с HBase может осуществляться через API для Java, REST-интерфейс, а также с помощью Avro и Thrift. HBase используется в крупных и высоконагруженных приложениях и проектах, таких как Facebook (сервис Facebook Messages) и Twitter (для поддержки MapReduce, поиска по людям и других задач).

4.4.2 Cassandra

Система Cassandra [17] была разработана и использовалась в Facebook. В её основе лежат идеи Google BigTable [52] и Amazon Dynamo [39]. В настоящее время Cassandra является проектом с открытым исходным кодом (на Java), поддерживаемым Apache Software Foundation.

По организации модели данных Cassandra схожа с BigTable и HBase, однако терминология и детали несколько различаются. База данных в Cassandra называется «пространством ключей» (keyspace) и содержит семейства столбцов (column family), которые являются аналогом таблиц и служат контейнерами для строк (рядов, rows), идентифицируемых уникальными ключами (row key). Строки состоят из столбцов (column) или супер-столбцов (super column). Столбец является минимальной единицей данных в Cassandra и состоит из имени, значения и временной метки (все эти поля предоставляются клиентом), хранится только последняя версия данных (в противоположность BigTable и HBase). Супер-столбцы, в свою очередь, содержат внутри себя столбцы, добавляя тем самым еще один уровень вложенности. Кроме того, поддерживаются специальные столбцы, такие как счётчики или столбцы с указанным временем жизни (TTL). Разным строкам необязательно должен соответствовать один и тот же набор столбцов или супер-столбцов. Семейства столбцов хранятся в отдельных файлах с сортировкой по ключам строк и должны содержать столбцы, доступ к которым в запросах предполагается осуществлять вместе.

Для работы с данными Cassandra поддерживает SQL-подобный язык CQL (Cassandra Query Language), кроме того, есть поддержка Hadoop MapReduce. Для ускорения запросов поддерживается создание вторичных индексов. Операции модификации данных являются атомарными на уровне одной строки таблицы, постоянство хранения обеспечивается с помощью записи в журнал, поддерживается сжатие данных. Cassandra позволяет гибко

варьировать уровень согласованности данных на уровне операций. Конфликты разрешаются на основании временных меток (выигрывает последняя версия).

Cassandra проектировалась так, чтобы обеспечить хорошую масштабируемость и надежность на большом количестве недорогих (и ненадежным) машин. В отличие от BigTable и HBase, в кластере Cassandra нет выделенных узлов, все они равноправны и выполняют одни и те же функции. Для распределения данных по узлам применяется консистентное хэширование (consistent hashing) и направленная передача (hinted handoff), новые узлы могут быть легко добавлены в кластер, а обнаружение сбоев и восстановление происходят автоматически. Разделение строк по узлам может осуществляться как случайным образом, так и с сохранением порядка. Репликация поддерживается как в пределах кластера, так и между географически распределёнными кластерами.

Клиентские библиотеки для работы с Cassandra доступны для большинства языков программирования (основаны на Thrift). Эта система используется во многих проектах с высокой нагрузкой [54].

4.4.3 Резюме

Рассмотренные системы во многом следуют архитектуре и подходам, применённым в BigTable. Эти системы созданы для поддержки высоконагруженных приложений и работы на больших кластерах недорогих машин, что достигается за счёт несколько более сложной модели данных, чем документная модель: требуется внимательное проектирование семейств столбцов и выбор ключей строк, а реализация некоторых функций (например, вторичные индексы в HBase) перекладывается на разработчика.

4.5 Другие системы

Как было отмечено, иногда под термином «NoSQL» понимают также вообще все системы управления данными, не являющиеся реляционными (SQL-ориентированными). Многие из этих систем появились еще до зарождения и популяризации NoSQL-движения, а также часто поддерживают ACID-транзакции и не всегда являются распределёнными, что нетипично для новых систем. Тем не менее, перечислим некоторые классы систем, не вошедшие в данный обзор, но иногда относимые к NoSQL: объектно-ориентированные СУБД, графовые системы, XML-ориентированные СУБД, многомерные системы и др.

4.6 Резюме

NoSQL-системы очень разнообразны: различные модели данных, языки запросов, поддерживаемые уровни согласованности, организация кластеров,

функциональность, производительность и многие другие аспекты отличают эти системы одну от другой. Общими чертами NoSQL-систем являются отказ от SQL и снижение гарантий согласованности данных ради достижения большей производительности и масштабируемости. Выбор конкретного NoSQL-решения зависит от решаемой задачи, однако для задач, где требуется транзакционная семантика, эти системы подходят плохо. Для решения проблем масштабируемости без отказа от ACID-транзакций и языка SQL был разработан ряд новых систем и подходов, которые можно объединить под общим названием «NewSQL».

5. NewSQL-системы

Термин «NewSQL» был впервые применен Мэттью Эслеттом (Matthew Aslett) в 2011 году для обозначения достаточного нового класса распределённых масштабируемых высокопроизводительных систем с поддержкой SQL [55] (стоит отметить, что сами эти системы появились раньше и продолжают активно развиваться). Также часто использовался и используется термин «Scalable SQL», чтобы подчеркнуть главное отличие новых систем от классических «односерверных» SQL-ориентированных СУБД: обеспечение хорошей горизонтальной масштабируемости за счет распределённой архитектуры при сохранении SQL и ACID-транзакций.

Потребность в масштабируемых системах с поддержкой SQL (и, главным образом, с поддержкой транзакций) обусловлена фактом наличия приложений (например, в области финансов, бизнеса, документооборота и т.д.), которые не могут мириться со слабыми моделями согласованности, предлагаемыми NoSQL-системами. Кроме того, поддержка SQL позволяет легче переносить существующие приложения, работающие с классическими SQL-ориентированными СУБД, и использовать опыт, накопленный за несколько десятилетий использования SQL в OLTP-системах (Online Transaction Processing).

Высокая производительность в NewSQL-системах достигается не только за счет горизонтального масштабирования, но и за счет оптимизации работы каждого отдельного узла: избежание блокировок (например, использование MVCC), применение оперативной памяти или твердотельных дисков в качестве основного хранилища данных, оптимиазация работы с индексами и т.д.

NewSQL-системы обычно делят на следующие категории (см., например, [56], [57]):

1) Системы с новыми архитектурами

2) Подсистемы хранения данных для MySQL

3) Решения для масштабирования поверх традиционных систем

Далее будут рассматриваться системы и подходы, которые могут быть отнесены к каждой из приведенных категорий.

5.1 Системы с новыми архитектурами

Многие хорошо известные традиционные СУБД, такие как MySQL, Oracle и Microsoft SQL Server включают некоторую функциональность, помогающую в какой-то мере масштабировать эти СУБД (например, за счет репликации), сохраняя при этом традиционную организацию самой СУБД. Для преодоления недостатков традиционных SQL-ориентированных систем были предприняты попытки создания новых решений с нуля, изначально закладывая в них возможности для горизонтального масштабирования. Эти системы, несмотря на поддержку SQL и ACID-транзакций, очень отличаются от традиционных как в плане поддерживаемой функциональности, так и в нюансах проектирования схемы и доступа к данным. В ряде приложений, где требуется обработка большого потока коротких транзакций, эти системы могут обеспечить гораздо более высокую производительность и масштабируемость.

5.1.1 VoltDB

VoltDB [58] – высокопроизводительная OLTP-система, разработанная при участии Майкла Стоунбрейкера (Michael Stonebraker) и реализующая архитектуру академического проекта H-Store [59]. В [60] Стоунбрейкер анализирует теорему CAP и обосновывает выбор поддержки согласованности данных в противоположность NoSQL-системам.

Архитектура VoltDB кардинально отличается от классической архитектуры SQL-ориентированных СУБД. Работа с данными производится с помощью хранимых процедур на языке Java, содержащих SQL-запросы. Хранимая процедура (все SQL-операторы в ней) исполняется как единая ACID-транзакция. И данные, и связанные с ними обработки распределяются по узлам кластера. Таблицы могут быть разбиты на разделы на основании значений колонки, а также могут быть реплицированы. Все данные хранятся в ОП, что позволяет отказаться от дисковых обменов и управления буферами, однако при необходимости данные можно скопировать на жесткий диск в качестве резервной копии. Поток VoltDB исполняется последовательно на каждом процессорном ядре и работает со своей частью данных; запросы к этим данным обрабатываются в порядке очереди. Если же транзакция требует доступа к нескольким разделам, то один узел координирует работу и составляет планы исполнения для других узлов. Перед фиксацией транзакции все данные синхронно фиксируются на всех узлах, содержащих реплики данных. Поддерживается также асинхронная репликация между кластерами.

При проектировании базы данных пользователь создает схему и хранимые процедуры, необходимые для приложений, а затем использует компилятор

VoltDB, чтобы создать экземпляр базы данных, готовый к размещению на кластере.

Подход, примененный в VoltDB, позволяет добиться высокой доступности данных, производительности и хорошей масштабируемости в приложениях, работающих с большим количеством коротких и «узких» транзакций. «Широкие» и долгие транзакции также возможны, но неэффективны. Кроме того, поддерживается не вся функциональность языка SQL, а проектирование схемы и хранимых процедур часто является нетривиальной задачей.

5.1.2 Clustrix

Clustrix [61], также являясь представителем NewSQL-движения, использует несколько иные подходы. Эта система предоставляется в качестве сервиса или в виде готовых для размещения аппаратных решений. Clustrix обладает совместимостью с MySQL, поэтому перенос приложений с этой СУБД не требует практически никаких усилий. В отличие от VoltDB, Clustrix использует SSD-накопители для хранения данных.

Clustrix распределяет данные по узлам и балансирует нагрузку автоматически, уменьшая, тем самым, сложность проектирования и администрирования кластера. Поддерживается репликация и автоматическое восстановление при сбоях. Запросы к этой системе обрабатываются путём деления их на фрагменты и отправки на узлы, содержащие данные для этих фрагментов, после чего происходит объединение результатов.

Clustrix обладает хорошей горизонтальной масштабируемостью, обеспечивает высокую доступность данных и легкость в использовании, позволяя заменить MySQL в уже существующих приложениях.

5.1.3 Резюме

NewSQL-системы с новыми архитектурами принципиально отличаются от классических SQL-ориентированных СУБД изначальной поддержкой распределённой архитектуры. Эти системы способны хорошо масштабироваться горизонтально и обеспечивать высокую производительность при работе с определёнными типами транзакций (затрагивающими небольшое число узлов). В целом на «небольших» транзакциях эти системы приближаются по масштабируемости и производительности к NoSQL-решениям, сохраняя при этом поддержку ACID и SQL для приложений с высокими требованиями к согласованности данных. В ряде систем проектирование схемы базы данных с учётом разделения данных по узлам может вносить дополнительные сложности.

Кроме рассмотренных примеров, к NewSQL-системам с новыми архитектурами можно отнести такие системы, как NuoDB, Google Spanner и др.

5.2 Подсистемы хранения данных для MySQL

Отличительной особенностью популярной SQL-ориентированной СУБД MySQL [62] является поддержка подключаемых подсистем хранения данных (см., например, [63]). Подсистема хранения (storage engine) реализует операции над таблицами, отделяя непосредственную работу с данными от ядра СУБД. Таким образом, становится возможным использовать более масштабируемую подсистему хранения данных с поддержкой интерфейса MySQL.

Например, NDB (MySQL Cluster) [64] позволяет организовать распределённое хранение данных, обеспечивая шардинг и репликацию. Поддерживаются три типа узлов: data node (обеспечивают распределённое хранение данных), management node (для управления кластером) и SQL node (позволяет манипулировать данными в NDB-кластере с помощью SQL). Последний тип узла (являющийся процессом MySQL Server) не является обязательным, так как доступ к данным может осуществляться как с помощью SQL, так и через NoSQL-интерфейсы.

Другим примером производительной подсистемы хранения данных для MySQL является TokuDB [65]. TokuDB оптимизирует операции с индексами, скорость записи данных и работу с SSD-накопителями, позволяя справляться с большими объемами данных, чем это возможно со стандартными MyISAM и InnoDB.

Подключаемые подсистемы хранения для MySQL позволяют оптимизировать работу с данными и помочь в масштабировании, сохраняя при этом возможность работать с интерфейсом MySQL и использовать ACID-транзакции (если они поддерживаются подсистемой хранения). Также к этой категории можно отнести такие системы, как XtraDB, ScaleDB, GenieDB и др.

5.3 Решения для масштабирования поверх традиционных СУБД

Последняя категория содержит промежуточное программное обеспечение для прозрачного шардинга и репликации между несколькими серверами под управлением традиционных SQL-ориентированных СУБД. Применение такого ПО позволяет не переписывать существующие СУБД и сохранить полную совместимость с существующими приложениями.

В качестве примера реализации такого подхода можно привести dbShards [66]. dbShards позволяет осуществлять разделение данных между несколькими экземплярами СУБД прозрачно для приложений. При этом поддерживаются запросы и транзакции, затрагивающие несколько разделов, а также репликация и автоматическое восстановление. Для работы с dbShards можно использовать JDBC, а также MySQL-совместимый драйвер, позволяющий работать с dbShards из C, Perl, PHP, Python и других языков.

Другой системой этой категории является Data Traffic Manager от ScaleBase [67]. Data Traffic Manager позволяет масштабировать MySQL прозрачно для приложений, обеспечивая, в том числе, возможность выполнения запросов, затрагивающих несколько экземпляров СУБД.

Таким образом, использование промежуточного слоя ПО позволяет прозрачно разделять данные между несколькими экземплярами традиционных SQL-ориентированных СУБД, снимая необходимость в реализации шардинга на стороне приложения. Производительность таких решений в целом ниже, чем, например, систем с новыми архитектурами, однако сохраняется полная совместимость с существующими приложениями и инструментами.

5.4 Резюме

Появление NewSQL-систем отражает потребность в масштабируемых СУБД с поддержкой SQL и ACID-транзакций. Эти системы стремятся достичь производительности, сравнимой с NoSQL-решениями, не ослабляя при этом гарантий согласованности данных. Такое возможно, к сожалению, не всегда: распределённые транзакции, затрагивающие много узлов системы трудно выполнять эффективно, учитывая требуемое активное сетевое взаимодействие между этими узлами. Синхронный режим репликации, требуемый для обеспечения согласованности данных, также повышает задержки, однако без репликации невозможно гарантировать отказоустойчивость системы и высокую доступность данных. В целом, если транзакции таковы, что их можно целиком выполнять на одном узле, то возможно добиться производительности, сравнимой с NoSQL-системами. NewSQL-системы с новыми архитектурами, кроме того, стремятся повысить производительность за счет оптимизации скорости доступа к данным (например, работая целиком в ОП и избегая блокировок). Существует также ряд решений для масштабирования классических SQL-ориентированных СУБД, помогающих масштабировать существующие приложения без значительных изменений логики работы с данными.

Главным преимуществом NewSQL-систем является, безусловно, поддержка ACID-транзакций, что делает возможным их применение там, где NoSQL-решения неприменимы. Кроме того, поддержка SQL позволяет использовать накопленный опыт и инструментарий для работы с базами данных, однако создание схемы и оптимизация SQL-запросов для их выполнения на кластере могут усложнить разработку приложений.

6. Заключение

В данной работе была дана общая характеристика ряда новых направлений в области управления данными. Мотивацией к созданию этих систем послужило, в том числе, активное развитие Web, что привело к появлению большого числа приложений с гигантской нагрузкой и огромными объемами

данных. NoSQL-системы фокусируются в основном на том, чтобы обеспечить требуемую масштабируемость (и отказоустойчивость) даже за счёт снижения гарантий согласованности данных и отказа от привычной транзакционной семантики. Модели данных, поддерживаемые NoSQL-системами, в целом проще, чем реляционная модель, а жестко определённой схемы данных и ограничений целостности, как правило, нет. При использовании NoSQL-систем разработка приложений часто упрощается за счёт более простых и гибких моделей данных (например, документной) и меньшего «impedance mismatch», то есть несоответствия объектно-ориентированной модели языка программирования и модели данных используемой СУБД (см., например, [68]). К сожалению, в целом NoSQL-системы плохо подходят для задач, где требуется транзакционная семантика. Для решения этой проблемы был разработан ряд новых систем и подходов, сохраняющих ACID-транзакции и поддержку языка SQL, но позволяющих обеспечить требуемую для современных приложений масштабируемость в ряде задач.

Большое разнообразие систем и подходов обусловлено общей тенденцией к специализации в области СУБД (см., например, [1]): каждая система управления данными приспособлена для решения определённого класса задач. Таким образом, выбор конкретных решений обусловлен спецификой решаемой задачи: предполагаемая нагрузка, соотношение интенсивности чтений и записи, вид хранимых данных и типы запросов к ним, желаемый уровень согласованности, требования к надежности, наличие клиентских библиотек для выбранного языка и т.д.

7. Список литературы

[1]. M. Stonebraker и U. Çetintemel, «"One Size Fits All": An Idea Whose Time Has Come and Gone,» в *ICDE ’05: Proceedings of the 21st International Conference on Data Engineering*, Washington, 2005.

[2]. W. Vogels, «Eventually Consistent,» *ACM Queue,* т. 6, № 6, 2008.

[3]. H. Wada, A. Fekete, L. Zhaoy, K. Lee и A. Liu, «Data Consistency Properties and the Tradeoffs in Commercial Cloud Storages: the Consumers’ Perspective,» в *Conference on Innovative Data Systems Research*, 2011.

[4]. R. Baldoni и M. Raynal, «Fundamentals of Distributed Computing - A Practical Tour of Vector Clock Systems,» 2002. [В Интернете]. URL: http://net.pku.edu.cn/~course/cs501/2008/reading/a_tour_vc.html. [Дата обращения: 8 июня 2013].

[5]. T. B. Douglas, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer и C. H. Hauser, «Managing update conflicts in Bayou, a weakly connected replicated storage system,» в *SOSP '95 Proceedings of the fifteenth ACM symposium on Operating systems principles*, New York, NY, USA, 1995.

[6]. D. Merriman, «On Distributed Consistency,» 26 марта 2010. [В Интернете]. URL: http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1. [Дата обращения: 8 июня 2013].

[7]. E. Brewer, «Towards Robust Distributed Systems,» в *ACM Symposium on the Principles of Distributed Computing*, Portland, Oregon, 2000.

[8]. S. Gilbert и N. Linch, «Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,» *ACM SIGACT News,* т. 33, № 2, pp. 51-59, 2002.
[9]. D. Abadi, «Problems with CAP, and Yahoo's little known NoSQL system,» 2010. [В Интернете]. URL: http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html. [Дата обращения: 8 июня 2013].
[10]. J. Dean и S. Ghemawat, «MapReduce: Simplified Data Processing on Large Clusters,» в *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004.
[11]. «Apache Hadoop,» [В Интернете]. URL: http://hadoop.apache.org/. [Дата обращения: 8 июня 2013].
[12]. «MongoDB,» 10gen, [В Интернете]. URL: http://www.mongodb.org/. [Дата обращения: 8 июня 2013].
[13]. «Apache CouchDB,» [В Интернете]. URL: http://couchdb.apache.org/. [Дата обращения: 8 июня 2013].
[14]. «Riak,» Basho, [В Интернете]. URL: http://basho.com/riak/. [Дата обращения: 8 июня 2013].
[15]. T. White, Hadoop: The Definitive Guide, O'Reilly Media, 2009.
[16]. «Apache HBase,» [В Интернете]. URL: http://hbase.apache.org/. [Дата обращения: 8 июня 2013].
[17]. «Apache Cassandra,» [В Интернете]. URL: http://cassandra.apache.org/. [Дата обращения: 8 июня 2013].
[18]. «Apache Pig,» [В Интернете]. URL: http://pig.apache.org/. [Дата обращения: 2 июня 2013].
[19]. C. Strozzi, «NoSQL: A Relational Database Management System,» [В Интернете]. URL: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page. [Дата обращения: 8 июня 2013].
[20]. J. Gray, «The Transaction Concept: Virtues and Limitations,» в *Seventh International Conference on Very Large Databases*, 1981.
[21]. «NOSQL Databases,» [В Интернете]. URL: http://nosql-database.org/. [Дата обращения: 8 июня 2013].
[22]. A. Wiggins, «SQL Databases Don't Scale,» 2009. [В Интернете]. URL: http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/. [Дата обращения: 8 июня 2013].
[23]. D. Obasanjo, «Building scalable databases: Denormalization, the NoSQL movement and Digg,» 2009. [В Интернете]. URL: http://www.25hoursaday.com/weblog/2009/09/10/BuildingScalableDatabasesDenormalizationTheNoSQLMovementAndDigg.aspx. [Дата обращения: 8 июня 2013].
[24]. B. A. Philip, V. Hadzilacos и N. Goodman, «Distributed Recovery,» в *Concurrency Control and Recovery in Database Systems*, Addison Wesley Publishing Company, 1987, pp. 240-264.
[25]. R. Cattell, «Scalable SQL and NoSQL Data Stores,» 2011. [В Интернете]. URL: http://www.cattell.net/datastores/Datastores.pdf. [Дата обращения: 8 июня 2013].
[26]. A. Lith и J. Mattsson, Investigating storage solutions for large data: A comparison of well performing and scalable data storage, Goteborg, Sweden: Chalmers University Of Technology, Department of Computer Science and Engineering, 2010.
[27]. C. Strauch, «NoSQL Databases,» 2011. [В Интернете]. URL: http://www.christof-strauch.de/nosqldbs.pdf. [Дата обращения: 8 июня 2013].

[28]. P. A. Bernstein и N. Goodman, «Concurrency Control in Distributed Database Systems,» *ACM Computing Surveys,* т. 13, № 2, pp. 185-221, 1981.
[29]. «memcached - a distributed memory object caching system,» [В Интернете]. URL: http://memcached.org/. [Дата обращения: 8 июня 2013].
[30]. «memcachedb - A distributed key-value storage system designed for persistent,» [В Интернете]. URL: http://memcachedb.org/. [Дата обращения: 8 июня 2013].
[31]. «Couchbase Server,» [В Интернете]. URL: http://www.couchbase.com/. [Дата обращения: 8 июня 2013].
[32]. «Project Voldemort,» [В Интернете]. URL: http://www.project-voldemort.com/. [Дата обращения: 8 июня 2013].
[33]. J. Kreps, «Project Voldemort: Scaling Simple Storage at LinkedIn,» 20 мая 2009. [В Интернете]. URL: http://blog.linkedin.com/2009/03/20/project-voldemort-scaling-simple-storage-at-linkedin/. [Дата обращения: 8 июня 2013].
[34]. D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins и Y. Yerushalmi, «Web Caching with Consistent Hashing,» MIT Laboratory for Computer Science, 1999. [В Интернете]. URL: http://www8.org/w8-papers/2a-webserver/caching/paper2.html. [Дата обращения: 8 июня 2013].
[35]. «Oracle Berkeley DB,» Oracle, [В Интернете]. URL: http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html. [Дата обращения: 8 июня 2013].
[36]. «Project Voldemort Design,» [В Интернете]. URL: http://www.project-voldemort.com/voldemort/design.html. [Дата обращения: 8 июня 2013].
[37]. «Amazon DynamoDB,» Amazon, [В Интернете]. URL: http://aws.amazon.com/dynamodb/. [Дата обращения: 8 июня 2013].
[38]. W. Vogels, «Amazon DynamoDB – a Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications,» 18 января 2012. [В Интернете]. URL: http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html. [Дата обращения: 8 июня 2013].
[39]. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall и W. Vogels, «Dynamo: Amazon’s Highly Available Key-value Store,» в *21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, 2007.
[40]. «Redis,» [В Интернете]. URL: http://redis.io/. [Дата обращения: 8 июня 2013].
[41]. «Who's using Redis?,» [В Интернете]. URL: http://redis.io/topics/whos-using-redis. [Дата обращения: 8 июня 2013].
[42]. «Introducing JSON,» [В Интернете]. URL: http://json.org/. [Дата обращения: 8 июня 2013].
[43]. «Riak Users,» [В Интернете]. URL: http://basho.com/riak-users/. [Дата обращения: 8 июня 2013].
[44]. «NoSQL Database, In-Memory or Flash Optimized and Web Scale - Aerospike,» [В Интернете]. URL: http://www.aerospike.com/. [Дата обращения: 8 июня 2013].
[45]. «Aerospike - Acid Compliant Database for Mission-Critical Applications,» [В Интернете]. URL: http://www.aerospike.com/performance/acid-compliance/. [Дата обращения: 8 июня 2013].
[46]. V. Srinivasan и B. Bulkowski, «Citrusleaf: A Real-Time NoSQL DB which Preserves ACID,» в *Very Large Databases (VLDB)*, 2010.
[47]. «DB-Engines Ranking,» [В Интернете]. URL: http://db-engines.com/en/ranking. [Дата обращения: 8 июня 2013].

[48]. «MongoDB Production Deployments,» [В Интернете]. URL: http://www.mongodb.org/about/production-deployments/. [Дата обращения: 8 июня 2013].
[49]. «CouchDB Lounge,» [В Интернете]. URL: http://tilgovi.github.com/couchdb-lounge/. [Дата обращения: 8 июня 2013].
[50]. «CouchDB in the Wild,» [В Интернете]. URL: http://wiki.apache.org/couchdb/CouchDB_in_the_wild. [Дата обращения: 8 июня 2013].
[51]. Couchbase, «Learn about Couchbase Server,» [В Интернете]. URL: http://www.couchbase.com/learn. [Дата обращения: 8 июня 2013].
[52]. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra и A. Fikes, «Bigtable: A Distributed Storage System for Structured Data,» в *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, Seattle, WA, 2006.
[53]. «Apache HBase ACID Properties,» [В Интернете]. URL: http://hbase.apache.org/acid-semantics.html. [Дата обращения: 8 июня 2013].
[54]. «Cassandra Users,» [В Интернете]. URL: http://www.datastax.com/cassandrausers. [Дата обращения: 8 июня 2013].
[55]. 451 Research, «NoSQL, NewSQL and Beyond: The drivers and use cases for database alternatives,» 2011.
[56]. P. Campaniello, «The NewSQL Market Breakdown,» 2011. [В Интернете]. URL: http://www.scalebase.com/the-story-of-newsql/. [Дата обращения: 4 июня 2013].
[57]. P. Venkatesh, «NewSQL - The New Way to Handle Big Data,» 2012. [В Интернете]. URL: http://www.linuxforu.com/2012/01/newsql-handle-big-data/. [Дата обращения: 4 июня 2013].
[58]. «VoltDB,» [В Интернете]. URL: http://voltdb.com. [Дата обращения: 8 июня 2013].
[59]. «H-Store,» [В Интернете]. URL: http://hstore.cs.brown.edu/. [Дата обращения: 8 июня 2013].
[60]. M. Stonebraker, «Errors in Database Systems, Eventual Consistency, and the CAP Theorem,» 2010. [В Интернете]. URL: http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext. [Дата обращения: 4 июня 2013].
[61]. «Clustrix,» [В Интернете]. URL: http://www.clustrix.com/. [Дата обращения: 5 июня 2013].
[62]. «MySQL,» [В Интернете]. URL: http://www.mysql.com/. [Дата обращения: 4 июня 2013].
[63]. S. K. Cabral и K. Murphy, MySQL Administrator's Bible, Wiley, 2009.
[64]. «MySQL Cluster,» [В Интернете]. URL: http://www.mysql.com/products/cluster/. [Дата обращения: 4 июня 2013].
[65]. «TokuDB for MySQL,» [В Интернете]. URL: http://www.tokutek.com/products/tokudb-for-mysql/. [Дата обращения: 4 июня 2013].
[66]. «dbShards,» [В Интернете]. URL: http://www.dbshards.com. [Дата обращения: 4 июня 2013].
[67]. «ScaleBase,» [В Интернете]. URL: http://www.scalebase.com/. [Дата обращения: 4 июня 2013].
[68]. T. Neward, «The Vietnam of Computer Science,» 26 июня 2006. [В Интернете]. URL: http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx. [Дата обращения: 8 июня 2013].

Modern Data Management Systems

S. D. Kuznetsov
ISP RAS, Moscow, Russia
kuzloc@ispras.ru
A. V. Poskonin
MSU, Moscow, Russia
aposk@yandex.ru

Abstract. Many modern applications (such as large-scale Web-sites, social networks, research projects, business analytics, etc.) have to deal with very large data volumes (also referred to as “big data”) and high read/write loads. These applications require underlying data management systems to scale well in order to accommodate data growth and increasing workloads. High throughput, low latencies and data availability are also very important, as well as data consistency guarantees. Traditional SQL-oriented DBMSs, despite their popularity, ACID transactions and rich features, do not scale well and thus are not suitable in certain cases. A number of new data management systems and approaches have emerged over the last decade intended to resolve scalability issues. This paper reviews several classes of such systems and key problems they are able to solve. A large variety of systems and approaches due to the general trend toward specialization in the field of SMS: every data management system has been adapted to solve a certain class of problems. Thus, the selection of specific solutions due to the specific problem to be solved: the expected load, the intensity ratio of read and write, the form of data storage and query types, the desired level of consistency, reliability requirements, the availability of client libraries for the selected language, etc.

Keywords: MapReduce; NoSQL; non-relational data models; scalability; data consistency; NewSQL.

References

[1]. M. Stonebraker и U. Çetintemel, «"One Size Fits All": An Idea Whose Time Has Come and Gone,» в ICDE ’05: Proceedings of the 21st International Conference on Data Engineering, Washington, 2005.
[2]. W. Vogels, «Eventually Consistent,» ACM Queue, т. 6, № 6, 2008.
[3]. H. Wada, A. Fekete, L. Zhaoy, K. Lee и A. Liu, «Data Consistency Properties and the Tradeoffs in Commercial Cloud Storages: the Consumers’ Perspective,» в Conference on Innovative Data Systems Research, 2011.
[4]. R. Baldoni и M. Raynal, «Fundamentals of Distributed Computing - A Practical Tour of Vector Clock Systems,» 2002. URL: http://net.pku.edu.cn/~course/cs501/2008/reading/a_tour_vc.html.
[5]. T. B. Douglas, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer и C. H. Hauser, «Managing update conflicts in Bayou, a weakly connected replicated storage system,» в SOSP '95 Proceedings of the fifteenth ACM symposium on Operating systems principles, New York, NY, USA, 1995.
[6]. D. Merriman, «On Distributed Consistency,» 26 марта 2010. URL: http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1

[7]. E. Brewer, «Towards Robust Distributed Systems,» в ACM Symposium on the Principles of Distributed Computing, Portland, Oregon, 2000.
[8]. S. Gilbert и N. Linch, «Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,» ACM SIGACT News, т. 33, № 2, pp. 51-59, 2002.
[9]. D. Abadi, «Problems with CAP, and Yahoo's little known NoSQL system,» 2010. URL: http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html.
[10]. J. Dean и S. Ghemawat, «MapReduce: Simplified Data Processing on Large Clusters,» в OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, 2004.
[11]. «Apache Hadoop,» URL: http://hadoop.apache.org/.
[12]. «MongoDB,» 10gen, URL: http://www.mongodb.org/.
[13]. «Apache CouchDB,» URL: http://couchdb.apache.org/.
[14]. «Riak,» Basho, URL: http://basho.com/riak/.
[15]. T. White, Hadoop: The Definitive Guide, O'Reilly Media, 2009.
[16]. «Apache HBase,» URL: http://hbase.apache.org/.
[17]. «Apache Cassandra,» URL: http://cassandra.apache.org/.
[18]. «Apache Pig,» URL: http://pig.apache.org/.
[19]. C. Strozzi, «NoSQL: A Relational Database Management System,» URL: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page.
[20]. J. Gray, «The Transaction Concept: Virtues and Limitations,» в Seventh International Conference on Very Large Databases, 1981.
[21]. «NOSQL Databases,» URL: http://nosql-database.org/.
[22]. A. Wiggins, «SQL Databases Don't Scale,» 2009. URL: http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/.
[23]. D. Obasanjo, «Building scalable databases: Denormalization, the NoSQL movement and Digg,» 2009. URL: http://www.25hoursaday.com/weblog/2009/09/10/BuildingScalableDatabasesDenormalizationTheNoSQLMovementAndDigg.aspx.
[24]. B. A. Philip, V. Hadzilacos и N. Goodman, «Distributed Recovery,» в Concurrency Control and Recovery in Database Systems, Addison Wesley Publishing Company, 1987, pp. 240-264.
[25]. R. Cattell, «Scalable SQL and NoSQL Data Stores,» 2011. URL: http://www.cattell.net/datastores/Datastores.pdf.
[26]. A. Lith и J. Mattsson, Investigating storage solutions for large data: A comparison of well performing and scalable data storage, Goteborg, Sweden: Chalmers University Of Technology, Department of Computer Science and Engineering, 2010.
[27]. C. Strauch, «NoSQL Databases,» 2011. URL: http://www.christof-strauch.de/nosqldbs.pdf.
[28]. P. A. Bernstein и N. Goodman, «Concurrency Control in Distributed Database Systems,» ACM Computing Surveys, т. 13, № 2, pp. 185-221, 1981.
[29]. «memcached - a distributed memory object caching system,» URL: http://memcached.org/.
[30]. «memcachedb - A distributed key-value storage system designed for persistent,» URL: http://memcachedb.org/.
[31]. «Couchbase Server,» URL: http://www.couchbase.com/.
[32]. «Project Voldemort,» URL: http://www.project-voldemort.com/.

[33]. J. Kreps, «Project Voldemort: Scaling Simple Storage at LinkedIn,» 20 мая 2009. URL: http://blog.linkedin.com/2009/03/20/project-voldemort-scaling-simple-storage-at-linkedin/.
[34]. D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins и Y. Yerushalmi, «Web Caching with Consistent Hashing,» MIT Laboratory for Computer Science, 1999. URL: http://www8.org/w8-papers/2a-webserver/caching/paper2.html.
[35]. «Oracle Berkeley DB,» Oracle, URL: http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html.
[36]. «Project Voldemort Design,» URL: http://www.project-voldemort.com/voldemort/design.html.
[37]. «Amazon DynamoDB,» Amazon, URL: http://aws.amazon.com/dynamodb/.
[38]. W. Vogels, «Amazon DynamoDB – a Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications,» 18 января 2012. URL: http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html.
[39]. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall и W. Vogels, «Dynamo: Amazon's Highly Available Key-value Store,» в 21st ACM Symposium on Operating Systems Principles, Stevenson, WA, 2007.
[40]. «Redis,» URL: http://redis.io/.
[41]. «Who's using Redis?,» URL: http://redis.io/topics/whos-using-redis.
[42]. «Introducing JSON,» URL: http://json.org/.
[43]. «Riak Users,» URL: http://basho.com/riak-users/.
[44]. «NoSQL Database, In-Memory or Flash Optimized and Web Scale - Aerospike,» URL: http://www.aerospike.com/.
[45]. «Aerospike - Acid Compliant Database for Mission-Critical Applications,» URL: http://www.aerospike.com/performance/acid-compliance/.
[46]. V. Srinivasan и B. Bulkowski, «Citrusleaf: A Real-Time NoSQL DB which Preserves ACID,» в Very Large Databases (VLDB), 2010.
[47]. «DB-Engines Ranking,» URL: http://db-engines.com/en/ranking.
[48]. «MongoDB Production Deployments,» URL: http://www.mongodb.org/about/production-deployments/.
[49]. «CouchDB Lounge,» URL: http://tilgovi.github.com/couchdb-lounge/.
[50]. «CouchDB in the Wild,» URL: http://wiki.apache.org/couchdb/CouchDB_in_the_wild.
[51]. Couchbase, «Learn about Couchbase Server,» URL: http://www.couchbase.com/learn.
[52]. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra и A. Fikes, «Bigtable: A Distributed Storage System for Structured Data,» в OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, 2006.
[53]. «Apache HBase ACID Properties,» URL: http://hbase.apache.org/acid-semantics.html.
[54]. «Cassandra Users,» URL: http://www.datastax.com/cassandrausers.
[55]. 451 Research, «NoSQL, NewSQL and Beyond: The drivers and use cases for database alternatives,» 2011.
[56]. P. Campaniello, «The NewSQL Market Breakdown,» 2011. URL: http://www.scalebase.com/the-story-of-newsql/.
[57]. P. Venkatesh, «NewSQL - The New Way to Handle Big Data,» 2012. URL: http://www.linuxforu.com/2012/01/newsql-handle-big-data/.
[58]. «VoltDB,» URL: http://voltdb.com.
[59]. «H-Store,» URL: http://hstore.cs.brown.edu/.

[60]. M. Stonebraker, «Errors in Database Systems, Eventual Consistency, and the CAP Theorem,» 2010. URL: http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext.
[61]. «Clustrix,» URL: http://www.clustrix.com/.
[62]. «MySQL,» URL: http://www.mysql.com/.
[63]. S. K. Cabral и K. Murphy, MySQL Administrator's Bible, Wiley, 2009.
[64]. «MySQL Cluster,» URL: http://www.mysql.com/products/cluster/.
[65]. «TokuDB for MySQL,» URL: http://www.tokutek.com/products/tokudb-for-mysql/.
[66]. «dbShards,» URL: http://www.dbshards.com.
[67]. «ScaleBase,» URL: http://www.scalebase.com/.
[68]. T. Neward, «The Vietnam of Computer Science,» 26 июня 2006. URL: http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx.

Инструментальные средства оценки качества научно-технических документов[1]

С.В. Герасимов, Р.В. Курынин, И.В. Машечкин, М.И. Петровский, Д.В. Царёв, А.А. Шестимеров
gerasimov@mlab.cs.msu.su, romaha@mlab.cs.msu.su, mash@cs.msu.su, michael@cs.msu.su, tsarev@mlab.cs.msu.su, andy@mlab.cs.msu.su
Факультет вычислительной математики и кибернетики Московского государственного университета имени М.В.Ломоносова

Аннотация. В статье предлагается комбинированный подход к оценке качества научно-технических документов, учитывающий различные категории автоматически рассчитываемых характеристик качества документов — как существующие библиометрические и наукометрические характеристики (рассчитываемые на основе сведений из «цитатных» баз), так и новые типы характеристик, основанные на семантическом анализе текстов научно-технических документов, применении эвристических правил, а также на применении методов оценки наличия прямых текстовых заимствований (плагиата). На основе полученных базовых оценок формируется интегральный показатель оценки качества научно-технических документов с использованием методов машинного обучения аналогично решению задачи ранжирования в информационном поиске. Представлена разработанная экспериментальная система, основанная на предложенном подходе, а также приводятся экспериментальные исследования разработанной системы, направленные на проверку точности оценки научно-технических документов.

Проведённый в статье анализ состояния исследований в РФ и за рубежом в области методов оценки качества научно-технических документов показал, что предложенный в статье подход на основе автоматического расчета базовых оценок из указанного «расширенного» набора групп никем не рассматривалась в настолько широкой постановке и в целом является новаторским.

[1] Работа выполнена при поддержке государственного контракта №14.514.11.4016 и грантов РФФИ 11-07-00616, 12-07-00585.

Ключевые слова: оценка качества научно-технических документов; библиометрия; наукометрия; латентно-семантический анализ; неотрицательная матричная факторизация; тематическое моделирование; методы машинного обучения.

1. Введение

Исследования в области оценки качества научно-технических документов (научно-технических статей, диссертаций, отчетных материалов о НИОКР, заявочных документов на проведение НИОКР, патентной документации и др.) активно ведутся на протяжении более десяти последних лет [[1]]. Традиционно данную оценку осуществляют на основе информации о публикациях авторов и о цитировании их статей, которая собирается в «цитатных» базах [[2]]. Широкое распространение в мире получили цитатные базы, представленные компанией Thomson Reuters (бывший Institute for Scientific Information, ISI): Science Citation Index, Social Sciences Citation Index и Arts & Humanities Citation Index, а также Journal Citation Reports (JCR) [[3]]. Другим примером системы количественного определения значимости публикаций является система CiteSeer, разработанная в 1997 году [[4]]. В России примером «цитатной» базы служит «Российский индекс научного цитирования» (РИНЦ) [[5], [6]] — библиографическая база данных научных публикаций российских ученых.

В дальнейшем были показаны недостатки использования подобных систем, основанных на анализе цитируемости, для оценки качества научных документов [[7], [8]]. В настоящее время активно ведутся исследования по выработке новых индикаторов и методов оценки качества научно-технических документов. Так, в 2008–2011 гг. в рамках седьмой рамочной программы ЕС (European 7th Framework Programme) осуществлялся проект European Educational Research Quality Indicators (EERQI) [[9]], по итогам которого был разработан набор методик и индикаторов оценки качества научных документов на национальных европейских языках (английском, французском, немецком и шведском). Часть разработанных индикаторов использует наиболее популярные «цитатные» базы, а также были предложены новые техники вычисления индикаторов оценки качества [[10]]. Важно отметить, что разработанные в проекте EERQI характеристики, оценивающие «точность», «оригинальность» и «значимость», задаются экспертом, а не вычисляются автоматически. Так, например, характеристика «оригинальность» тесно связана с задачей поиска плагиата и может оцениваться автоматически.

Кроме того, согласно [[11], [12]] можно отметить характеристики, иллюстрирующие типичные ошибки авторов при написании статей: например, «слишком длинная аннотация к статье», «статья не содержит ключевых разделов», «слишком частое использование аббревиатур и/или незначимых слов». Очевидно, что эти характеристики можно легко автоматически вычислять путём задания экспертом соответствующих параметров. Данный

тип эвристических характеристик не встречается в существующих решениях по автоматической оценке качества документов.

Другим недостатком существующих подходов является отсутствие вычислительных критериев (характеристик, индикаторов) оценки качества документов, основанных на семантическом анализе текстового содержимого. Поэтому можно сделать вывод об актуальности исследования и разработки оценок качества научно-технических документов на естественных языках на основе анализа моделей семантики.

В статье предлагается комплексный подход к вычислению оценки качества анализируемых научно-технических документов, основанный на расчете следующих пяти групп базовых оценок: «семантические» (на основе семантических моделей документов); «ссылочные» (основанные на анализе цитирования документов); «репутационные» (использующие наукометрические оценки авторов и журналов, связанных с анализируемым документом); «оценки наличия плагиата» (оценка вероятности наличия прямых текстовых заимствований); «эвристические» (использующие заданные экспертом правила и словари). На основе полученных базовых оценок формируется интегральный показатель оценки качества документов с использованием методов машинного обучения аналогично подходу к решению задачи ранжирования в информационном поиске.

Настоящая статья имеет следующую структуру. В разделе 2 приводится аналитический обзор существующих подходов к оценке качества научно-технических документов. В разделе 3 подробно рассматриваются концепции предлагаемого подхода оценки качества научно-технических документов. Раздел 4 содержит сведения о разработанной экспериментальной системе, основанной на предложенном подходе. Раздел 5 посвящен экспериментальному исследованию разработанной системы.

2. Существующие подходы оценки качества научно-технических документов

Традиционный подход оценки качества научно-технических документов основывается на использовании библиометрических показателей, рассчитываемых на основе сведений из «цитатных» баз, содержащих расширенный набор сведений о публикациях (т.е. не только данные об авторах, заглавии, наименовании журнала, годе, томе, выпуске, страницах, но и о списке литературы, цитируемой и цитирующей данную статью) [[2]]. На основе этой информации вычисляются (по различным методикам) оценки качества как самих статей, так и их авторов и журналов публикаций. В качестве примеров наиболее известных в мире «цитатных» баз можно привести системы продукции компании Thomson Scientific (Science Citation

Index, Social Sciences Citation Index и Arts & Humanities Citation Index, Journal Citation Reports) [[3]], а также системы CiteSeer [[4]] и Google Scholar [[13]].

Поскольку в подавляющем большинстве случаев иностранные индексы («цитатные» базы) работают с англоязычными публикациями, определение библиографических показателей крайне затруднительно для неанглоязычных стран (таких, как Россия, Китай, Франция, Португалия, Япония и др.). Для решения указанной проблемы необходимо создание «цитатной» базы, содержащей необходимые сведения о статьях, публикующихся на национальных языках. В частности, в РФ в 2005 г. проходили исследования, посвященные разработке Российского индекса научного цитирования (РИНЦ) [[5], [6]].

В дальнейшем были показаны и другие недостатки использования подходов на основе анализа цитируемости для оценки качества научных документов, наиболее часто в литературе выделяют следующие [[7], [8]]:

- «шум» в оценке цитируемости, получаемый за счёт цитирования автором своих же статей (самоцитирование);
- «цитатные» базы сами по себе не включают все существующие статьи (проблема неполноты наполнения баз), кроме того, их наполнения сильно различаются друг от друга (различные журналы входят в различные «цитатные» базы), что приводит к различным оценкам одного и того же документа или автора.

В 2008–2011 гг. в рамках седьмой рамочной программы ЕС (European 7th Framework Programme) осуществлялся проект EERQI [[9]], направленный как на решение указанных проблем, так и на выработку новых подходов к оценке качества научно-технических документов. По итогам проекта был разработан набор методик и индикаторов оценки качества научных документов на национальных европейских языках (в качестве тестового набора использовались научные документы на английском, французском, немецком и шведском языках). Часть разработанных индикаторов по-прежнему использует наиболее популярные «цитатные» базы, но также были предложены новые техники вычисления индикаторов оценки качества. Разработанные индикаторы можно условно разделить на два типа:

- «Внутренние» (вычисляются лишь на основе текстового содержимого анализируемого документа) — расчет ведется на основе опросника, который заполняют эксперты. Используются три группы характеристик:
    - точность (англ. *rigour*) — оцениваются такие характеристики, как четкость изложения подхода, методов и результатов, а также полнота и корректность полученных результатов;
    - оригинальность (англ. *originality*) — оценивается новизна предлагаемых методов и подходов;

- значимость (англ. *significance*) — оценивается научный вклад в исследуемую проблемную область.

– «Внешние» (вычисляются с применением метаданных документа, т.е информации о документах из внешних по отношению к ним источников) — программно-инструментальный набор по определению следующих характеристик:
  - Библиографические (с использованием Google Scholar, Google Web Search и MetaGer):
    - количество опубликованных статей (для каждого автора),
    - количество цитирований (для каждого автора),
    - диапазон дат извлеченных публикаций,
    - количество цитирований в год,
    - количество цитирований каждой статьи,
    - g-индекс (улучшенная модификация h-индекса),
    - сопоставление авторства (на основе информации из Google Web Search и MetaGer).
  - На основе сервисов следующих социальных сетей Интернет (оценивается сопоставление авторства и заголовков статей): citeulike (www.citeulike.org), LibraryThing (www.librarything.com), Connotea (www.connotea.org), Mendeley (www.mendeley.com).

Не смотря на то, что в проекте EERQI была предложена идея вычисления индикаторов на основе содержимого анализируемого документа («внутренний» тип индикаторов), все указанные характеристики данного типа, оценивающие точность, оригинальность и значимость, задаются экспертом, а не вычисляются автоматически. Хотя, например, характеристика «оригинальность» тесно связана с задачей поиска плагиата и может оцениваться автоматически. Кроме того, никак не используется семантический анализ текстового содержимого для вычисления каких-либо оценок качества документов.

По итогам обзора публикаций [[11], [12]], посвящённых экспертной оценке качества научных статей, можно отметить характеристики, иллюстрирующие типичные ошибки авторов при написании статей, например, «слишком длинная аннотация к статье», «статья не содержит ключевых разделов», «слишком частое использование аббревиатур и/или незначимых слов». Очевидно, что подобные характеристики можно автоматически вычислять путём задания экспертом соответствующих параметров — рекомендуемой длины аннотации, названия разделов, частоты использования аббревиатур, списка стоп-слов и частоты их встречаемости. Данный тип эвристических характеристик также не встречается в существующих решениях по автоматической оценке качества документов.

3. Концепции предлагаемого подхода оценки качества научно-технических документов

В рамках данной работы авторами предлагается комплексный подход автоматического вычисления оценки качества научно-технических документов, основанный на расчете следующих групп базовых оценок:

- «семантические» — оценки, рассчитываемые с использованием семантического анализа текстового содержимого документов;
- «ссылочные» — библиометрические оценки, основанные на анализе цитирования документов;
- «репутационные» — оценки, использующие библиометрическую и наукометрическую информацию об авторах и журналах документов из внешних по отношению к ним источников (метаданные документов);
- «оценка наличия плагиата» — оценка вероятности наличия прямых текстовых заимствований;
- «эвристические» — оценки, использующие заданные экспертом правила и словари.

При анализе каждого научно-технического документа вычисляются значения каждой базовой оценки, и на их основе рассчитывается интегральный показатель оценки качества с использованием методов машинного обучения аналогично решению задачи ранжирования в информационном поиске. В предлагаемом подходе проводится расчет всех групп базовых оценок и интегрального показателя как для исходного анализируемого документа, так и для коллекции семантически близких ему документов, формируемой в процессе анализа исходного документа.

Ниже в настоящем разделе приводятся описания подходов, используемых для автоматического вычисления всех групп базовых оценок и для расчёта результирующего интегрального показателя.

3.1 «Семантические» базовые оценки

Семантические оценки в предлагаемом подходе основаны на анализе тематических моделей семантики отдельных документов и коллекций документов [[14], [15], [16], [17]]. Тематические модели объединяют семантически схожие термы в тематики, при этом выделенные тематики ставятся в соответствие текстовым фрагментам. При тематическом моделировании отдельного документа в качестве текстовых фрагментов использовались предложения текста, при тематическом моделировании коллекции документов — сами документы.

Для тематического моделирования используется метод латентно-семантического анализа (англ. *Latent semantic analysis, LSA*) [[14], [15], [16]]. Основная идея данного метода состоит в том, что совокупность всех текстовых фрагментов приводит к взаимным ограничениям использований термов, определяющим сходство семантических значений термов и фрагментов. Латентно-семантический анализ работает с векторным представлением типа *“bag-of-words”* текстовых фрагментов [[14]]. Таким образом, анализируется числовая матрица отображения терм-фрагмент, строки которой соответствуют термам, а столбцы — фрагментам. Объединение термов в тематики и представление фрагментов в пространстве тематик осуществляется путём применения к данной матрице одного из матричных разложений. В настоящее время наиболее популярными матричными разложениями являются сингулярное разложение (англ. *Singular Value Decomposition, SVD*) [[14]] и неотрицательная матричная факторизации (англ. *Non-negative Matrix Factorization, NMF*) [[15]]. После применения к текстовой матрице одного из указанных матричных разложений формируется семантическая модель совокупности текстовых фрагментов, состоящая из:

1. Матрицы отображения пространства термов в пространство тематик.
2. Вектора, чьи элементы соответствуют весу выделенных тематик в тексте (или диагональной матрицы, чьи диагональные элементы соответствуют весам тематик). Для сингулярного разложения веса выделенных тематик аппроксимируются квадратом соответствующих сингулярных чисел [[18], [19]]. В случае применения неотрицательной матричной факторизации веса тематик предлагается рассчитывать на основе оригинального метода, разработанного коллективом авторов, для которого была показана применимость для задачи выделения ключевых предложений текста (автоматического аннотирования) [[20], [21], [22], [23]].
3. Матрицы представления текстовых фрагментов в пространстве тематик.

На основе получаемой информации о тематиках документов и коллекций документов вычисляются следующие семантические оценки качества документов:

1. *Оценки информационной сжимаемости документа (тематическое моделирование отдельного документа).* Используя веса выделенных тематик в качестве оценок количества информации, содержащейся в них, можно определить минимальное число тематик, требующееся для покрытия заданного процента информации, содержащейся в тексте[19]. Аналогично, используя тематическое представление текста документа, можно вычислить значимость (релевантность) каждого предложения текста [[20]], которую также можно использовать в качестве оценки количества информации, содержащейся в них. Таким образом, задав сетку

информационных процентов от 10% до 100% с шагом 10%, получаем списки, состоящие из 10 элементов, содержащие спектр оценок информационной сжимаемости документа по его предложениям и тематикам, соответственно, для сингулярного разложения и неотрицательной матричной факторизации.

2. *Оценки принадлежности документа к набору тематик коллекции документов (тематическое моделирование коллекции документов)*. Одним из результатов построения любой из рассматриваемых семантических моделей для коллекции семантически близких документов является представление документов в пространстве тематик коллекции. Таким образом, в качестве базовых семантических оценок качества документа использовался вектор его представления в пространстве тематик коллекции, получаемый при тематическом моделировании коллекции с помощью матричной факторизации [[15], [24]].

3.2 «Ссылочные» и «репутационные» базовые оценки

Предлагается в расчетах «ссылочных» и «репутационных» оценок использовать значения характеристик, приведенных в табл. 1 (для характеристик, значения которых получить не удается, допускается использовать значение «неизвестно»). Данные характеристики определяются для всех документов, участвующих в расчетах «ссылочных» и «репутационных» оценок [[6]].

Табл. 1 Характеристики ссылочных и репутационных оценок.

Категория	Характеристика	Способ получения
Информация о документе	год издания	на основе метаданных
	длина списка библиографии	на основе метаданных
	количество цитирований	на основе метаданных, из информационной базы или из внешних источников
	значение PageRank	рассчитываемая величина
	информация о распределении значений PageRank цитируемых	рассчитываемая величина

Категория	Характеристика	Способ получения
	документов	
	информация о распределении длин библиографических списков цитируемых статей	рассчитываемая величина
	информация о распределении значений индексов Хирша авторов документа	рассчитываемая величина
Информация о каждом авторе документа	индекс Хирша	из информационной базы или из внешних источников
Информация об организациях, в которых состоят авторы	индекс Хирша	из информационной базы или из внешних источников
Информация о журнале, в котором публикуется документ	классический импакт-фактор	из информационной базы или из внешних источников

В данной таблице под *значением PageRank* понимается рассчитанное значение, получаемое на основе применения алгоритма PageRank, описанного в [[25]], к некоторому набору документов, в качестве связей между которыми выступает цитирование (т.е. объектом анализа является ориентированный граф цитирования, где узлы соответствуют документам, а дуги — ссылкам одних документов на другие).

В качестве *информации о распределении значений* какой-либо величины понимается совокупность рассчитанных значений следующих статистических характеристик:

- минимум;
- первый квартиль (известный в литературе также, как нижний квартиль, или 0,25-квантиль);
- медиана (0,5-квантиль);
- третий квартиль (или верхний квартиль, или 0,75-квантиль);
- максимум.

В третьей колонке таблицы приводятся способы определения значения характеристики:

- на основе метаданных — значение характеристики определяется на основе текстового содержимого самого документа;
- из информационной базы — значение характеристики определяется из информационной базы, представляющей собой локальное хранилище, содержащее наукометрические и библиометрические данные о документах, полученные из внешних «цитатных» баз;
- из внешних источников — значение определяется с использованием данных из внешних источников (например, из какой-либо «цитатной» базы);
- рассчитываемая величина — значение характеристики рассчитывается в ходе проведения анализа по расчету «ссылочных» и «репутационных» оценок.

3.3 «Оценка наличия плагиата»

Для анализа документов на наличие прямых текстовых заимствований предлагается подход, основанный на следующих этапах:

- Разбивка исходного документа на текстовые фрагменты (англ. *chunking*).
- Получение коллекции документов для анализа на основе поискового запроса, сформированного из ключевых слов анализируемого документа (выделение ключевых слов происходит на стадии семантического анализа текста [[29]]), и библиографии документа.
- Фильтрация полученного набора для выделения подмножества документов для детального сравнения. На основе сравнения алгоритмов поиска заимствований на коллекции PAN [[26]] для фильтрации документов предлагается использовать представление в виде символьных n-грамм и n-грамм из слов. В качестве меры сходства предлагается использовать меру Жаккара [[27]] и скалярное произведение.
- Детальный анализ схожести двух документов. При определении численной характеристики присутствия заимствований предлагается использовать метод *String Sequence Kernel (SSK*) [[28]], устойчивый к локальным модификациям заимствованного текста. При использовании метода предлагается следующая модификация: в суммарную оценку сходства двух документов входит сходство для длин подпоследовательностей на отрезке [k, n] (где k и n — минимальная и максимальная длины подпоследовательностей), что

уменьшает влияние совпадающих устойчивых слов или выражений, связанных с общей тематикой сравниваемых текстов.

3.4 «Эвристические» оценки текста

В задачах поиска плагиата и Web-поиска часто используются статистические характеристики — эвристики, направленные на оценку «читабельности» (англ. *readability*) текста, т.е. простоту восприятия текста, и на оценку его стилистического написания [[30], [31], [32]]. К примерам эвристик «читабельности» можно отнести такие характеристики, как средняя длина предложений (в словах) и средняя длина слов (в символах) текста [[32]]. В качестве стилистических эвристик можно привести количество употребления частиц, предлогов, а также среднюю длину предложений и слов [[30], [31]]. В работе [[30]] показывается, как анализ набора стилистических характеристик позволяет определить принадлежность текста к конкретному автору.

Как видно даже из приведённых примеров, наборы эвристик для оценок «читабельности» и стилистического написания пересекаются. Поэтому приведём единый список наиболее популярных эвристик рассматриваемых категорий, которые используются в качестве эвристических базовых оценок качества научно-технических документов:

1. Среднее количество слов в предложениях (средняя длина предложений);
2. Среднее количество символов в словах (средняя длина слов);
3. Доля слов длиннее 7 символов;
4. Частота употребления стоп-слов (предлогов, союзов, частиц) в тексте;
5. Средняя по предложениям частота употребления стоп-слов;
6. Среднее количество знаков пунктуации на предложение;
7. Количество знаков экспрессивной пунктуации («!», «?», «…»);
8. Среднее по предложениям количество знаков экспрессивной пунктуации.

3.5 Расчёт интегрального показателя оценки качества научно-технических документов

В рамках данной работы предлагается подход вычисления оценки качества научно-технического документа как интегральной оценки на основе комбинации индивидуальных базовых оценок качества — семантических оценок, репутационных, ссылочных, эвристических, а также оценок уровня текстуального заимствования. Данный подход основан на использовании методов машинного обучения аналогично подходу к решению задачи ранжирования в задачах информационного поиска. Алгоритмы машинного обучения обладают возможностью обобщения, они способны корректно ранжировать или классифицировать документы, которые не встречались непосредственно в обучающей выборке. В нашем случае предлагается процедура расчета интегральной оценки, включающей три этапа:

1. Формирование обучающей выборки документов — эксперт ранжирует документы из выборки по их качеству, т.е. задает относительный порядок на основе попарных сравнений. На основе заданных попарных сравнений качества документов строится оценка относительных рангов документов коллекции на шкале с использование модели Бредли-Терри с ничьей [[33]]. Оценки попарных сравнений качества документов могут задаваться экспертно или вычисляться на основе любой из базовых оценок (например, индекса цитируемости).
2. Строится модель ранжирования в виде функции экспоненциальной регрессии [[34]] с пошаговым выбором независимых переменных (для борьбы с переобучением) [[35]] для прогнозирования относительного ранга, рассчитанного на предыдущем этапе.
3. Построенная модель ранжирования используется для прогноза ранга научно-технического документа относительно заданной коллекции семантически близких ему документов. Относительный ранг в рамках коллекции семантически близких документов рассматривается как оценка качества научно-технического документа.

4. Экспериментальная система оценки качества научно-технических документов

Данный раздел статьи посвящён архитектуре экспериментальной системы (ЭС), выполняющей вычислительную оценку качества научно-технических документов. В разделе приводятся описания программных компонент, входящих в состав ЭС, и используемые технологии.

Реализация ЭС предполагает проведение многоэтапной разнородной обработки поданных на вход документов. Приведём описание предложенных этапов обработки коллекции документов в виде описаний задач, которые выполняются на том или ином этапе.

1. Этап «Базовая обработка» (БО) — выполняется с каждым анализируемым документом. Данный этап обработки документа включает в себя три задачи:
   - Первичная обработка: выделение из файла анализируемого документа текстового содержимого (текста) и метаинформации, включающей авторов и заголовок документа, список библиографии; идентификация документа по метаинформации в информационной базе системы.
   - Обработка текста: вычисление оценок информационной сжимаемости и эвристических оценок документа; выделение ключевых слов документа.
   - Обработка метаинформации: поиск библиометрической и наукометрической информации во внешних источниках; вычисление ссылочных и репутационных оценок качества документа.
2. Этап «Формирование контекста». Данный этап состоит из одной задачи: для каждого анализируемого документа выполняется поиск документов, сходных с анализируемым, по ключевым словам, сформированным на предыдущем этапе, в сети Интернет и локальном хранилище системы (представляющим собой индексированный массив научно-технических документов); объединение всех найденных документов с анализируемыми в расширенный контекст.
3. Этап «Базовая обработка контекста». Данный этап состоит из одной задачи — выполнение базовой обработки документов расширенного контекста, для которых данная обработка ещё не была произведена.
4. Этап «Семантический анализ контекста». Данный этап состоит из одной задачи: проведение вычисления оценки принадлежности документов к набору тематик расширенного контекста документов (тематическое моделирование коллекции документов), на основе которого производится построение семантического контекста — коллекции семантически близких документов, полученной путем удаления из расширенного контекста документов, семантически далеких от исходных анализируемых.
5. Этап «Оценка текстовых заимствований». Данный этап состоит из одной задачи — вычисление базовых оценок качества всех документов семантического контекста на основе оценок степени взаимного текстового заимствования.

6. Этап «Ранжирование». Данный этап обработки документов включает в себя две задачи:
   - формирование попарных сравнений качества документов;
   - ранжирование документов (вычисление интегрального показателя).

Часть задач, входящих в рассмотренные этапы обработки, может выполняться параллельно (например, обработка текста и обработка метаинформации в «Базовой обработке»); другие задачи связанны по входным/выходным данным и требуют последовательного исполнения (например, цепочка последовательного выполнения задач этапов «Формирование контекста», «Базовая обработка контекста», «Семантический анализ контекста»). При этом результаты отдельных задач могут быть использованы сразу в нескольких других задачах (например, исходный текст документа необходим при вычислении задачи обработки текста в «Базовой обработке» и задачи этапа «Оценка текстовых заимствований»). Некоторые этапы обработки могут занимать значительное время (например, обработка метаинформации в «Базовой обработке», т.к. в этой задаче выполняется анализ внешних источников наукометрической информации, вычисление ссылочных и репутационных базовых оценок). Поэтому важно, чтобы пересчет последующих этапов обработки (например, в случае аппаратного сбоя) не приводил к пересчету результатов нижележащих этапов.

Перечисленные особенности процедуры обработки исходных документов привели к необходимости создания в рамках ЭС подсистемы управления вычислительными задачами (Диспетчер задач). Таким образом, диспетчер задач должен удовлетворять следующим требованиям:

1. Разбиение исходной процедуры обработки научно-технических документов на этапы, предполагающие как последовательное, так и параллельное исполнение.
2. Возможность долговременного хранения результатов вычислений.
3. Возможность пересчета выбранных этапов и этапов, использующих их, без необходимости полного пересчета всех задач.
4. Распределенное вычисление задач.
5. Восстановление корректного состояния вычислений после аппаратного сбоя.
6. Мониторинг состояния вычислений.

На рис. 1 представлены программные компоненты, реализующие задачи этапов обработки, и программный компонент управления ими.

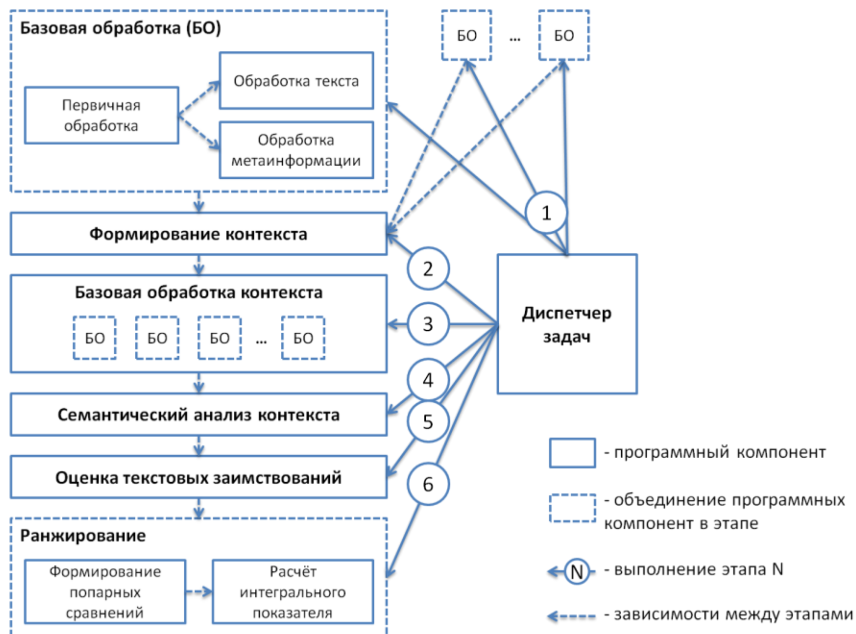


Рис. 1. *Программные компоненты ЭС.*

Диспетчер задач, реализующий механизм вычислительных задач ЭС, основывается на фреймворке Twisted Framework [[36]]. Использование асинхронного программирования позволило эффективно организовать в ЭС вычисления, обусловленные всеми стадиями обработки научно-технических документов, исполняя «легкие» операции и операции ввода-вывода в основной нити Twisted, направляя относительно ресурсоемкие вычисления в пул нитей, а вычисления, требующие значительного процессорного времени, выносить в отдельные исполняемые модули, написанные на языке программирования C++ и исполняемые в отдельных процессах ОС либо на отдельных физических машинах, связанных локальной сетью. Так, задача обработки метаинформации, вычисляющая репутационные и ссылочные оценки, реализована на языке Python и выполняется в основной нити Twisted, а задача расчета интегрального показателя реализована на языке C++ и выполняется в отдельном процессе ОС.

Указанный механизм задач обладает поддержкой регулирования степени параллелизма, используемого при вычислении значений задач. Механизм позволяет задать общее ограничение на число задач, параллельно исполняемых в рамках процессов ОС.

Исходные тексты ЭС и все используемые в ЭС сторонние библиотеки являются кросс-платформенными в рамках ОС семейств Unix (ядро Linux версии 2.4 и выше) и MS Windows.

Пользовательский интерфейс ЭС реализован на языке JavaScript в дополнительном программном компоненте взаимодействия с пользователем. Реализация базируется на использовании Ajax-фреймворка для создания пользовательского интерфейса «насыщенных» Интернет-приложений qooxdoo [[37]], поддерживающего следующие версии веб-браузеров: Internet Explorer 6+, Firefox 2+, Opera 9+, Safari 3.0+ и Chrome 2+.

5. Результаты экспериментального исследования

Настоящий раздел посвящен проведению экспериментальных исследований ЭС по проверке точности рассчитываемой оценки качества научно-технических документов на русском и английском языках.

Для проведения экспериментов были сформированы тестовые наборы данных, содержащие целевые документы:

- первый тестовый набор сформирован из русскоязычных публикаций. В него вошли статьи из конференции «Математические методы распознавания образов» 2011 г. [[38]] и тематического сборника факультета ВМК МГУ имени М.В.Ломоносова «Программные системы и инструменты» 2012 г.;
- второй тестовый набор сформирован из англоязычных публикаций. В него вошли статьи из конференций ICDM 2006 [[39]] и ICMET 2011 [[40]].

Каждый тестовый набор состоит из 10 документов, написанных на схожие тематики. При этом документы каждого тестового набора были размечены экспертом на две категории «А» и «Б» по 5 документов. Категоризация документов каждого тестового набора выполнялась по следующей эвристике: документ категории «Б» имеет худшее качество по сравнению с документом категории «А» для любых документов из тестового набора.

Для каждого тестового набора прошло по одной серии экспериментов. Каждая серия экспериментов заключалась в формировании различных правил разметки тестового набора на основе попарных сравнений, т.е. в задании правил вида «документ Д1 лучше документа Д2», и последующем ранжировании тестового набора. При этом сформированные правила попарных сравнений различались как по составу участвующих документов, так и по количеству самих правил. Т.к. каждый тестовый набор состоит из 5 документов категории «А» и 5 документов категории «Б», то всего можно сформировать 25 различных правил попарных сравнений (без ничьих).

В дальнейшем для оценки точности ранжирования использовался критерий *Ranking Loss* [[41]], который отражает среднюю долю некорректно

упорядоченных пар документов (рис. 2). Пара документов считается некорректно упорядоченной, если документ категории «Б» располагался выше документа категории «А» в соответствии с рассчитанным рангом. Чем меньше значение *Ranking Loss*, тем лучше качество ранжирования. Качество ранжирования совершенно, когда *Ranking Loss = 0*. Итоговая точность ранжирования оценивалась в процентах и вычислялась по формуле: 100*(1 – *Ranking Loss*).

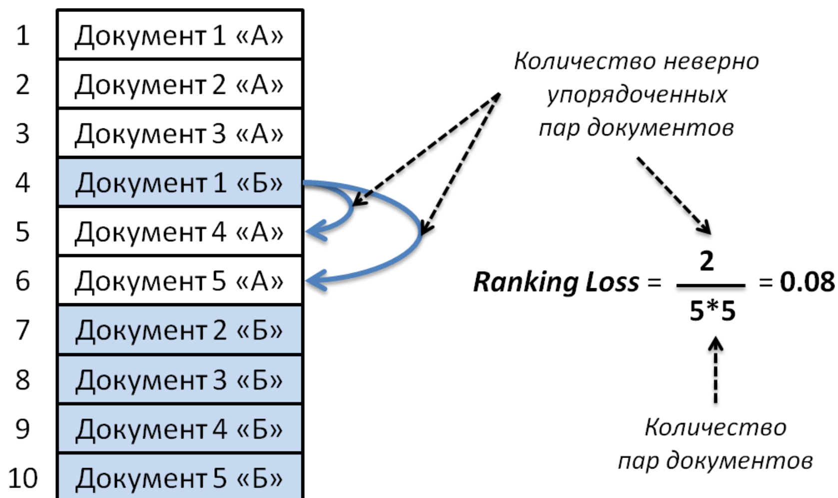


Рис. 2. *Пример вычисления критерия Ranking Loss для оценки качества ранжирования.*

В результате экспериментального исследования было получено, что достаточно сформировать 4-5 правил попарных сравнений, что составляет 16-20% от общего числа различных правил, для получения точности не меньшей 98.8%.

6. Заключение

В статье предлагается комплексный подход к оценке качества научно-технических документов, включающий автоматический расчет следующих групп базовых оценок качества: семантические, построенные на основе семантических моделей отдельных документов и коллекции семантически близких документов; библиометрические и наукометрические, основанные на анализе графа цитирования документов и использующие репутационные оценки авторов и изданий; оценки наличия прямых текстовых заимствований из семантически близких документов; эвристические, использующие заданные экспертом правила и словари. На основе полученных базовых оценок

формируется интегральный показатель оценки качества научно-технических документов с использованием методов машинного обучения аналогично решению задачи ранжирования в информационном поиске. Для этого на основе попарных сравнений качества документов коллекции, заданных экспертно или вычисленных на основе базовых оценок, строится модель ранжирования, которая определяет интегральную оценку как функцию от совокупности базовых оценок, представленных выше типов. Полученная функция ранжирования далее применяется для вычисления интегральной оценки качества документов, семантически близких к данной коллекции.

В статье были представлены экспериментальные исследования по проверке точности рассчитываемой оценки качества научно-технических документов на русском и английском языках. Для этого были разработаны тестовые наборы данных, содержащие целевые документы на русском и английском языках, соответственно. Исследования показали достаточность задания 4-5 правил попарных сравнений, что составляет 16-20% от общего числа различных правил, для получения точности оценки документов не меньшей 98.8%.

В настоящее время проблема оценки качества научно-технических документов на основе автоматического расчета базовых оценок из предложенного «расширенного» набора групп никем не решена, и даже не рассматривалась в настолько широкой постановке. На основе приведённого анализа состояния исследований в РФ и за рубежом можно сделать вывод, что предложенный в статье подход для решения данной проблемы в целом является новаторским.

7. Список литературы

[1]. Steve Lawrence, Kurt Bollacker, C . Lee Giles. Indexing and Retrieval of Scientific Literature // Eighth International Conference on Information and Knowledge Management, CIKM 99, Kansas City, Missouri, November 2–6, pp. 139–146, 1999.

[2]. В.В. Писляков. Методы оценки научного знания по показателям цитирования // М.: Социологический журнал, 2007, N1, стр. 128-140.

[3]. Официальный сайт ISI Web of Knowledge (ныне подразделение Healthcare & Science business в Thomson Reuters) // http://www.webofknowledge.com.

[4]. Официальный сайт системы CiteSeer // http://citeseerx.ist.psu.edu.

[5]. Российский Индекс Научного Цитирования // http://elibrary.ru/project_risc.asp.

[6]. Писляков В. В. Наукометрические методы и практики, рекомендуемые к применению в работе с российским индексом научного цитирования // Отчёт о научно-исследовательской работе (промежуточный) по теме «Разработка системы статистического анализа российской науки на основе данных российского индекса цитирования». — М., 2005.

[7]. Meho L (Meho, Lokman); Yang K (Yang, Kiduk). Fusion approach to citation-based quality assessment // Proceedings Of Issi 2007: 11th International Conference Of The International Society For Scientometrics And Informetrics, Vols I And II : 568-581.

[8]. Angela Vorndran, Alexander Botte. Analysis and evaluation of existing methods and indicators for quality assessment of scientific publications // http://www.eerqi.eu/sites/default/files/Analysis_and_evaluation_of_existing_methods_and_indicators.pdf [PDF].
[9]. Официальный сайт проекта EERQI – European Educational Research Quality Indicators // www.eerqi.eu.
[10]. EERQI Project Final Report (2011) // http://eerqi.eu/sites/default/files/Final_Report.pdf [PDF].
[11]. Moyses Szklo. Quality of scientific articles // Revista Saúde Pública vol.40 special issue São Paulo Aug. 2006.
[12]. Dr Navneet Gupta BSc (Hons) PhD MCOptom FBCLA. How to Evaluate a Scientific Research Article // http://www.optometry.co.uk/uploads/articles/ARTICLE%200309.pdf [PDF].
[13]. Официальный сайт системы Google Scholar// http://scholar.google.ru.
[14]. Berry M.W., Dumais S.T., O'Brien G.W. Using Linear Algebra for Intelligent Information Retrieval // University of Tennessee Knoxville. TN. USA, 1994.
[15]. Lee D.D., Seung H.S. Learning the parts of objects by non-negative matrix factorization // Nature, 401, pp. 788-791, 1999.
[16]. Rakesh P., Shivapratap G., Divya G., Soman KP. Evaluation of SVD and NMF Methods for Latent Semantic Analysis // International Journal of Recent Trends in Engineering, Vol. 1, No. 3, 2009.
[17]. Griffiths T L, Steyvers M. Finding scientific topics // In: Proceedings of the National Academy of Sciences. USA, 2004, 101: 5228–5235.
[18]. Steinberger J., Ježek K. Text Summarization and Singular Value Decomposition // In Lecture Notes for Computer Science vol. 2457, Springer-Verlag, 2004, pp. 245-254.
[19]. Steinberger J. Text Summarization within the LSA Framework // PhD Thesis, University of West Bohemia in Pilsen, Czech Republic, January 2007.
[20]. Машечкин И.В., Петровский М.И., Царёв Д.В. Методы вычисления релевантности фрагментов текста на основе тематических моделей в задаче автоматического аннотирования // Вычислительные методы и программирование. Том 14, 2013. 91-102.
[21]. Mashechkin I.V., Petrovskiy M.I., Popov D.S., Tsarev D.V. Automatic text summarization using latent semantic analysis // Programming and Computer Software, pp. 299-305, 2011.
[22]. Tsarev D., Petrovskiy M., Mashechkin I. Using NMF-based text summarization to improve supervised and unsupervised classification // 11th International Conference on Hybrid Intelligent Systems (HIS), Malacca, MALAYSIA. P. 185-189, 2011.
[23]. Dmitry Tsarev, Mikhail Petrovskiy and Igor Mashechkin, Supervised and Unsupervised Text Classification via Generic Summarization International Journal of Computer Information Systems and Industrial Management Applications. MIR Labs, Volume 5, 2013, pp. 509-515.
[24]. Wei Xu, Xin Liu, Yihong Gong Document clustering based on non-negative matrix factorization // Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval, Toronto, Canada, 2003.
[25]. Y. Ding. Applying weighted PageRank to author citation networks. In Proceedings of JASIST. 2011, pp. 236-245.

[26]. M. Potthast, T. Gollub, M. Hagen, J. Graßegger, J. Kiesel, M. Michel, A. Oberländer, M. Tippmann, A. Barrón-Cedeño, P. Gupta, P. Rosso, B. Stein. Overview of the 4th International Competition on Plagiarism Detection. CLEF2012. 2012.
[27]. S. Alzahrani, N. Salim. Fuzzy Semantic-Based String Similarity for Extrinsic Plagiarism Detection, Lab Report for PAN at CLEF2010, 2010.
[28]. A. Martins. String kernels and similarity measures for information retrieval. 2006.
[29]. Berry M.W., Browne M., Langville A.N., Pauca V.P., Plemmons R.J. Algorithms and applications for approximate nonnegative matrix factorization // Computational Statistics and Data Analysis, pp. 155-173, 2007.
[30]. Фоменко В.П., Фоменко Т.Г. Авторский инвариант русских литературных текстов. Предисловие А.Т. Фоменко. // Фоменко А.Т. Новая хронология Греции: Античность в средневековье. Т. 2. М.: Изд-во МГУ, 1996, с.768-820.
[31]. Braslavski P. Document Style Recognition Using Shallow Statistical Analysis. In Proceedings of the ESSLLI 2004 Workshop on Combining Shallow and Deep Processing for NLP, Nancy, France, 2004, p. 1–9.
[32]. DuBay, W.H. The Principles of Readability. Costa Mesa, CA: Impact Information. 2004.
[33]. P.V. Rao and L.L. Kupper, “Ties in paired-comparison experiments: A generalization of the Bradley–Terry model”, Amer. Statist. Assoc, 62, 1967, pp. 194–204.
[34]. Turner, H and Firth, D (2012). Bradley-Terry Models in R: The BradleyTerry2 Package. Journal of Statistical Software 48(9), 1–21.
[35]. Hastie, Tibshirani and Friedman (2008). The Elements of Statistical Learning (2nd edition) Springer-Verlag. 763 pages.
[36]. Официальный сайт Twisted Framework // http://twistedmatrix.com.
[37]. Официальный сайт qooxdoo // http://qooxdoo.org.
[38]. Конференция «Математические методы распознавания образов» // http://www.mmro.ru.
[39]. The IEEE International Conference on Data Mining (ICDM) // http://www.cs.uvm.edu/~icdm.
[40]. International Conference on Mechanical and Electrical Technology (ICMET) // http://www.icmet.ac.cn.
[41]. Zhang M.-L., Zhou Z.-H. A k-nearest neighbor based algorithm for multi-label classification // Proceedings of the 1st IEEE International Conference on Granular Computing (GrC'05). Beijing, China, 2005. pp. 718-721.

Tools for Quality Assessment of Scientific and Technical Documents

S.V. Gerasimov, R.V. Kurynin, I.V. Mashechkin, M.I. Petrovskiy, Tsarev D.V., A.A.Shestimerov
Moscow State University, Moscow, Russia
gerasimov@mlab.cs.msu.su, romaha@mlab.cs.msu.su, mash@cs.msu.su, michael@cs.msu.su, tsarev@mlab.cs.msu.su, andy@mlab.cs.msu.su

Abstract: In the paper the complex approach to scientific and technical document quality assessment is proposed based on various automatically calculated document quality characteristics as widely used bibliometric and scientometric (based on citation indices), and the new types of characteristics based on the text semantic analysis, heuristics, and also on plagiarism detection methods. The integrated indicator of scientific and technical document quality assessment is formed on the basis of the received basic characteristics with use of machine learning methods similar to the problem of ranking in information retrieval. The developed prototype system based on offered approach is presented, and also the experimental investigations of the developed system directed on check of scientific and technical document quality assessment accuracy are carried out. The analysis of the state of art researches of scientific and technical document quality assessment showed the offered approach based on enhanced list of basic characteristic groups was considered by nobody in so broad statement and as a whole is innovative. The main part of the paper has the following structure. The second section contains an analytical overview of existing approaches to assess quality of scientific and technical documents. The third section provides detail of a proposed approach to assess quality of scientific and technical documents. The forth section describes a prototype system based on the proposed approach. The fifth section discusses results of experiments.

Keywords: scientific and technical document quality assessment; bibliometrics; scientometrics; latent semantic analysis; non-negative matrix factorization; topic model; machine learning

References

[1]. Steve Lawrence, Kurt Bollacker, C . Lee Giles. Indexing and Retrieval of Scientific Literature. Eighth International Conference on Information and Knowledge Management, CIKM 99, Kansas City, Missouri, November 2–6, pp. 139–146, 1999.
[2]. V.V. Pisljakov. Metody ocenki nauchnogo znanija po pokazateljam citirovanija [Methods of assessment of scientific knowledge in terms of citation]. M.: Sociologicheskij zhurnal, 2007, N1, str. 128-140 (in Russian).
[3]. ISI Web of Knowledge. http://www.webofknowledge.com.
[4]. CiteSeer. http://citeseerx.ist.psu.edu.
[5]. Rossijskij Indeks Nauchnogo Citirovanija [Russian Science Citation Index]. http://elibrary.ru/project_risc.asp (in Russian).
[6]. Pisljakov V. V. Naukometricheskie metody i praktiki, rekomenduemye k primeneniju v rabote s rossijskim indeksom nauchnogo citirovanija [Scientometric methods and

practices that are recommended for use in working with the Russian Science Citation Index]. Otchjot o nauchno-issledovatel'skoj rabote (promezhutochnyj) po teme «Razrabotka sistemy statisticheskogo analiza rossijskoj nauki na osnove dannyh rossijskogo indeksa citirovanija». — M., 2005 (in Russian).

[7]. Meho L (Meho, Lokman); Yang K (Yang, Kiduk). Fusion approach to citation-based quality assessment. Proceedings Of Issi 2007: 11th International Conference Of The International Society For Scientometrics And Informetrics, Vols I And II : 568-581.

[8]. Angela Vorndran, Alexander Botte. Analysis and evaluation of existing methods and indicators for quality assessment of scientific publications. http://www.eerqi.eu/sites/default/files/Analysis_and_evaluation_of_existing_methods_and_indicators.pdf [PDF].

[9]. EERQI – European Educational Research Quality Indicators. www.eerqi.eu.

[10]. EERQI Project Final Report (2011). http://eerqi.eu/sites/default/files/Final_Report.pdf [PDF].

[11]. Moyses Szklo. Quality of scientific articles. Revista Saúde Pública vol.40 special issue São Paulo Aug. 2006.

[12]. Dr Navneet Gupta BSc (Hons) PhD MCOptom FBCLA. How to Evaluate a Scientific Research Article. http://www.optometry.co.uk/uploads/articles/ARTICLE%200309.pdf [PDF].

[13]. Google Scholar. http://scholar.google.ru.

[14]. Berry M.W., Dumais S.T., O'Brien G.W. Using Linear Algebra for Intelligent Information Retrieval. University of Tennessee Knoxville. TN. USA, 1994.

[15]. Lee D.D., Seung H.S. Learning the parts of objects by non-negative matrix factorization. Nature, 401, pp. 788-791, 1999.

[16]. Rakesh P., Shivapratap G., Divya G., Soman KP. Evaluation of SVD and NMF Methods for Latent Semantic Analysis. International Journal of Recent Trends in Engineering, Vol. 1, No. 3, 2009.

[17]. Griffiths T L, Steyvers M. Finding scientific topics. In: Proceedings of the National Academy of Sciences. USA, 2004, 101: 5228–5235.

[18]. Steinberger J., Ježek K. Text Summarization and Singular Value Decomposition. In Lecture Notes for Computer Science vol. 2457, Springer-Verlag, 2004, pp. 245-254.

[19]. Steinberger J. Text Summarization within the LSA Framework. PhD Thesis, University of West Bohemia in Pilsen, Czech Republic, January 2007.

[20]. Mashechkin I.V., Petrovskij M.I., Carjov D.V. Metody vychislenija relevantnosti fragmentov teksta na osnove tematicheskih modelej v zadache avtomaticheskogo annotirovanija [Methods for calculating the relevance of text fragments on the basis of thematic patterns in the problem of automatic annotation]. Vychislitel'nye metody i programmirovanie. Tom 14, 2013. 91-102 [in Russian].

[21]. Mashechkin I.V., Petrovskiy M.I., Popov D.S., Tsarev D.V. Automatic text summarization using latent semantic analysis. Programming and Computer Software, pp. 299-305, 2011.

[22]. Tsarev D., Petrovskiy M., Mashechkin I. Using NMF-based text summarization to improve supervised and unsupervised classification. 11th International Conference on Hybrid Intelligent Systems (HIS), Malacca, MALAYSIA. P. 185-189, 2011.

[23]. Dmitry Tsarev, Mikhail Petrovskiy and Igor Mashechkin, Supervised and Unsupervised Text Classification via Generic Summarization International Journal of Computer Information Systems and Industrial Management Applications. MIR Labs, Volume 5, 2013, pp. 509-515.

[24]. Wei Xu, Xin Liu, Yihong Gong Document clustering based on non-negative matrix factorization. Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval, Toronto, Canada, 2003.
[25]. Y. Ding. Applying weighted PageRank to author citation networks. In Proceedings of JASIST. 2011, pp. 236-245.
[26]. M. Potthast, T. Gollub, M. Hagen, J. Graßegger, J. Kiesel, M. Michel, A. Oberländer, M. Tippmann, A. Barrón-Cedeño, P. Gupta, P. Rosso, B. Stein. Overview of the 4th International Competition on Plagiarism Detection. CLEF2012. 2012.
[27]. S. Alzahrani, N. Salim. Fuzzy Semantic-Based String Similarity for Extrinsic Plagiarism Detection, Lab Report for PAN at CLEF2010, 2010.
[28]. A. Martins. String kernels and similarity measures for information retrieval. 2006.
[29]. Berry M.W., Browne M., Langville A.N., Pauca V.P., Plemmons R.J. Algorithms and applications for approximate nonnegative matrix factorization. Computational Statistics and Data Analysis, pp. 155-173, 2007.
[30]. Fomenko V.P., Fomenko T.G. Avtorskij invariant russkih literaturnyh tekstov [Author invariant Russian literary texts]. Predislovie A.T. Fomenko.. Fomenko A.T. Novaja hronologija Grecii: Antichnost' v srednevekov'e. T. 2. M.: Izd-vo MGU, 1996, c.768-820 (in Russian].
[31]. Braslavski P. Document Style Recognition Using Shallow Statistical Analysis. In Proceedings of the ESSLLI 2004 Workshop on Combining Shallow and Deep Processing for NLP, Nancy, France, 2004, p. 1–9.
[32]. DuBay, W.H. The Principles of Readability. Costa Mesa, CA: Impact Information. 2004.
[33]. P.V. Rao and L.L. Kupper, “Ties in paired-comparison experiments: A generalization of the Bradley–Terry model”, Amer. Statist. Assoc, 62, 1967, pp. 194–204.
[34]. Turner, H and Firth, D (2012). Bradley-Terry Models in R: The BradleyTerry2 Package. Journal of Statistical Software 48(9), 1–21.
[35]. Hastie, Tibshirani and Friedman (2008). The Elements of Statistical Learning (2nd edition) Springer-Verlag. 763 pages.
[36]. Twisted Framework. http://twistedmatrix.com.
[37]. qooxdoo. http://qooxdoo.org.
[38]. Konferencija «Matematicheskie metody raspoznavanija obrazov» [The Conference «Mathematical Methods of Pattern Recognition»]. http://www.mmro.ru (In Russian).
[39]. The IEEE International Conference on Data Mining (ICDM). http://www.cs.uvm.edu/~icdm.
[40]. International Conference on Mechanical and Electrical Technology (ICMET). http://www.icmet.ac.cn.
[41]. Zhang M.-L., Zhou Z.-H. A k-nearest neighbor based algorithm for multi-label classification. Proceedings of the 1st IEEE International Conference on Granular Computing (GrC'05). Beijing, China, 2005. pp. 718-721.

Современные методы поиска и индексации многомерных данных в приложениях моделирования больших динамических сцен

Золотов В.А., Семенов В.А.

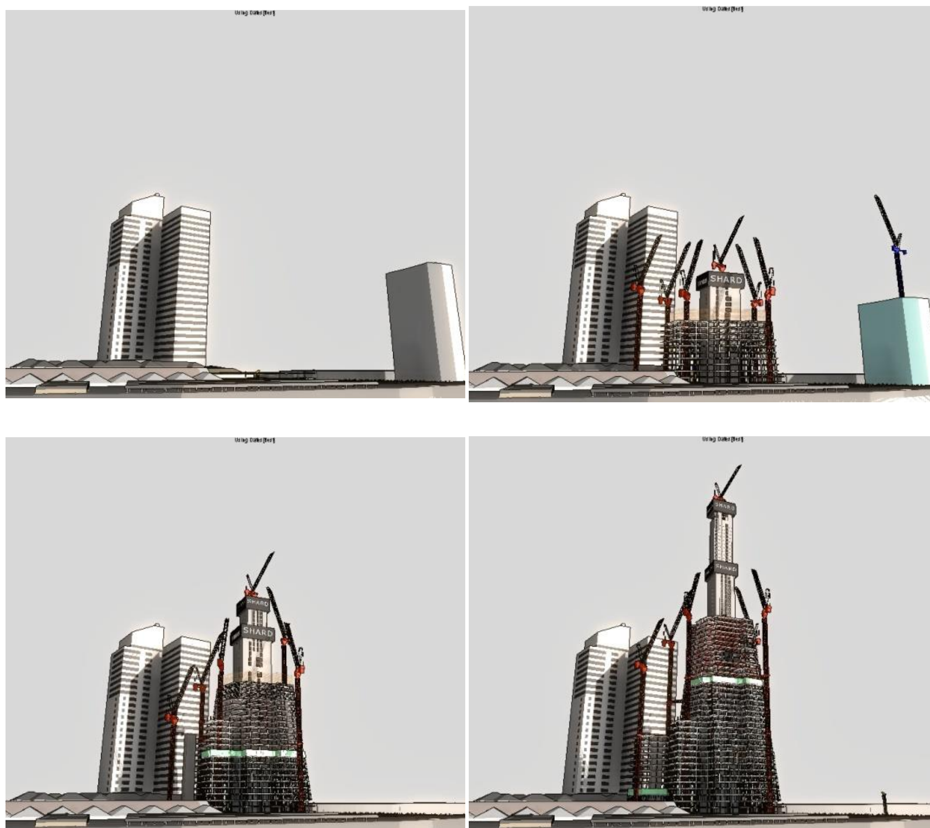
Аннотация. Статья посвящена современным методам поиска и индексации многомерных данных применительно к задачам моделирования больших динамических сцен. Подобные задачи имеют разнообразные приложения в компьютерной графике и анимации, мультимедийных базах данных, системах виртуальной и дополненной реальности, системах автоматизированного проектирования и производства, робототехнике, геоинформационных системах, системах комплексной инженерии и управления проектами, однако их решение часто оказывается невозможным из-за высокой вычислительной сложности алгоритмов пространственно-временной локализации объектов сцен и требует применения специальных схем индексации многомерных данных. В статье обсуждаются два фундаментальных подхода к организации подобных схем, а также проводится их сравнительный анализ в контексте комплексных требований, предъявляемых к приложениям обсуждаемого класса.

1. Введение

В последние годы проблемы управления сложными многомерными данными привлекают все большее внимание как со стороны научного сообщества, так и со стороны производителей прикладного программного обеспечения [1]. Однако, несмотря на обширные исследования в этой области и многочисленные публикации, посвященные, в частности, пространственным и темпоральным СУБД [2], вопросы обеспечения эффективного доступа к многомерным данным с учетом функционала программного приложения и особенностей решаемых прикладных задач остаются открытыми и требуют дальнейшей проработки.

В настоящей работе предпринимается попытка систематизировать современные методы индексации многомерных данных и выделить семейства,

наиболее перспективные для приложений моделирования больших динамических сцен. Рассматриваемый класс приложений чрезвычайно широк и охватывает системы компьютерной графики и анимации, программирование игр, мультимедийные базы данных, системы виртуальной и дополненной реальности, геоинформационные системы (ГИС), системы автоматизированного проектирования и производства (САПР), робототехнику, системы комплексной инженерии, информационные системы управления проектами (ИСУП). На рис. 1 приведен пример большой динамической сцены, возникающей в приложении ИСУП и состоящей из сотен тысяч геометрических объектов. Серия изображений иллюстрирует ход работ по сооружению масштабного, технологически сложного комплекса зданий “Осколок стекла” в центре Лондон-сити.



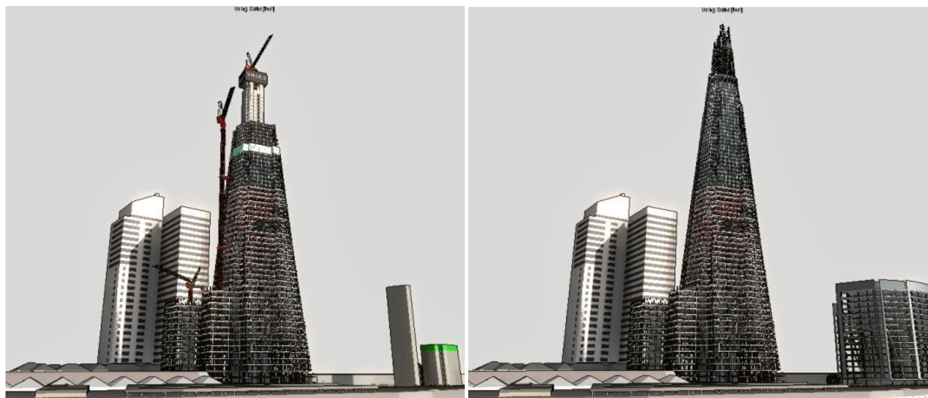


Рис. 1. Пример большой динамической сцены.

Принципиальными особенностями обсуждаемых приложений являются:

— сложность и масштабность сцен, содержащих в себе миллионы объектов с индивидуальными геометрическими и поведенческими характеристиками;

— многообразие геометрических типов данных, представленных в сценах примитивами, сплайнами, неявно заданными алгебраическими кривыми и поверхностями, выпуклыми и невыпуклыми многогранниками, облаками точек, твердотельными конструкциями;

— жесткий характер пространственно-временной когерентности сцен, связанный с сочетанием относительно быстрых и медленных перемещений объектов при их существенно неравномерном пространственном распределении;

— отсутствие строгой детерминированности в динамике сцен и возможная индукция дополнительных случайных событий;

— высокая размерность пространства (более точно, конфигурационного пространства) при моделировании кинематических конструкций с большим числом степеней свободы;

— разнообразие функций визуального моделирования сцен, в частности, поиск объектов, находящихся в заданной области пространства (как правило, имеющей форму параллелепипеда) в заданный промежуток времени; отсечение объектов сцены призмой или конусом видимости; формирование альтернативных геометрических представлений объекта в виде иерархий ограничивающих объемов (bounding volume hierarchies); определение степени детализации геометрического представления объекта (level of details), необходимой для визуально адекватной растеризации сцены; определение пространственных столкновений объектов сцены

(collision detection) на заданном временном интервале; формирование и обновление структур наполненности (occupancy grids) и топологических карт (topological maps) сцены; поиск бесконфликтных путей в псевдо-динамических сценах (motion planning) и т.п.

Перечисленные особенности диктуют довольно жесткие требования к эффективности исполнения типовых запросов и сбалансированности затрат на обновление пространственных индексов при перманентных изменениях в динамической сцене. Примечательно, что вопросы хранения пространственных данных и индексов во внешней памяти часто отходят на второй план, поскольку деградация производительности вычислительных процедур наступает даже в случаях, когда данные целиком помещаются в оперативной памяти. Поэтому основным требованием оказывается эффективность исполнения типовых запросов. В настоящей работе анализируются четыре основных типа запросов, а именно: поиск объектов в заданной области пространства, поиск ближайших соседей в постановках NN–задач (nearest neighbor problem) и *k*NN–задач (*k* nearest neighbor problem), добавление объекта в сцену в заданном положении и удаление объекта из сцены. Примечательно, что многие упомянутые выше функции естественным образом редуцируются к данным запросам при условии, что пространственные индексы обновляются и поддерживаются в согласованном состоянии на каждом временном шаге моделирования динамической сцены. В частности, перемещение объекта в сцене может эмулироваться операцией удаления объекта и последующей его вставкой в новом положении. Определение коллизий объектов может осуществляться путем предварительной локализации соседей и последующего попарного пересечения их граничных представлений [3]. Нужно отметить, что когда динамика сцены полностью предопределена, возникают дополнительные возможности для пространственно-временной индексации событий. В частности, альтернативные схемы подобной индексации, основанные на октальных деревьях, обсуждаются в нашей работе [1].

Естественно, охватить в рамках статьи все известные методы и отобрать наиболее эффективные для обсуждаемого класса приложений не представляется возможным. Поэтому в работе, прежде всего, обсуждаются основополагающие принципы индексации многомерными данных, которые могли бы служить отправной точкой для разработки новых перспективных стратегий управления сложными пространственно-временными данными в приложениях моделирования больших динамических сцен. В разделе 2 мы выделяем два фундаментальных принципа индексирования пространственных данных и на основе них выстраиваем общую классификацию методов. В разделе 3 рассматривается класс методов, в основе которых лежит принцип кластеризации объектов. В разделе 4 анализируются методы индексации, использующие декомпозицию пространства. В заключение проводится их сравнительный анализ и даются рекомендации по их применению.

2. Классификация методов индексации многомерных данных

Хотя арсенал существующих методов поиска и индексации многомерных данных на основе деревьев поиска, хэш-таблиц и пространственных сеток довольно разнообразен, в их основе, как правило, лежат два фундаментальных принципа: кластеризации объектов и декомпозиции пространства. Первый принцип, иногда называемый в иностранной литературе объектными пирамидами (object pyramids), заключается в многоуровневой композиции объектов сцены в кластеры с тем, чтобы обеспечить их плотное покрытие иерархиями ограничивающих объемов и обеспечить быструю пространственную локализацию объектов. Второй принцип (spatial decomposition) предполагает последовательную декомпозицию пространства всей сцены и определение принадлежности каждого объекта той или иной из полученных областей. Каждый из принципов допускает огромное число вариаций структур индексации и алгоритмов заполнения, обновления и поиска, поэтому мы выделяем дополнительные семейства методов.

К методам, реализующим принцип кластеризации объектов, следует отнести

— сбалансированные, хорошо ветвистые деревья, ориентированные на страничный доступ к данным большого объема и хранимым во внешней памяти. Некоторые авторы непосредственно адаптируют для этих целей популярные B [4], B+ [5], B* [6] деревья, устанавливая для объектов отношение линейного порядка, порождаемое их относительным пространственным расположением. Для задания отношения обычно используются кривые построчного, спирального, Z-образного, N-образного, U-образного заполнения пространства, а также известные функции упорядочивания Мортона, Пеано-Гильберта, Грэя и Кантора [7] (см. рис.2). В результате близлежащие объекты помещаются в одну вершину и оказываются на одной странице, что важно при обработке ряда запросов. Тем не менее, выполнение типового запроса — поиска объектов по заданной области остается проблематичным. С этой целью базовые структуры обобщаются до R [8], R+ [9], R* [10] деревьев, которые разбивают пространство на множество иерархически вложенных и, возможно, пересекающихся, прямоугольников

(параллелепипедов в пространственно-трехмерном случае или *n*-мерных политопов в случае более высокой размерности). Каждый прямоугольник является минимально ограничивающим для объектов, ассоциируемых с соответствующей вершиной дерева, что дает возможность ограничить поиск объектов теми вершинами, которые имеют непустое пересечение с областью поиска. Ключевым моментом здесь является выбор техники кластеризации объектов, которая с одной стороны должна обеспечить сбалансированность и наполненность дерева, а с другой стороны — четкую локализацию поиска, обусловленную минимальным перекрытием ограничивающих прямоугольников. Часто это оказывается проблематичным из-за особенностей пространственного заполнения сцены и существенных затрат на кластеризацию и перебалансировку всего дерева. Как обычно, поиск компромисса достигается релаксацией ряда требований и условий;

— метрические деревья, обеспечивающие быстрый поиск соседей в постановках NN– и *k*NN–задач. Линейная сложность наивного поиска часто оказывается неприемлемой для приложений, реализующих вычислительно сложные методы. Особый интерес в связи с этим представляют iDistance-структуры [11], деревья покрытий (cover-tree) [12] и BK-деревья [13]. Например, при формировании iDistance-структур выбирается относительно небольшое количество вспомогательных опорных вершин (обычно в центрах выявленных кластеров объектов), а основные объекты индексируются с использованием в качестве ключа расстояния до ближайшей опорной вершины. При поиске соседей анализу подлежат лишь те объекты, которые лежат в кольце, образованным ближайшей опорной вершиной и имеющим внутренний и внешний радиусы, согласованные с расстоянием до основного объекта на основании правила треугольника. Аналогичным образом реализуются запросы поиска объектов в заданной пространственной области.

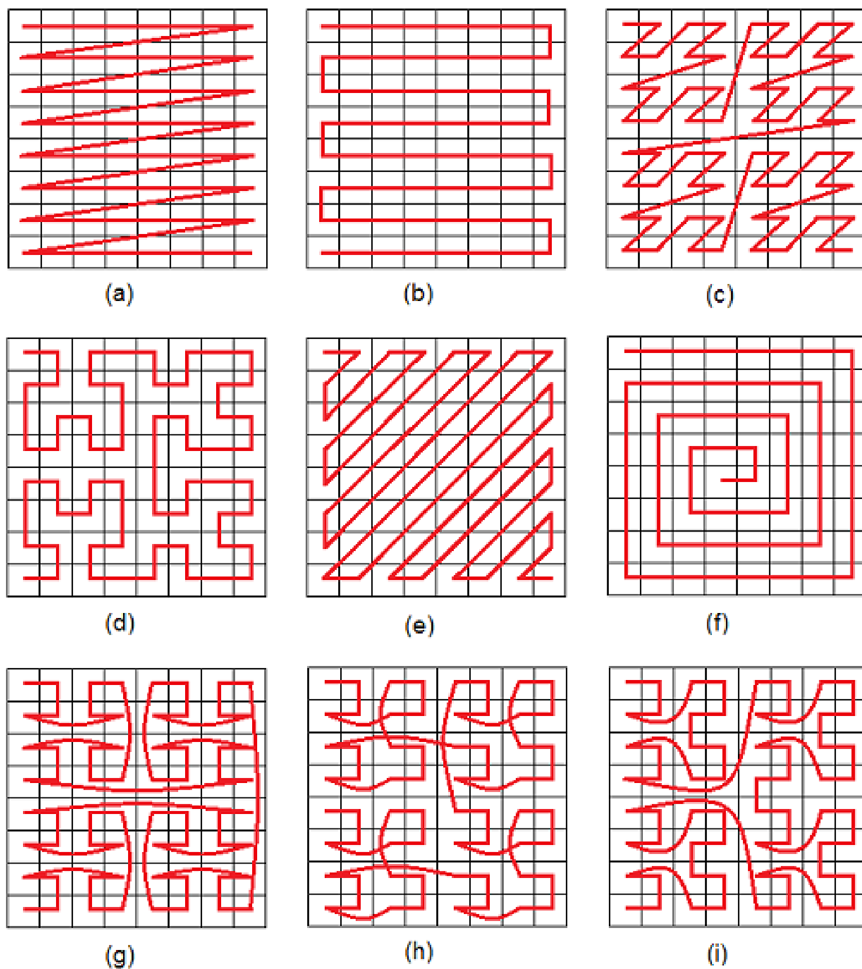


Рис. 2. Кривые упорядочивания пространства. (a) Строчное упорядочивание. (b) Двунаправленное строчное упорядочивание. (c) Z-упорядочивание. (d) Упорядочивание Пиано-Гильберта. (e) Диагональное упорядочивание Кантора. (f) Спиральное упорядочивание. (g) Упорядочивание Грея. (h) Двойное упорядочивание Грея. (i) U-упорядочивание.

Методы, основанные на пространственной декомпозиции, можно условно подразделить на следующие семейства:

— Структуры, обеспечивающие быстрый поиск на интервалах, такие как деревья диапазонов (range trees) [14], сегментов (segment trees) [15],

интервалов (interval trees) [16], деревья приоритетного поиска (priority search trees) [17]. Данные структуры естественным образом обобщаются на случай произвольной размерности, однако редко применяются в приложениях с размерностью выше двух из-за значительных затрат по памяти.

— Бинарные деревья пространственной декомпозиции, к самому известному варианту которых следует отнести BSP-деревья (binary spatial partitioning) [18], [19]. Деревья этого типа предполагают разбиение произвольными одиночными плоскостями, что широко используется в приложениях компьютерной графики для упорядочивания фрагментов сцен и удаления невидимых элементов в зависимости от местоположения камеры. Однако вряд ли они могут использоваться в качестве универсального пространственного индекса. Известные попытки применить полосы, параллелепипеды и даже обобщенные политопы в качестве пространственных элементов разбиения (как это реализуется, например, в LSD (local split decision tree) [20], HB (holey brick tree) [21] и D-деревьях [22] соответственно), оказались не совсем удачными в силу неоправданного усложнения вычислений. Более востребованными представляются kD-деревья [23], допускающие последовательное разбиение ортогональными плоскостями и являющиеся весомой альтернативой интервальным структурам при более высокой размерности пространства. Выбор переменной пространства, относительно которой осуществляется разбиение, и сам способ построения плоскости существенно различаются в многочисленных адаптивных (adaptive kD) [24], гибридных (hybrid kD-B) [25] и обобщенных (generalized k+D) версиях этой структуры. Для обеспечения сбалансированности деревьев успешно применяются техника плавающей медианы в SMP (sliding-midpoint) структурах [26], техника аппроксимации медианы в VAMSplit (varience approximate median split) деревьях [27] и техника выделения кластеров, как это реализуется в BBD (balanced box decomposition) деревьях [28]. Версии DQS (dynamically quantized space) [29], DQP (dynamically quantized pyramids) деревьев [30] учитывают наполненность вершин и в большей степени ориентированы на страничную подгрузку фрагментов сцен.

— Регулярные сетки, обычно применяемые в пространственно-трехмерном случае и имеющие ячейки в виде параллелепипедов, тетраэдров, призм с треугольниками или шестиугольниками в основании. Сопоставляя объекты сцены с ячейками сетки, с которыми они имеют непустое пересечение, можно сформировать необходимый пространственный индекс. Иногда для этого применяют, так называемые, сеточные файлы (grid directory files) [31], которые допускают возможность ассоциирования нескольких ячеек с одним объектом, но исключают привязку нескольких объектов к одной ячейке. Существует ряд работ, в которых для этих целей используют хэш-таблицы на основе известных техник OPLH (order preserving linear hashing),

LHRBI (linear hashing with reversed bit interleaving), MDEH (multidimensional extendible hashing), LHPE (linear hashing with partial expansions), PLOP (piecewise linear order preserving), квантильного (quantile hashing) и спирального (spiral hashing) хэширования [32]. На наш взгляд они предпочтительны для обсуждаемых приложений, поскольку позволяют ассоциировать несколько объектов сцены с каждой ячейкой и, тем самым, непосредственно разрешать запросы пространственной локализации.

— Многоуровневые сетки, в большинстве приложений представленные квадрантными (quadtrees) и октальными (octrees) деревьями, и реже — вложенными тетраэдральными и гексагональными блоками. Как правило, в данных иерархически организованных сетках применяется рекурсивное разбиение пространства на конгруэнтные или подобные ячейки с простыми вычислительными правилами определения пересечений с объектами сцены и областями поиска. Алгоритмы поиска используют эти правила для принятия решения о необходимости распространения анализа в дочерние ячейки. Тем самым, большинство ячеек исключаются из рассмотрения, а время исполнения запроса локализации определяется размером области поиска и глубиной представления многоуровневой сетки в ней. Заметим, что обсуждаемое семейство методов существенно шире и представлено также многочисленными вариантами ортогональных многоуровневых сеток. Например, в обобщенных ATree структурах допускается разбиение D -мерного пространства на переменное число дочерних ячеек 2^d, $1 \le d \le D$, объединяющих смежные D-мерные кубы [33]. Способ разбиения выбирается индивидуально для каждой ячейки с целью наиболее компактно локализовать объекты сцены. В случаях, когда значительная доля объектов пересекается с внутренними вершинами, ребрами или гранями и должны быть приписаны нескольким смежным ячейкам, ATree структуры получают ряд преимуществ. Варианты разбиений, используемые в близких BinTree [34], AHC (adaptive hierarchical coding) [35], X-Y [36], Puzzle деревьях [37] и Treemap структурах [38], предусматривают циклическую и ациклическую смену осей пространственного разбиения на каждом уровне иерархии. Данные структуры лишены существенного недостатка kD деревьев, проявляемого при необходимости разбить ячейку несколькими параллельными гиперплоскостями и приводящего к большей глубине дерева поиска за счет вставки дополнительных вершин. Развертывание перечисленных структур происходит приблизительно одинаково путем последовательной вставки объектов в сцену и инкрементальной коррекции многоуровневого представления сетки. Процедура разбиения пространства применяется рекурсивно до тех пор, пока не достигнуты требуемые условия на листьях дерева. Например, в приложениях моделирования сцен процедура завершается при невозможности более четко локализовать объекты в листовых вершинах или при их относительно небольшом количестве. В

приложениях пространственного анализа (spatial reasoning) с использованием октальных деревьев наполненности (occupancy octrees) обычно используются условия достижения полностью занятых или полностью пустых ячеек, естественно, с заданной точностью представления границ объектов и допустимой ошибкой определения пересечений.

В следующих разделах остановимся более подробно на характерных представителях каждого выделенного класса методов и особенностях реализации типовых запросов в них.

3. Методы, основанные на кластеризации объектов

Принцип кластеризации предполагает многоуровневую композицию объектов в виде сбалансированного дерева, листья которого соответствуют отдельным объектам, нелистовые вершины — выделенным кластерам, а корень дерева — всей сцене. С каждой нелистовой вершиной ассоциируется $[m, M]$ дочерних вершин, где фиксированные параметры $m \geq 1$, $M \geq m$ ограничивают их число снизу и сверху. Структуры данного семейства являются развитием B-деревьев, широко применяемых в СУБД общего назначения для хранения во внешней памяти со страничным доступом, однако предполагают особые способы пространственной кластеризации. Так, например, гильбертово R-дерево [39] является B+-деревом, в котором для кластеризации объектов применено упорядочивание Пиано-Гильберта (см. рис. 2).

Параметр M обычно выбирается, исходя из размера страницы файловой системы. Чем больше его значение, тем более кустистым и менее глубоким будет полученное дерево, что повышает эффективность пространственной локализации объектов. Этими же соображениями объясняется интерес к заполненным (packed) структурам. Формирование подобных структур может происходить как снизу вверх, так и сверху вниз. В первом случае объекты рекурсивно объединяются на каждом уровне дерева таким образом, что каждая нелистовая вершина содержит ровно M дочерних. Такая схема используется, в частности, в упакованных гильбертовых R-деревьях (packed Hilbert R-tree) [40]. Во втором случае объекты сцены рекурсивно разбиваются на M групп с одинаковым количеством объектов в каждой до тех пор, пока в группе не окажется менее M объектов. Такая схема иногда применяется в STR-методе (sort tile recurse method) [41]. Важно отметить, обе схемы предполагают использование дополнительных пространственных соображений при кластеризации объектов и обеспечении надлежащей сбалансированности деревьев.

Тем не менее, практическое применение заполненных структур довольно ограничено при наличии динамики в сценах и необходимости обновлений,

часто сводящихся к полному перестроению дерева. Для приложений, оперирующими статическими и квазистатическими сценами, как, например, ГИС, использование заполненных структур вполне оправданно. Пример подобной структуры с параметром заполнения $M = 3$ для сцены, состоящей из 9 объектов, приведен на рис. 3.

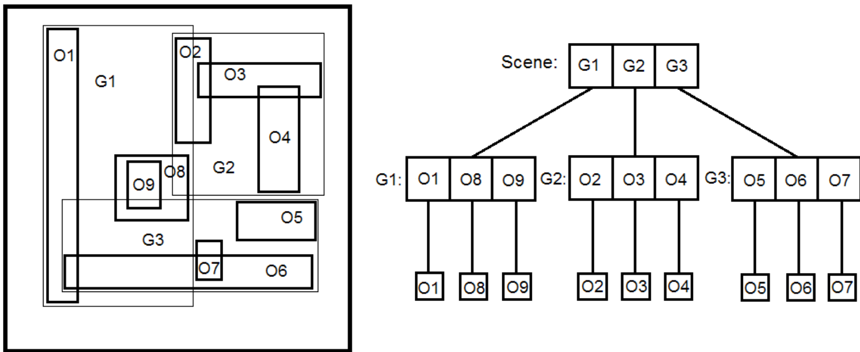


Рис. 3. Пример заполненной структуры параметром заполнения $M = 3$.

Для более динамичных сцен на практике применяются структуры с нижней границей заполнения $m = 0.3 - 0.4M$. Такое значение порога, установленное эмпирически в ряде экспериментов, несколько ниже, чем величина $m = 0.5M$, обычно применяемая для B-деревьев, и величина $m = 0.66M$, устанавливаемая для B*-деревьев. Уменьшение нижнего порога объясняется значительными вычислительными затратами на пространственную кластеризацию объектов, которые, тем не менее, могут быть частично скомпенсированы релаксацией требований наполненности и отложенными операциями перебалансировки дерева.

Для непосредственного снижения затрат на пространственную кластеризацию и локализацию объектов обычно привлекают методы ограничивающих объемов, которые составляют традиционные главы вычислительной геометрии. Данные методы предусматривают использование альтернативных геометрических представлений объектов, обычно основанных на упрощенной аппроксимации их границ. С их использованием удается заменить вычислительно сложные процедуры определения взаимных пересечений объектов аналогичными тестами для ограничивающих объемов и, тем самым, выносить быстрый отрицательный вердикт в тех случаях, когда объемы действительно не пересекаются.

Чаще всего для этой цели применяют описанные вокруг объекта прямоугольные параллелепипеды с ребрами параллельными координатным осям (AABB от английского axis-aligned bounding box). Известны попытки использования описанных сфер, цилиндров, выпуклых оболочек,

многогранников с заданным количеством возможных ориентаций граней k-DOP (k-discrete orientation polyhedron) [42] (используемый, в частности в cell-tree [43]), произвольно ориентированных параллелепипедов (OBB от oriented bounding box) [44] . Однако использование более точной аппроксимации неизбежно приводит к более высоким вычислительным затратам. Так пересечение двух AABB параллелепипедов требует всего 6 операций сравнения, в то время как пересечение двух OBB параллелепипедов требует от 89 до 252 операций в лучшем и худшем случае соответственно [45]. Поэтому применение вычислительно простых методов представляется наиболее оправданным для обсуждаемого класса приложений. В дальнейшем мы будем полагать, что объекты сцены и сформированные на их основе кластеры представляются своими ограничивающими объемами так, что с каждой вершиной дерева композиции объектов ассоциирован соответствующий AABB параллелепипед.

Способ пространственной кластеризации является ключевым алгоритмическим элементом, определяющим эффективность поиска на основе индексированных иерархических структур. Чтобы проиллюстрировать этот факт, рассмотрим типовой запрос пространственной локализации — поиск объектов, имеющих непустое пересечение с заданной областью пространства. Исполнение такого запроса сводится к обходу вершин дерева, чьи ограничивающие объемы пересекают заданную область, сверху вниз. Если выявлено пересечение для некоторой вершины дерева, то обход распространяется на всех ее детей. В наихудшем случае, когда область поиска охватывает всю сцену, неизбежному анализу подлежат все вершины дерева. Естественно, что кластеры объектов могут формироваться на основе условий естественной пространственной декомпозиции сцены, а могут следовать и иным требованиям. В первом случае удается существенно сократить количество анализируемых вершин и время исполнения типовых запросов. В качестве таких условий обычно используют критерий минимального покрытия, предусматривающий минимизацию суммарного ограничивающего объема для формируемого кластера объектов, и критерий минимального перекрытия, препятствующий значительному взаимному пересечению выделенных кластеров. На первый взгляд, оба критерия обеспечивают компактное пространственное представление для каждого кластера объектов. Тем не менее, критерии не эквивалентны, что демонстрирует пример, представленный на рис. 4. Результат кластеризации на основе критерия минимального перекрытия (см. рис. 4а) существенно отличается от аналогичного результата на основе критерия минимального покрытия (см. рис. 4б). Хотя первый критерий приводит к более рациональной пространственной кластеризации, спекулятивный комбинаторный анализ перекрытий оказывается довольно затратным и при практической реализации R, R+, R*-деревьев чаще применяется второй критерий, основанный на минимизации покрытия. Иногда используются комбинированные критерии, основанные на взвешенной оценке объемов покрытия и перекрытия [46], [47].

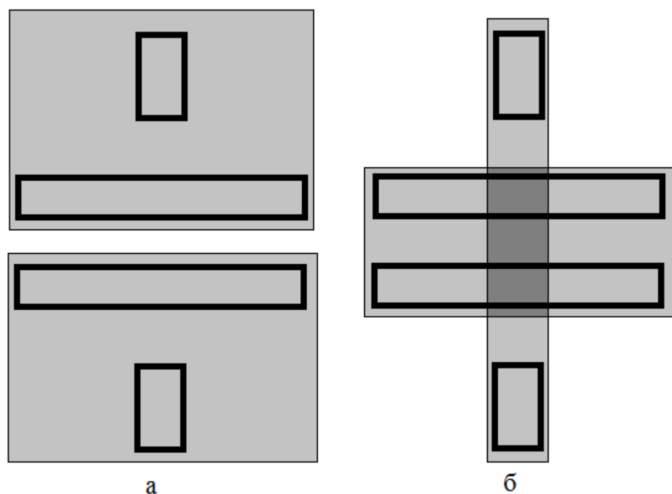


Рис. 4. а) Результат кластеризации объектов на основе минимального перекрытия. б) Результат кластеризации объектов на основе минимального покрытия.

Задача о минимальном покрытии носит комбинаторный характер, поэтому вычислительная сложность наивного алгоритма – $O(2^M)$, где M – количество объектов, которые необходимо сгруппировать. Однако разработаны эффективные методы, решающие задачу о минимальном покрытии за существенно меньшее время. В работе [48] описан метод, который в пространственно двумерном и трехмерном случае приводит к более оптимистическим оценкам сложности $O(M^3)$ и $O(M^5)$ соответственно. В общем случае асимптотическая оценка сложности этого метода составляет $O(dM \log M + d^2 M^{2d-1})$, где d — размерность пространства. В работе [49] предложен метод с лучшей оценкой сложности $O(M^2)$ для двумерного случая, однако его применение не гарантирует балансировки структуры пространственного индекса. Оба упомянутых метода допускают параметризацию с помощью монотонной функции оценки группового покрытия и обеспечивают поиск точного решения задачи. Ряд авторов констатирует, что вычислительные затраты на точный поиск минимального покрытия не всегда оправданы в контексте общей проблемы пространственной кластеризации и эвристические подходы, в частности, метод разбиения пространства гиперплоскостями [50] и метод группирования по ключевым объектам, приводят к вполне удовлетворительным результатам [49].

Рассмотрим более подробно метод разбиения пространства гиперплоскостями. Идея метода состоит в делении пространства сцены плоскостями, перпендикулярными одной из осей. Принадлежность объекта тому или иному кластеру определяется его положением относительно секущих плоскостей, а именно, максимальной координатой ограничивающего параллелепипеда вдоль выбранной оси разбиения. Плоскости проводятся таким образом, чтобы обеспечить наиболее равномерное распределение объектов по кластерам. Для этой цели объекты сортируются в порядке возрастания максимальной координаты ограничивающего параллелепипеда и последовательно объединяются в необходимое число кластеров. Аналогичные попытки предпринимаются для каждой из осей, а окончательное разбиение выбирается, исходя из критериев равномерности распределения объектов по кластерам, минимизации покрытия или минимизации перекрытия в соответствии с назначенными приоритетами. Анализ сложности метода приводит к оценке $O(dM)$ при размерности пространства d. Результаты экспериментов показывают [50], что метод демонстрирует хорошие результаты для сцен с равномерно распределенными объектами.

Следует отметить, что близкий метод используется для минимизации перекрытия в R*-деревьях. Однако выбор положения плоскости разбиения происходит, исходя из условия минимальности среднего периметра ограничивающих параллелепипедов сформированных кластеров. Такой метод требует более высоких вычислительных затрат с оценкой $O(M \log M)$, однако несколько уменьшает расходы на хранение структур индексации [10], [51].

Интересную альтернативу составляют эвристики, которые реализуют инкрементальную технику формирования кластеров объектов. Вначале выбирается несколько характерных объектов, каждый из которых образует свой собственный кластер. Затем остальные объекты приписываются к одному из них в соответствии с критерием минимизации покрытия. При образовании кластеров следует выбирать наиболее удаленные и протяженные объекты. Для этого обычно применяются условие максимальности объема параллелепипеда, описанного вокруг пары образующих вершин, и условие максимальной разницы в одной из координат объектов, нормированной на ширину ограничивающего параллелепипеда. При использовании первого условия пара образующих вершин может быть выбрана за $O(M^2)$ операций, при применении второго — за $O(M)$ операций, что существенно улучшает ситуацию с общими затратами на кластеризацию. Вычислительные эксперименты показывают, что приведенные условия выбора образующих объектов незначительно влияют на итоговое время исполнения типовых запросов [8], поэтому применение второго условия, безусловно, выглядит предпочтительнее для практических реализаций. На рис. 5 приведен пример сцены из 9 объектов и R-дерево с параметрами наполненности $m = 2$,

$M = 3$, построенное с помощью инкрементальной кластеризации на основе данного условия.

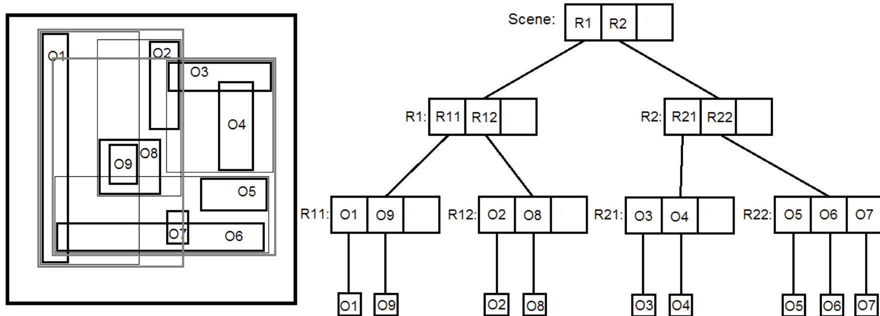


Рис. 5. Пример сцены из 9 объектов и R-дерево с параметрами наполненности $m = 2$, $M = 3$, построенное с помощью инкрементальной кластеризации.

Наличие динамики в сцене может существенно поменять оценку рассмотренных методов кластеризации, поскольку появление в сцене новых объектов, удаление или перемещение существующих объектов вызывает необходимость перманентных обновлений структур индексации. Поскольку подобные события неизбежно влекут изменения в площади покрытия и перекрытия ограничивающих параллелепипедов (причем эти изменения могут зависеть как от очередности событий в сцене, так и от их пространственного распределения), эффективность исполнения запросов и производительность программного приложения могут существенно деградировать в сценах с интенсивной динамикой.

Один из способов решения этой проблемы состоит в сохранении сбалансированности дерева, даже если события происходят в одной и той же области пространства и влекут переполнение или исчерпание соответствующих вершин дерева. При переполнении вершины проводится ее пространственное расщепление и часть объектов переносится в соседнюю свободную и неизмененную на данной транзакции вершину того же уровня. Если соседняя вершина переполнена или изменена, то предпринимается попытка перенести их в следующую соседнюю вершину, а если переполнены все соседи — то она вставляется заново в верхнюю родительскую вершину.

Данный способ, называемый повторной вставкой (forced reinsertion), широко применяется при реализации R*-деревьев. При переполнении одной из вершин часть ее объектов (обычно 30% от максимально допустимого заполнения [10]) поочередно вставляются в дерево. Для повторной вставки выбираются объекты наиболее удаленные от центра ограничивающего объема

расщепляемой вершины. Как было отмечено выше, очередность событий может существенно влиять на организацию дерева и эффективность исполнения запросов. Поэтому на практике, чтобы придать некоторый детерминизм данной процедуре, следуют одной из двух дисциплин, предусматривающий порядок повторной вставки объектов, начиная от самых дальних к ближним (far-reinsert) или от самых ближних к дальним (close-reinsert). Чтобы избежать попыток повторного включения объекта в саму расщепляемую вершину или в соседние переполненные вершины, применение процедуры ограничивается однократной попыткой на каждом уровне дерева.

Несмотря на многочисленные подходы к пространственной кластеризации и обеспечению сбалансированности структур индексации, область их практического применения в случае динамических сцен довольно ограниченна из-за значительных вычислительных затрат. Для их содержательной адаптации к динамическим сценам были предложены такие специальные структуры, как параметрические R-деревья [52], TPR-tree [53] и TPR*-tree [54]. Основная идея, лежащая в их основе, заключается в использовании описываемых объемов движущихся объектов. Такие объемы могут быть насчитаны на любом заданном временном интервале с учетом всех промежуточных положений объекта и применяться при пространственной кластеризации сцены и разрешении запросов на данном интервале. Естественно, что использование подобной аппроксимации динамического объекта существенно расширяет границы его локализации в любой фиксированный момент времени и неизбежно приводит к дополнительным расходам. По-видимому, разумный компромисс достигается в случае, если временной интервал моделирования динамической сцены разбит на участки, на каждом из которых применяется своя структура индексации с учетом объемов, описываемых движущимися объектами локально на каждом отдельном участке. К сожалению, применимость данных методов ограничена сценами с предопределенным характером динамики, когда положения всех объектов в любой временной точке моделирования известны заранее.

Обсудим вопросы реализации типовых операций с использованием структур объектной кластеризации. Псевдокод операции поиска объектов в заданной пространственной области представлен на рис. 6а.

```
procedure FIND_OBJECTS_IN_VOLUME(T,V,R)
pointer node T
pointer volume V
pointer collection R

if(IS_INTERSECTING(VOLUME(T), V ) )
        {
        if(IS_LEAF(T))
                {
                ADD(R,T)
```

```
                return
                }

        for_each(node child in CHILDREN(T))
                FIND_OBJECTS_IN_VOLUME(child, V, R)
        }
```

Рис. 6а. Псевдокод операции поиска объектов в заданной области.

Приведенный алгоритм осуществляет рекурсивный обход дерева сверху вниз, начиная с корневой вершины. Функция IS_INTERSECTING возвращает статус принадлежности вершины заданному объему. В случае, когда вершина не принадлежит заданному объему, обход приостанавливается, поскольку все дочерние вершины априори не попадают в заданную область, иначе статус ее принадлежность листовым вершинам устанавливается при помощи процедуры IS_LEAF. Если вершина является листовой, то она ассоциируется с объектом сцены и этот объект добавляется к результатам запроса, в противном случае процедура вызывается для всех ее дочерних вершин. Вычислительная сложность этого алгоритма составляет $O(N)$ в худшем случае, поскольку каждая вершина дерева пересекается с заданным объемом и для выполнения запроса необходимо проанализировать все объекты.

На рис. 6б приведен набор функций, осуществляющий поиск ближайшего соседнего объекта.

```
procedure FIND_NEAREST_NEIGHBOUR(T,R)
pointer node T
pointer collection R

pointer node P=PARENT(T)
float D = ∞
return FIND_NEAREST_NEIGHBOUR_IN_ANCESTORS(P,T,T,D,R)


procedure FIND_NEAREST_NEIGHBOUR_ANCESTORS(T,C,N,D,R)
pointer node T
pointer node C
pointer node N
pointer float D
pointer collection R

for_each(node child in CHILDREN(T))
        {
        if(child != C)
                {
                FIND_NEAREST_LEAF(child,N,D,R)
```

```
                }
        }
    if(IS_ROOT(T))
        return

    return
FIND_NEAREST_NEIGHBOUR_ANCESTORS(PARENT(T),T,N,D,R))

    procedure FIND_NEAREST_LEAF(T,N,DST,R)
    pointer node T
    pointer node N
    pointer float D
    pointer collection R

    float distance = DISTANCE(T,N)
    if(distance>DST)
        return

    if(IS_LEAF(T))
        {
        if(distance<DST)
            {
            CLEAR(R)
            DST = distance;
            }
        ADD(R,T)
        return
        }

    for_each(node child in CHILDREN(T))
        {
        FIND_NEAREST_LEAF(child,N,DST,R)
        }
```

Рисунок 6б. Псевдокод алгоритма поиска ближайших объектов.

Функция FIND_NEAREST_NEIGHBOUR принимает в качестве аргумента объект и возвращает коллекцию ближайших соседей. Для этого используется вспомогательная функция FIND_NEAREST_NEIGHBOUR_IN_ANCESTORS, принимающая в качестве аргументов текущую вершину дерева T, предыдущую вершину дерева C, вершину объекта N, а также найденное минимальное расстояние DST и соответствующий ему результат в виде коллекции соседних объектов R. Она осуществляет рекурсивный поиск

ближайших объектов в текущей вершине, всех ее дочерних вершинах и затем распространяет поиск на родителей. Для корректной работ алгоритма переменные устанавливаются следующим образом: DST=∞, R=Ø. Поиск ближайших объектов среди дочерних вершин осуществляется с помощью вспомогательной процедуры FIND_NEAREST_LEAF. Существенно, что рекурсивный обход в ней прекращается в случае, если расстояние до текущей вершины больше, чем найденное ранее расстояние до ближайшего объекта. В наихудшем случае для выполнения запроса придется пройти все вершины дерева, поэтому вычислительная сложность алгоритма составляет $O(N)$.

Несмотря на многообразие подходов к кластеризации объектов, ограничимся при дальнейшем рассмотрении классическим R-деревом. Оценки для других подобных структур этого семейства, таких как R*-дерево, оказываются значительно выше.

```
procedure INSERT(T,O,M)
pointer node T
pointer object O
integer M

if(STORE_OBJECTS(T))
        {
        ADD(T,O)
        if(NUM_STORED(T) > M)
                SPLIT(T,M)
        return
        }

return INSERT(FIND_BEST_CHILD(T,O),O,M)


pointer node procedure FIND_BEST_CHILD(T,O)
pointer node T
pointer object O

pointer node result;
float delta = ∞
for_each(node child in CHILDREN(T))
        {
        volume selfVolume = VOLUME(child)
        volume volumeWithObject = EXPAND(selfVolume,
VOLUME(O))

        if(volumeWithObject – selfVolume < delta)
                result=child
```

```
        }
return result


procedure SPLIT(T,M)
pointer node T
integer M

pointer node P = PARENT(T)
if(P==NULL)
        P=CREATE_NEW_ROOT()
else
        REMOVE_FROM_PARENT(P,T)
FIND_GROUPS(T,L,R)
ADD(P,L)
ADD(P,R)
if(NUM_STORED(P)>M)
        return SPLIT(P,M)

return
```

Рисунок 6в. Псевдокод операции вставки объекта в R-дерево.

На рис. 6в. приведен псевдокод процедуры INSERT, которая вставляет объект O в R-дерево с параметром заполнения M. Для этого статус текущей вершины проверяется при помощи функции STORE_OBJECTS, которая возвращает логическое значение TRUE, если в вершине хранятся ссылки на объекты, и FALSE в противоположном случае. Если функция STORE_OBJECTS возвращает TRUE, выполняется вставка нового объекта и, если не нарушается верхняя граница наполненности вершин дерева, то процедура завершается. В противном случае объекты, содержащиеся в вершине, разбиваются с помощью вспомогательной процедуры FIND_GROUPS на две группы, которые и формируют новую пару вершин. Существенно, что при этом родительская вершина может оказаться переполненной и эту процедуру следует повторить на верхнем уровне иерархии. Если функция STORE_OBJECTS возвращает FALSE, то с помощью вспомогательной функции FIND_BEST_CHILD ищется дочерняя вершина, в которую будет вставлен объект. Приведенный псевдокод реализует критерий минимального покрытия, предполагающий поиск дочерней вершины, вставка в которую приведет к минимальному увеличению ее объема. В наихудшем случае каждая из пройденных вершин окажется переполненной и подлежит последующей перегруппировке, тогда сложность операции вставки будет составлять $O(M \log N)$ в предположении, что для кластеризации объектов

использовалась рассмотренная выше процедура с вычислительной трудоемкостью $O(M)$.

```
procedure REMOVE(T,m,M)
pointer node T
integer m
integer M

pointer node P = PARENT(T)
REMOVE_FROM_PARENT(P,T)

if(IS_ROOT(P))
        return

if(NUM_STORED(P)>=m)
        return

pointer node PP = PARENT(P)
REMOVE_FROM_PARENT(PP,P)
for_each(node child in P)
        {
        pointer node NP = FIND_BEST_CHILD(PP,child)
        ADD(NP,node)
        if(NUM_STORED(NP) > M)
                SPLIT(NP,M)
        }

if(NUM_STORED(PP)<m)
        REMOVE(PP,m)

return
```

Рисунок 6д. Псевдокод операции удаления объекта из R-дерева.

Рассмотрим процедуру удаления объекта из R-дерева, псевдокод которой приведен на рис. 6д. В качестве параметра она принимает удаляемую вершину T, а также параметры заполнения дерева m и M. В теле процедуры выполняется удаление ссылки на удаляемую вершину из ее родителя. В случае, когда число объектов, оставшихся в родительской вершине, лежит в интервале $[m,M]$, процедура завершается. В противном случае родительская вершина удаляется, а ее объекты перемещаются в смежные вершины. В случае их переполнения осуществляется рекурсивное обновление дерева, как при вставке объекта. В наихудшем случае вычислительная сложность операции удаления составит $O(M \log N)$.

4. Методы, основанные на декомпозиции пространства

Структуры индексации, обсуждаемые в настоящем разделе, реализуют фундаментальный принцип декомпозиции пространства. Следуя данному принципу, каждый объект сцены ассоциируется с одной или несколькими ячейками пространства в зависимости от его положения. Способ ассоциирования, при котором объект содержит ссылки на ячейки пространства, называется в англоязычной литературе явным (explicit). Данный способ позволяет быстро установить ячейки пространства, в которых находится заданный объект, и эффективно выполнить ряд операций, например, удалить объект из сцены без каких-либо обновлений структур индексации. Однако выполнение других запросов, таких как, поиск объектов в заданной области пространства или поиск соседей, становится трудоемким, поскольку требуется проанализировать положение всех объектов в сцене перед тем, как сформировать окончательный результат. Поэтому для обсуждаемого класса приложений более перспективным представляется неявный (implicit) способ ассоциирования, предполагающий, что ячейки пространственного разбиения хранят ссылки на объекты, которые в них содержатся. В этом случае пространственные запросы разрешаются более естественным образом на основе анализа относительного положения ячеек разбиения. Проблема установления ячеек, принадлежащих заданному объекту, также не является критичной, если объекты сцены представлены своими ограничивающими объемами (AABB, OBB, k-DOP) и существуют быстрые вычислительные процедуры локализации этих объемов в ячейках разбиения.

В зависимости от способа задания формы ячеек пространства выделяют полигональные и аналитические разбиения. При полигональном разбиении ячейки пространства представляются многогранниками, в то время как аналитическое разбиение предполагает использование алгебраических уравнений для описания их формы. На практике получили распространение полигональные, топологически регулярные разбиения, ячейки которых геометрически подобны или даже конгруэнтны друг другу.

Чтобы контролировать глубину иерархических разбиений, допустимый размер ячеек обычно ограничивается заданным параметром. Ячейки минимального размера называются единичными. В зависимости от формы единичных ячеек пространственные разбиения могут обладать различными свойствами [55]. Определенное распространение получила сотовая структура (septree) [56], единичные ячейки которой имеют форму правильных шестиугольников. Для ландшафтных сцен, двумерные объекты которых располагаются на некоторой сфере, известны случаи применения в качестве единичных ячеек треугольников. Поверхность сферы в этом случае аппроксимируется икосаэдром, грани которого иерархически разбиваются на треугольные ячейки, как показано на рис. 8.

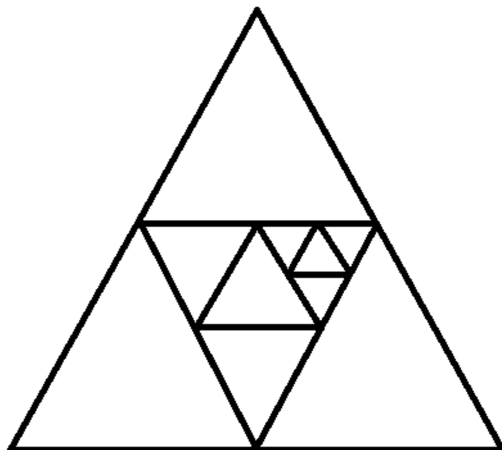


Рис. 8. Иерархическая декомпозиция двумерной области пространства треугольными ячейками.

Пространственные разбиения, использующие одни и те же геометрические построения, могут допускать различные структуры доступа к своим ячейкам (data access structures). В качестве таких структур могут применяться многомерные массивы ссылок, хэш-таблицы с соответствующими функциями пространственного хэширования и деревья. Остановимся более подробно на их особенностях.

Преимущества многомерных массивов проявляются в тех случаях, когда может быть определена функция явного преобразования пространственных координат в соответствующие индексы массива ссылок. При наличии подобной функции быстро осуществляется локализация объектов в заданной области, однако поиск соседей затруднен из-за необходимости организации процедуры распространения по смежным ячейкам, что может быть затратным с вычислительной точки зрения. Другой недостаток массивов связан с необходимостью хранения в памяти записей для всех ячеек пространства, полученных при разбиении, независимо от наличия в них объектов сцены. Для разреженных сцен, в которых объекты распределены неравномерно, это может приводить к существенным накладным расходам по памяти.

Для устранения данного недостатка структура доступа должна быть организована таким образом, чтобы исключать использование пустых ячеек. Определяется порядок обхода ячеек, например, с использованием кривых заполнения пространства, и непустые ячейки нумеруются в соответствии с ним. Установленный порядок позволяет использовать традиционные структуры поиска, такие как AVL-деревья [57], красно-черные деревья [58], B-деревья, хэш-таблицы [59], а присвоенный номер служит ключем.

Поиск объектов в заданной области сводится к поиску экстремальных ячеек, принадлежащих области и имеющих минимальный и максимальный порядковые номера. Ячейки с номерами, лежащими в данном диапазоне, находятся в результате запроса к соответствующей структуре поиска, и далее индивидуально подвергаются точной проверке принадлежности заданной области. Поиск ближайших соседей также реализуем на основе рассмотренных структур доступа [60].

Однако более перспективным для решения обсуждаемого круга проблем нам представляется использование деревьев, вершины которых ассоциированы с ячейками соответствующих уровней пространственного иерархического разбиения. Терминальным условием для роста дерева в глубину выступает ограничение минимального количества объектов, содержащихся в каждом листе. Так в случае применения иерархической декомпозиции пространства размерности d на 2^d равных частей плоскостями, перпендикулярными каждой из координатных осей, получается обобщенное октальное дерево. В трехмерном случае каждая нелистовая вершина октального дерева содержит восемь дочерних вершин, ассоциированных с соответствующими октантами пространственного разбиения. Верхняя вершина дерева соответствует AABB параллелепипеду всей сцены. Пример пространственной декомпозиции и полученного октального дерева приведен на рис. 8.

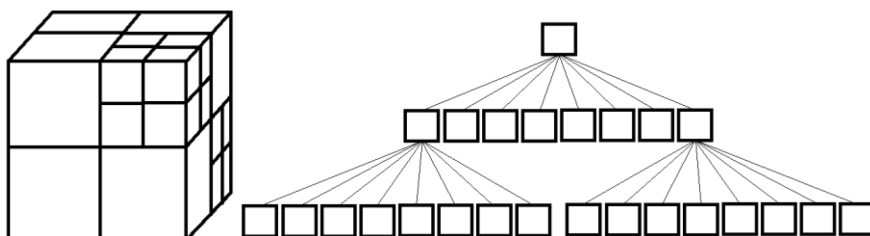


Рис. 8. Пример пространственной декомпозиции и соответствующего ей октального дерева.

Главным достоинством октальных структур является их простота, обусловленная априори известным положением секущих плоскостей и, как следствие, исключением дополнительного пространственного анализа при проведении разбиения. Основным недостатком октальных структур является несбалансированность дерева в случае неравномерного распределения объектов по сцене. Поэтому время исполнения типовых запросов может существенно варьироваться в зависимости от области сцены, подлежащей анализу. Другим недостатком является обязательное хранение восьми дочерних вершин даже в случае, если объекты локализуются только в одной их них.

Проблема избыточности представления отчасти решена в ATree структурах, в которых предполагается выбор приоритетных осей разбиения в каждой вершине дерева. Правильный выбор осей может значительно уменьшить общее количество вершин дерева. В деревьях Bintree данная проблема решается путем бинарной декомпозиции, основанной на циклическом выборе осей разбиения. В случае произвольного выбора данное дерево обобщается до AHC структуры (adaptive hierarchical coding). Если в Bintree деревьях сохранить цикличность выбора осей, но допустить выбор положения секущих плоскостей, то достигается обобщение до kD-деревьев. Существенной чертой данных структур является возможность обеспечить сбалансированность дерева. В обобщенных kD-деревьях снимается условие цикличности, а в X-Y деревьях, treemap структурах и puzzle деревьях допускается разбиение несколькими параллельными секущими плоскостями в каждой вершине. Тем самым, достигается сбалансированность дерева и наполненность вершин. Однако это может быть достигнуто в результате трудоемкого анализа расположения объектов и релевантного выбора плоскостей сечений. Попытки обеспечить сбалансированность данных структур в случае динамических сцен оказываются безнадежными вследствие того, что незначительное ускорение запросов локализации не компенсируется существенными затратами на обновление структур индексации.

Вернемся к ключевым вопросам реализации октальных деревьев. Одним из интересных аспектов является способ разрешения коллизий, связанных с неоднозначностью приписывания объектов, лежащих на секущих плоскостях, тому или иному дочернему октанту. Существует несколько альтернатив. В случае приписывания объекта нескольким октантам возрастают расходы на хранение избыточных ссылок, а также дополнительные вершины подвергаются анализу в ходе выполнения типовых запросов.

Другой альтернативой является возможность ассоциировать такие объекты непосредственно с разбиваемой вершиной. В этом случае каждый объект оказывается ассоциированным только с одним октантом, что упрощает реализацию операций добавления и удаления объектов. Данный способ применяется, в частности, в структуре со строгой многоуровневой локализацией, известной в литературе как MX-CIF octree [61]. Однако при ее использовании существует вероятность, что большая часть объектов сцены попадет на секущие плоскости верхних октантов и окажется приписанной им. В этом случае дерево вырождается и его дальнейшее использование для разрешения пространственных запросов теряет смысл. Проблема особенно усугубляется для объектов, соразмерных габаритам единичных октантов, но локализующихся на верхних уровнях дерева.

В работах [62], [63], [64] данная проблема решается путем пропорционального увеличения габаритов каждого октанта в $p > 0$ раз вдоль каждой из осей. Параметр p выбирается, исходя из размеров объектов, подлежащих более строгой локализации. Недостатком данного метода является перекрытие

октантов и необходимость анализа большего числа вершин при типовых запросах (этот аспект довольно близок проблеме пространственного перекрытия в методах кластеризации объектов). Для преодоления этого недостатка в работе [64] было предложено смещать положение плоскостей разбиения относительно центра разбиваемого октанта на четверть его размера вдоль каждой из осей. В этом случае исключается пересечение дочерних октантов, однако увеличивается число дочерних ячеек. Можно показать, что при выборе параметра $p = 1/2$ любой объект сцены локализуется в ячейке, габариты которой не превышают размеры объекта более чем в 4 раза. Несмотря на увеличение числа вершин и усложненный пространственный анализ, данный метод хорошо компенсирует недостатки классических октальных деревьев поскольку обеспечивает точную локализацию объектов и позволяет установить однозначное соответствие объектов вершинам октального дерева.

На рисунках 9а-9д приведены псевдокоды алгоритмов для выполнения основных пространственных запросов к рассмотренным в этом разделе MX-CIF октальным деревьям со строгой многоуровневой локализацией объектов, где в качестве критерия разбиения используется ограничение максимально допустимого количества объектов M, хранимого в вершине.

```
procedure FIND_OBJECTS_IN_VOLUME(T,V,R)
pointer node T
pointer volume V
pointer collection R

if(IS_INTERSECTING(T,V) )
        {
        for_each(object o in OBJECTS(T))
                {
                if(IS_INTERSECTING(o,V)
                        ADD(R,o)
                }

        for_each(node child in CHILDREN(T))
                FIND_OBJECTS_IN_VOLUME(child, V, R)
        }
```

Рис. 9а. Псевдокод операции поиска объектов в заданной области.

Рассмотрим псевдокод алгоритм поиска объектов в заданной области сцены, изображенный на рис. 9а. Он незначительно отличается от приведенного для структур, использующих композицию объектов. Отличие связано с тем, что нелистовые вершины так же содержат ссылки на объекты подлежащие анализу. Как и в случае R-дерева, в наихудшем случае анализу подвергнутся

все объекты, входящие в сцену, поэтому вычислительная сложность запроса составляет $O(N)$.

Псевдокод алгоритма поиска ближайших соседних объектов приведен на рис. 9б. Основная процедура FIND_NEAREST_NEIGHBOURS принимает в качестве аргумента объект и возвращает множество его ближайших соседей. Используемая в ней вспомогательная функция NODE по данному объекту находит ассоциированную с ним вершину V октального дерева. Для этого рекурсивный обход октального дерева сверху вниз продолжается до тех пор, пока данный объект целиком содержится в рассматриваемой вершине дерева. Последняя вершина, удовлетворяющая этому условию, является искомой. Вычислительная сложность такой процедуры составляет $O(D)$, где D — глубина октального дерева. Также существенной деталью является возможность задать начальную аппроксимацию расстояния до ближайших объектов. Действительно, если при построении октального дерева вершина была разбита, то в ней должен содержаться, по меньшей мере, $M+1$ объект. Следовательно, родитель вершины V содержит еще, как минимум, один объект, расстояние до которого не превышает удвоенной длины диагонали V. Это значение и выбирается в качестве начальной аппроксимации. В остальном алгоритм поиска ближайших соседей в октальных деревьях концептуально повторяет описанный алгоритм поиска ближайших соседей в R-деревьях. В наихудшем случае трудоемкость выполнения этого запроса составляет $O(N)$.Необходимо отметить схожесть приведенного алгоритма с алгоритмом поиска октантов, смежных данному. В работе [65] было показано, что для поиска соседних вершин заданной, лежащих на том же уровне иерархии, в среднем необходимо 4 раза проследовать по ссылке, связывающей родительскую и дочернюю вершины. Это дает основание полагать, что в среднем трудоемкость поиска ближайших соседних объектов будет значительно ниже.

```
procedure FIND_NEAREST_NEIGHBOURS(O, R)
pointer object O
pointer collection R

pointer node V=NODE(O)
float D = 2*DIAG(V)
FIND_NEAREST_LEAF(O,V,D,R)
FIND_NEAREST_NEIGHBOUR_ANCESTORS(PARENT(V),O,V,D,R)

procedure FIND_NEAREST_LEAF (O,N,D,R)
pointer object O
pointer node N
pointer float D
pointer collection R
```

```
if(DISTANCE(O,N)>D)
        return

for_each(object o != O  in OBJECTS(N))
        {
        float distance = DISTANCE(o, O)
        if(D< distance)
                continue
        else
                {
                if(D > distance)
                        {
                        CLEAR(R)
                        D = distance
                        }
                ADD(R,o)
                }
        }
for_each(node child in CHILDREN(N))
        FIND_NEAREST_LEAF(O,child,D,R)

procedure FIND_NEAREST_NEIGHBOUR_ANCESTORS (N,O,E,D,R)
pointer node N
pointer object O
pointer node E
pointer float D
pointer collection R

for_each(object o in OBJECTS(N))
        {
        float distance = DISTANCE(o, O)
        if(D< distance)
                continue
        else
                {
                if(D > distance)
                        {
                        CLEAR(R)
                        D = distance
                        }
                ADD(R,o)
                }
        }
```

```
for_each(node n != E in CHILDREN(N))
        FIND_NEAREST_LEAF (O,n,D,R)

if(IS_ROOT(N))
        return

return FIND_NEAREST_NEIGHBOUR_ANCESTORS
(PARENT(N),O,N,D,R)
```

Рис. 9б. Псевдокод алгоритма поиска ближайших объектов.

Вставка объекта в октальное дерево осуществляется процедурой INSERT, псевдокод которой приведен на рис. 9в. В качестве параметров она объект и вершину октального дерева, в которую он вставляется. Используемая в методе вспомогательная функция COUNT_OVERLAPPED_CHILDREN возвращает число дочерних октантов заданного, пересекающихся с данным объектом. Если их более одного, то объект вставляется в текущую вершину дерева при помощи процедуры ADD, иначе вставка рекурсивно повторяется для дочерних вершин. Вычислительная трудоемкость процедуры ADD составляет $O(\log N_o)$ где N_o — количество объектов ассоциированных с данной вершиной (мы полагаем, что объекты, ассоциированные с вершинами октального дерева, проиндексированы для быстрого поиска). В наихудшем случае, когда все объекты сцены оказываются ассоциированными с единственной ячейкой, $N_o = N$, где N — число объектов в сцене. Функция SHOULD_SPLIT принимает в качестве параметра вершину октального дерева и возвращает TRUE, если количество объектов, ассоциированных с ней, превышает допустимое, и ее разбиение приведет к тому, что как минимум один из дочерних узлов будет непустым. В противном случае возвращается значение FALSE. Процедура SPLIT выполняет разбиение октанта и переносит часть ассоциированных с ним объектов в дочерние вершины. При условии, что объекты, которые могут быть помещены в дочерние блоки, помечаются заранее, затрачиваемое время составит $O(M)$. Таким образом, трудоемкость вставки объекта в октальное дерево в наихудшем случае составляет $O(\log N + D + M)$.

```
procedure INSERT(T,O)
pointer node T
pointer object O

if(!IS_OVERLAPPING(T, O)
        return

if(!IS_LEAF(T))
```

```
{
if( COUNT_OVERLAPPED_CHILDREN(T,O)>1 )
        ADD(T,O)
else
        for_each(t in CHILDREN(T))
                INSERT(t,O)
return
}

ADD(T,O)
if(SHOULD_SPLIT(T))
        SPLIT(T)
return

procedure SPLIT(T)
pointer node T

collection B = SPLIT_BOUNDS(BOUNDS(T))
for_each(bounds b in B)
        ADD_CHILD(T, NODE(b))

for_each(object o in OBJECTS(T))
        INSERT(T,o)
```

Рис. 9в. Псевдокод операции вставки объекта в октальное дерево.

Ниже на рис. 9д приведен псевдокод процедуры REMOVE, удаляющей объект из октального дерева. В качестве параметра она принимает объект, который при помощи процедуры REMOVE_NODE удаляется из вершины, которая с ним ассоциирована. Для ее нахождения применяется функция NODE, описанная выше. Вычислительная сложность процедуры REMOVE_NODE составляет $O(\log N_o)$. Как уже было сказано ранее, в наихудшем случае, когда все объекты попадают в единственную ячейку пространства $N_o = N$. Функция SHOULD_MERGE возвращает TRUE, если для данной вершины не выполняются условия разбиения. Процедура MERGE принимает в качестве параметра вершину и удаляет все дочерние вершины. При этом все ассоциированные с ними объекты переносятся в родителя. Процедура рекурсивно выполняется до тех пор, пока в рассматриваемой вершине не выполняются условия разбиения. Вычислительная сложность этой процедуры не превышает $O(M + D)$. Таким образом, вычислительные затраты на удаление объекта составляют $O(\log N + D + M)$.

```
procedure REMOVE(O)
```

```
pointer object O

pointer node N = NODE(O)
REMOVE_NODE(N,O)

for_each(node p in PARENTS(N))
        if(SHOULD_MERGE(p))
                MERGE(p)

procedure MERGE(T)
pointer node T

for_each(node child in CHILDREN(T))
        for_each(object o in OBJECTS(child))
                ADD(T, o)
        REMOVE_CHILD(T, child)

if(SHOULD_MERGE(PARENT(T))
        return MERGE(PARENT(T))
```

Рис. 9д. Псевдокод операции удаления объекта из октального дерева.

5. Заключение

Таким образом, проведен сравнительный анализ современных методов поиска и индексации многомерных данных в приложениях моделирования больших динамических сцен. В рамках двух фундаментальных подходов, реализующих принципы объектной кластеризации и пространственной декомпозиции, выделены основные семейства методов и обсуждены их алгоритмические особенности. Анализ методов проведен в контексте комплексных требований, предъявляемых к приложениям обсуждаемого класса, и прежде всего, требований эффективности исполнения типовых пространственных запросов в больших сценах с недетерминированной динамикой и жестким характером пространственно-временной когерентности. Проведенный сравнительный анализ вычислительной сложности позволяет сделать вывод о перспективности методов, основанных на декомпозиции пространства, и, в частности, октальных деревьев со строгой многоуровневой локализацией объектов (MX-CIF) и с нижней границей кардинальности ячеек. Данный вывод основывается на следующих фактах:

— Методы, основанные на пространственной декомпозиции, и методы, использующие объектную кластеризацию, имеют одинаковую асимптотическую сложность операций поиска объектов в заданной области и поиска соседей объектов, составляющую $O(N)$. Вместе с тем,

сложность операций добавления объекта в сцену и его удаления для методов декомпозиции выражается оценкой $O(\log N + D + M)$, что более оптимистично, чем затраты $O(M \log N)$, необходимые для методов кластеризации.

— Методы кластеризации в большей степени ориентированы на особенности страничной работы с внешней памятью, что приводит к требованию высокой наполненности вершин дерева и его сбалансированности. Однако эти требования не являются абсолютно критичными для скорости разрешения пространственных запросов. Более гибкие условия наполненности ячеек в методах декомпозиции могут приводить к некоторой разбалансировке дерева, однако позволяют более точно локализовать объекты уже на верхних уровнях и избежать дополнительные проверки на нижних уровнях.

— Кластеризация объектов приводит к чрезмерно высоким затратам в динамических сценах, связанным с необходимостью перманентной перебалансировки дерева и сложным пространственным анализом. Для обновления структур индексации в методах декомпозиции достаточно идентифицировать ячейку сцены, в которой произошли события, и соответствующим образом модифицировать ее и, возможно, ее локальных соседей.

— Наконец, структуры пространственной декомпозиции инварианты по отношению к порядку событий в сцене и определяются лишь ее текущим представлением. В методах кластеризации необходим дополнительный анализ для обработки потока событий, чтобы обеспечить требуемую инвариантность структур индексации и избежать их деградации при масштабных изменениях в сцене.

Список литературы

[1]. V. Semenov, K. Kazakov, S. Morozov, O. Tarlapan, V. Zolotov and T. Dengenis, "4D modeling of large industrial projects using spatio-temporal decomposition," in *eWork and eBusiness in Architecture, Engineering and Construction*, London, UK, 2010.

[2]. С. Кузнецов и Б. Костенко, «История и актуальные проблемы темпоральных баз данных,» *Труды Института системного программирования,* т. 2, № 13, стр. 77-114, 2007.

[3]. V. Semenov, K. Kazakov, V. Zolotov, H. Jones and S. Jones, "Combined strategy for efficient collision detection in 4D planning applications," in *Computing in Civil and Building Engineering*, Nottingham, UK, 2010.

[4]. R. Bayer and E. M. McCreight, "Organization and maintanence of large ordered indexes," *Acta Informatica,* vol. 3, no. 1, pp. 173-189, 1972.

[5]. D. J. Abel, "A B+-tree structure for large quadtrees," *Computer Vision, Graphics, and Image Processing,* vol. 1, no. 27, pp. 19-31, July 1984.

[6]. [6] D. Comer, "The ubiquitos B-tree," *ACM Computing Surveys,* vol. 2, no. 11, pp. 121-137, June 1979.

[7]. H. Sagan, Space-Filling curves, New York: Springer-Verlag, 1994.
[8]. A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the ACM SIGMOD Conference*, Boston, USA, 1984.
[9]. T. Sellis, N. Roussopoulos and C. Faloutsos, "The R+-tree: a dynamic index for multi-dimentional objects," in *Proceedings of the 13th International Conference on Very Large Databases (VLDB)*, Brighton, UK, 1987.
[10]. N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *Proceedings of the ACM SIGMOD Conference*, Atlantic City, USA, 1990.
[11]. C. Yu, B. C. Ooi, K.-L. Tan and H. V. Jagadish, "Indexing the distance: an efficient method to KNN processing," in *Proceedings of the 27st international Conference on Very Large Databases (VLDB)*, Roma, Italy, 2001.
[12]. A. Beygelzimer, S. Kakade and J. Langford, "Cover Trees for Nearest Neighbor," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2006.
[13]. W. A. Burkhard and R. Keller, "Some approaches to best-match file searching," *Communications of the ACM,* vol. 4, no. 16, pp. 230-236, April 1973.
[14]. J. L. Bentley, "Decomposable searching problems," *Information Processing Letters,* vol. 5, no. 8, pp. 244-251, June 1979.
[15]. J. L. Bentley and D. Wood, "An optimal worst-case algorithm for reporting intersections of rectangles," *IEEE Transactions on Computers,* vol. 7, no. 29, pp. 571-577, July 1980.
[16]. H. Edelsbrunner, "A new approach to rectangle intersections: part II," *International Journal of Computer Mathematics,* Vols. 3-4, no. 13, pp. 221-229, 1983.
[17]. E. M. McCreight, "Priority search trees," *SIAM Journal on Computing,* vol. 2, no. 14, pp. 257-276, May 1985.
[18]. H. Fuchs, G. Abram and E. Grant, "Near real-time shaded display of rigid objects," *Computer Graphics,* vol. 3, no. 17, pp. 65-72, 1983.
[19]. H. Fuchs, Z. Kedem and B. Naylor, "On visible surface generation by a priori tree structures," *Computer Graphics,* vol. 3, no. 14, pp. 124-133, 1980.
[20]. A. Henrich, H. W. Six and P. Widmayer, "The LSD-tree: spatial access to multidimentional point an non-point data," in *Proceedings of the 15th International Conference on Very Large Databases (VLDB)*, Amsterdam, Netherlands, 1989.
[21]. D. Lomet and B. Salzberg, "The hB-tree: a multi-attribute indexing method with good guaranteed performance," *ACM Transactions on Database Systems,* vol. 4, no. 15, pp. 625-658, December 1990.
[22]. J. Xu, B. Zheng, W. C. Lee and D. L. Lee, "The D-tree: an index structure for planar point queries in location-based wireless services," *IEEE Transactions on Knowledge and Data Engineering,* vol. 12, no. 16, pp. 1526-1542, December 2004.
[23]. J. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM,* vol. 9, no. 18, pp. 509-517, September 1975.
[24]. J. H. Friedman, J. L. Bentley and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software,* vol. 3, no. 3, pp. 209-226, September 1977.
[25]. K. Chakrabarti and S. Mehrotra, "The hybrid tree: an index structure for high dimentional feature spaces," in *Proceedings of the 15th IEEE International Conference on Data Engineering*, Sydney, Australia, 1999.
[26]. D. M. Mount and S. Arya, "ANN: a library for aproximate nearest neighbour searching," in *Proceedings of the 2nd Annual Center for Geometric Computing Workshop on Computational Geometry*, Durham, 1997.

[27]. D. A. White and R. Jain, "Similarity indexing with the SS-tree," in *Proceedings of the 12th IEEE International Conference on Data Engineering*, New Orleans, USA, 1996.

[28]. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman and A. Y. Wu, "An optimal algorithm for approximate nearest neighbour searching in fixed dimentions," *Journal of the ACM,* vol. 6, no. 45, pp. 891-923, November 1998.

[29]. J. O'Rourke, "Dynamically quantized spaces for focusing Hough transform," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, Vancouver, Canada, 1981.

[30]. K. R. S. Jr., "Dynamicaly quantized pyramids," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, Vancouver, Canada, 1981.

[31]. L. Becker, K. Hinrichs and U. Finke, "A new algorithm for computing joins with grid files," in *Proceedings of the 9th IEEE Conference on Data Engineering*, Vienna, Austria, 1993.

[32]. H. Samet, Foundations of Multidimentional and Metric Data Structures, San Francisco: Morgan Kaufmann, 2006.

[33]. P. Bogdanovich and H. Samet, "The ATree: a data structure to support a very large scientific databases," in *Springer-Verlag Lecture Notes in Computer Science*, vol. 1737, P. Agouris and A. Stefanidis, Eds., Springer-Verlag, 1990, pp. 235-248.

[34]. K. Knowlton, "Progressive transmission of gray-scale and binary pictures by simple efficient and lossless encoding schemes," *Proceedings of IEEE,* vol. 7, no. 68, pp. 885-896, 1980.

[35]. Y. Cohen, M. Landy and M. Pavel, "Hierarchical coding of binary images," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 3, no. 7, pp. 284-298, 1985.

[36]. G. Nagy and S. Wagle, "Hierarchical representation of optically scanned documents," in *Proceedings of 7th International Conference on Pattern Recognition*, 1984.

[37]. A. Dengel, "Object-oriented representation of image space by puzzle-trees," *SPIE Visual Communications and Image Processing,* pp. 20-30, 1991.

[38]. B. Shneiderman, "Tree visualization with tree maps: 2-d space-filling approach," *ACM Transactions on Graphics,* vol. 1, no. 11, pp. 92-99, 1992.

[39]. I. Kamel and C. Faloutsos, "Hilbert R-tree: an improved R-tree using fractals," *Proceedings of 20th International Conference on Very Large Data Bases,* vol. 3, no. 3, 1994.

[40]. I. Kamel and C. Faloutsos, "On packing R-trees," in *Proceedings of 2nd International Conference on Information and Knowledge Management*, 1993.

[41]. S. T. Leutenegger, M. A. Lopez and J. Edgington, "STR: a simple and efficient algorithm for R-tree packing," in *Proceedings of the 13th IEEE International conference on Data Engineering*, 1997.

[42]. J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-DOPs," *IEEE Transactions on Visualization and Computer Graphics,* vol. 1, no. 4, pp. 21-36, Januray 1998.

[43]. O. Gunther and J. Bilmes, "Tree-based access methods for spatial databases: implementation and performance evaluation," *IEEE Transactions on Knowledge and Data Engineering,* vol. 3, no. 3, pp. 342-356, 1991.

[44]. S. Gottschalk, M. C. Lin and D. Manocha, "OBB Tree: a hierarchical structure for rapid interference detection," in *Proceedings of the SIGGRAPH'96 Conference*, New Orleans, USA, 1996.

[45]. S. Gottschalk, Collision queries using oriented bounding boxes, Chapel Hill: The University of North Carolina, 2000.

[46]. Y. Theodoris and T. Sellis, "Optimization issues in R-tree construction," in *IGIS'94: Geographic Information Systems, International Workshop on Advanced Research in Geographic Information Systems*, 1994.
[47]. Y. Garcia, M. Lopez and S. Leutenneger, "A greedy algorithm for bulk loading R-trees," in *Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems*, 1997.
[48]. B. Becker, P. Franciosa, S. Gschwind, T. Ohler, G. Thiemt and P. Widmayer, "Enclosing many boxes by an optimal pair of boxes," in *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, 1992.
[49]. Y. Garcia, M. Lopez and S. Leutenneger, "An optimal node splitting for R-trees," in *Proceedings of the 24th International Conference on Very Large Data Bases*, 1998.
[50]. C. Ang and T. Tan, "New linear node splitting algorithm for R-trees," in *Advances in Spatial Databases – 5th International Symposium, SSD'97*, 1997.
[51]. E. G. Hoel and H. Samet, "Benchmarking spatial join operations with spatial output," in *Proceedings of the 21th International Conference on Very Large Databases (VLDB)*, Zurich, Switzerland, 1995.
[52]. M. Cai and P. Revesz, "Parametric rectangles: an index structure for moving objects," in *Proceedings of the 10th COMAD International conference on management of data*, 2000.
[53]. S. Saltenis, C. S. Jensen, S. T. Leuteenegger and M. A. Lopez, "Indexing the positions of continuously moving objects," in *Proceedings of the ACM SIGMOD Conference*, 2000.
[54]. Y. Tao, D. Papadias and J. Sun, "The TRP*-tree: an optimized spatio-temporal access for predictive queries," in *Proceedings of the 29th international conference on Very Large Data Bases (VLDB)*, 2003.
[55]. S. B. M. Bell, B. M. Diaz, F. Holroyd and M. J. Jackson, "Spatially referenced method of processing raster and vector data," *Image and Vision Computing,* vol. 4, no. 1, pp. 211-220, 1983.
[56]. L. Gibson and D. Lucas, "Vectorization of raster images using hierarchical methods," *Computer Graphics and Image Processing,* vol. 1, no. 20, pp. 82-29, 1982.
[57]. Г. М. Адельсон-Вельский и Е. М. Ландис, «Один алгоритм организации информации,» *Доклады Академии Наук СССР,* № 146, стр. 263-266, 1962.
[58]. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Red–Black Trees," in *Introduction to Algorithms*, MIT Press and McGraw-Hill, 2001, p. 273–301.
[59]. A. Amir, A. Efrat, P. Indyk and H. Samet, "Efficient algorithms and regular data structures for dilation, location and proximity problems," *Algorithmica,* vol. 2, no. 30, pp. 164-187, 2001.
[60]. G. Schrack, "Finding neighbors of equal size in linear quadtrees and octrees in constant time," *CVGIP: Image Understanding,* vol. 3, no. 55, pp. 221-230, 1992.
[61]. G. Kedem, "The quad-CIF tree: a data structure for hierarchical on-line algorithms," in *Proceedings of the 19th Design Automation Conference*, 1992.
[62]. A. U. Frank, "Problems of realizing LIS: storage methods for space related data: the fieldtree," ETH, 1983.
[63]. A. U. Frank and R.Barrera, "The Fieldtree: a data structure for geographic information systems," in *Springer-Verlag Lecture Notes in Computer Science*, vol. 409, A. Buchmann, O. Gunter, T. Smith and Y. Wang, Eds., Springer-Verlag, 1989, pp. 29-44.
[64]. T. Ulrich, "Loose octrees," in *Game programming gems*, Rockland, Charles river media, 2000, pp. 444-453.

[65]. H. Samet, "Neighbor finding techniques for images represented by quadtrees," *Computer Graphics and Image processing,* vol. 1, no. 18, pp. 37-57, 1982.

Advanced indexing methods for large spatial data in complex dynamic scenes

V.A. Zolotov, V.A. Semenov(ISP RAS, Moscow, Russia)

Annotation. This paper is dedicated to review of recent methods for indexing of multidimensional data and their use for modeling of large-scale dynamic scenes. The problem arises in many application domains such as computer graphics systems, virtual and augmented reality systems, CAD systems, robotics, geographical information systems, project management systems, etc.

Two fundamental approaches to the indexing of multidimensional data, namely object aggregation and spatial decomposition, have been outlined. In the context of the former approach balanced search trees, also referred as object pyramids, have been discussed. In particular, generalizations of B-trees such as R-trees, R*-trees, R+-trees have been discussed in conformity to indexing of large data located in external memory and accessible by pages. The conducted analysis shows that object aggregation methods are well suited for static scenes but very limited for dynamic environments.

The latter approach assumes the recursive decomposition of scene volume in accordance with the cutting rules. Space decomposition methods are octrees, A-trees, bintrees, K-D-trees, X-Y-trees, treemaps and puzzle-trees. They support more reasonable compromise between performance of spatial queries and expenses required to update indexing structures and to keep their in concordant state under permanent changes in the scenes. Compared to object pyramids, these methods look more promising for dramatically changed environments. It is concluded that regular dynamic octrees are most effective for the considered applications of modeling of large-scale dynamic scenes.

Keywords: spatial indexing, multidimensional data, dynamic scene modeling

References

[1]. Semenov V.A., Kazakov K.A., Morozov S.V., Tarlapan O.A., Zolotov V.A., Dengenis T., "4D modeling of large industrial projects using spatio-temporal decomposition" in eWork and eBusiness in Architecture, Engineering and Construction, London, UK, pp. 89-95, 2010.

[2]. Kuznecov S.D., Kostenko B.B. "Istoriya i aktual'nye problemy temporal'nykh baz dannykh" [History and actual state of temporal databases]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 2, № 13, pp. 77-114, 2007. (in Russian)

[3]. Semenov V.A., Kazakov K.A., Zolotov V.A., Jones H., Jones S. "Combined strategy for efficient collision detection in 4D planning applications" in Computing in Civil and Building Engineering, Nottingham, UK, pp. 31-39, 2010.

[4]. Bayer R., McCreight E. M., "Organization and maintanence of large ordered indexes" Acta Informatica, vol. 3, no. 1, pp. 173-189, 1972.

[5]. Abel D. J. "A B+-tree structure for large quadtrees" Computer Vision, Graphics, and Image Processing, vol. 1, no. 27, pp. 19-31, July 1984.

[6]. Comer D. "The ubiquitos B-tree" ACM Computing Surveys, vol. 2, no. 11, pp. 121-137, June 1979.

[7]. Sagan H. "Space-Filling curves", New York: Springer-Verlag, 1994.

[8]. Guttman A. "R-trees: a dynamic index structure for spatial searching" in Proceedings of the ACM SIGMOD Conference, Boston, USA, 1984.
[9]. Sellis T., Roussopoulos N., Faloutsos C. "The R+-tree: a dynamic index for multi-dimentional objects" in Proceedings of the 13th International Conference on Very Large Databases (VLDB), Brighton, UK, 1987.
[10]. Beckmann N., Kriegel H. P., Schneider R., Seeger B. "The R*-tree: an efficient and robust access method for points and rectangles" in Proceedings of the ACM SIGMOD Conference, Atlantic City, USA, 1990.
[11]. Yu C., Ooi B. C., Tan K.-L., Jagadish H. V. "Indexing the distance: an efficient method to KNN processing" in Proceedings of the 27st international Conference on Very Large Databases (VLDB), Roma, Italy, 2001.
[12]. Beygelzimer A., Kakade S., Langford J., "Cover Trees for Nearest Neighbor" in Proceedings of the International Conference on Machine Learning (ICML), 2006.
[13]. Burkhard W. A., Keller R. "Some approaches to best-match file searching" Communications of the ACM, vol. 4, no. 16, pp. 230-236, April 1973.
[14]. Bentley J. L. "Decomposable searching problems" Information Processing Letters, vol. 5, no. 8, pp. 244-251, June 1979.
[15]. Bentley J. L., Wood D. "An optimal worst-case algorithm for reporting intersections of rectangles" IEEE Transactions on Computers, vol. 7, no. 29, pp. 571-577, July 1980.
[16]. Edelsbrunner H. "A new approach to rectangle intersections: part II" International Journal of Computer Mathematics, Vols. 3-4, no. 13, pp. 221-229, 1983.
[17]. McCreight E. M. "Priority search trees" SIAM Journal on Computing, vol. 2, no. 14, pp. 257-276, May 1985.
[18]. Fuchs H., Abram G., Grant E. "Near real-time shaded display of rigid objects" Computer Graphics, vol. 3, no. 17, pp. 65-72, 1983.
[19]. Fuchs H., Kedem Z., Naylor B. "On visible surface generation by a priori tree structures" Computer Graphics, vol. 3, no. 14, pp. 124-133, 1980.
[20]. Henrich A., Six H. W., Widmayer P. "The LSD-tree: spatial access to multidimentional point an non-point data" in Proceedings of the 15th International Conference on Very Large Databases (VLDB), Amsterdam, Netherlands, 1989.
[21]. Lomet D., Salzberg B. "The hB-tree: a multi-attribute indexing method with good guaranteed performance" ACM Transactions on Database Systems, vol. 4, no. 15, pp. 625-658, December 1990.
[22]. Xu J., Zheng B., Lee W. C., Lee D. L. "The D-tree: an index structure for planar point queries in location-based wireless services" IEEE Transactions on Knowledge and Data Engineering, vol. 12, no. 16, pp. 1526-1542, December 2004.
[23]. Bentley J.L. "Multidimensional binary search trees used for associative searching" Communications of the ACM, vol. 9, no. 18, pp. 509-517, September 1975.
[24]. Friedman J. H., Bentley J. L., Finkel R. A. "An algorithm for finding best matches in logarithmic expected time" ACM Transactions on Mathematical Software, vol. 3, no. 3, pp. 209-226, September 1977.
[25]. Chakrabarti K., Mehrotra S. "The hybrid tree: an index structure for high dimentional feature spaces" in Proceedings of the 15th IEEE International Conference on Data Engineering, Sydney, Australia, 1999.
[26]. Mount D. M., Arya S. "ANN: a library for aproximate nearest neighbour searching" in Proceedings of the 2nd Annual Center for Geometric Computing Workshop on Computational Geometry, Durham, 1997.
[27]. White D. A., Jain R. "Similarity indexing with the SS-tree" in Proceedings of the 12th IEEE International Conference on Data Engineering, New Orleans, USA, 1996.

[28]. Arya S., Mount D. M., Netanyahu N. S., Silverman R., Wu A. Y. "An optimal algorithm for approximate nearest neighbour searching in fixed dimentions" Journal of the ACM, vol. 6, no. 45, pp. 891-923, November 1998.
[29]. O'Rourke J. "Dynamically quantized spaces for focusing Hough transform" in Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver, Canada, 1981.
[30]. Kenneth R. Sloan Jr., "Dynamicaly quantized pyramids" in Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver, Canada, 1981.
[31]. Becker L., Hinrichs K., Finke U. "A new algorithm for computing joins with grid files" in Proceedings of the 9th IEEE Conference on Data Engineering, Vienna, Austria, 1993.
[32]. Samet H. "Foundations of Multidimentional and Metric Data Structures", San Francisco: Morgan Kaufmann, 2006.
[33]. Bogdanovich P., Samet H. "The ATree: a data structure to support a very large scientific databases" in Springer-Verlag Lecture Notes in Computer Science, vol. 1737, P. Agouris and A. Stefanidis, Eds., Springer-Verlag, 1990, pp. 235-248.
[34]. Knowlton K. "Progressive transmission of gray-scale and binary pictures by simple efficient and lossless encoding schemes" Proceedings of IEEE, vol. 7, no. 68, pp. 885-896, 1980.
[35]. Cohen Y., Landy M., Pavel M. "Hierarchical coding of binary images" IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 3, no. 7, pp. 284-298, 1985.
[36]. Nagy G., Wagle S. "Hierarchical representation of optically scanned documents" in Proceedings of 7th International Conference on Pattern Recognition, 1984.
[37]. Dengel A. "Object-oriented representation of image space by puzzle-trees" SPIE Visual Communications and Image Processing, pp. 20-30, 1991.
[38]. Shneiderman B. "Tree visualization with tree maps: 2-d space-filling approach" ACM Transactions on Graphics, vol. 1, no. 11, pp. 92-99, 1992.
[39]. Kamel I., Faloutsos C. "Hilbert R-tree: an improved R-tree using fractals" Proceedings of 20th International Conference on Very Large Data Bases, vol. 3, no. 3, 1994.
[40]. Kamel I., Faloutsos C. "On packing R-trees" in Proceedings of 2nd International Conference on Information and Knowledge Management, 1993.
[41]. Leutenegger S. T., Lopez M. A., Edgington J. "STR: a simple and efficient algorithm for R-tree packing" in Proceedings of the 13th IEEE International conference on Data Engineering, 1997.
[42]. Klosowski J. T., Held M., Mitchell J. S. B., Sowizral H., Zikan K. "Efficient collision detection using bounding volume hierarchies of k-DOPs" IEEE Transactions on Visualization and Computer Graphics, vol. 1, no. 4, pp. 21-36, Januray 1998.
[43]. Gunther O., Bilmes J. "Tree-based access methods for spatial databases: implementation and performance evaluation" IEEE Transactions on Knowledge and Data Engineering, vol. 3, no. 3, pp. 342-356, 1991.
[44]. Gottschalk S., Lin M. C., Manocha D. "OBB Tree: a hierarchical structure for rapid interference detection" in Proceedings of the SIGGRAPH'96 Conference, New Orleans, USA, 1996.
[45]. Gottschalk S. "Collision queries using oriented bounding boxes", Chapel Hill: The University of North Carolina, 2000.
[46]. Theodoris Y., Sellis T. "Optimization issues in R-tree construction" in IGIS'94: Geographic Information Systems, International Workshop on Advanced Research in Geographic Information Systems, 1994.

[47]. Garcia Y., Lopez M., Leutenneger S. "A greedy algorithm for bulk loading R-trees" in Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems, 1997.

[48]. Becker B., Franciosa P., Gschwind S., Ohler T., Thiemt G., Widmayer P., "Enclosing many boxes by an optimal pair of boxes" in Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science, 1992.

[49]. Garcia Y., Lopez M., Leutenneger S. "An optimal node splitting for R-trees" in Proceedings of the 24th International Conference on Very Large Data Bases, 1998.

[50]. Ang C., Tan T. "New linear node splitting algorithm for R-trees" in Advances in Spatial Databases – 5th International Symposium, SSD'97, 1997.

[51]. Hoel E. G., Samet H. "Benchmarking spatial join operations with spatial output" in Proceedings of the 21th International Conference on Very Large Databases (VLDB), Zurich, Switzerland, 1995.

[52]. Cai M., Revesz P. "Parametric rectangles: an index structure for moving objects" in Proceedings of the 10th COMAD International conference on management of data, 2000.

[53]. Saltenis S., Jensen C. S., Leuteenegger S. T., Lopez M. A., "Indexing the positions of continuously moving objects" in Proceedings of the ACM SIGMOD Conference, 2000.

[54]. Tao Y., Papadias D., Sun J. "The TRP*-tree: an optimized spatio-temporal access for predictive queries" in Proceedings of the 29th international conference on Very Large Data Bases (VLDB), 2003.

[55]. Bell S. B. M., Diaz B. M., Holroyd F., Jackson M. J. "Spatially referenced method of processing raster and vector data" Image and Vision Computing, vol. 4, no. 1, pp. 211-220, 1983.

[56]. Gibson L., Lucas D. "Vectorization of raster images using hierarchical methods" Computer Graphics and Image Processing, vol. 1, no. 20, pp. 82-29, 1982.

[57]. Adel'son-Vel'sky G.M, Landis E.M. "Odin algoritm organizatsii informatsii"[An algorithm for information organization] Proceedings of RAS USSR, № 146, pp. 263-266, 1962.

[58]. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. "Red–Black Trees" in Introduction to Algorithms, MIT Press and McGraw-Hill, 2001, p. 273–301.

[59]. Amir A., Efrat A., Indyk P., Samet H. "Efficient algorithms and regular data structures for dilation, location and proximity problems" Algorithmica, vol. 2, no. 30, pp. 164-187, 2001.

[60]. Schrack G. "Finding neighbors of equal size in linear quadtrees and octrees in constant time" CVGIP: Image Understanding, vol. 3, no. 55, pp. 221-230, 1992.

[61]. Kedem G. "The quad-CIF tree: a data structure for hierarchical on-line algorithms" in Proceedings of the 19th Design Automation Conference, 1992.

[62]. Frank A. U., "Problems of realizing LIS: storage methods for space related data: the fieldtree" ETH, 1983.

[63]. Frank A. U., Barrera R. "The Fieldtree: a data structure for geographic information systems" in Springer-Verlag Lecture Notes in Computer Science, vol. 409, [ed] Buchmann A., Gunter O., Smith T., Wang Y., Springer-Verlag, 1989, pp. 29-44.

[64]. Ulrich T., "Loose octrees" in Game programming gems, Rockland, Charles river media, 2000, pp. 444-453.

[65]. Samet H. "Neighbor finding techniques for images represented by quadtrees" Computer Graphics and Image processing, vol. 1, no. 18, pp. 37-57, 1982.

Особенности табличных выражений SQL и их соответствие с концепциями реляционной модели данных

И.В. Блудов. ivan.bludov@gmail.com

Аннотация. В данной статье изложены материалы сравнения и критики SQL по поводу соответствия операторов SQL реляционной модели данных. Рассмотрены особенности табличных выражений в стандарте SQL и показаны случаи, в которых эти выражения противоречат реляционной модели. Приведены варианты использования таких табличных выражений в SQL, которые наиболее точно отражают концепции реляционной модели.

Ключевые слова: SQL, реляционная модель данных, SQL запросы, табличные выражения

1. Введение

Общеизвестно, что SQL является стандартным языком реляционных баз данных, но этот факт не делает его реляционным. В действительности, он отдалился от реляционной модели достаточно давно. Следует понимать различия SQL и реляционной теории, чтобы избежать возможных проблем, связанных с этими различиями.

Данная статья посвящена исследованию различий SQL и реляционной модели данных. Более конкретно, в ней рассматривается один из важных разделов SQL – табличные выражения. В статье проводится анализ табличных выражений стандарта SQL, их особенностей и сравнение с операторами реляционной модели данных. На основе этого анализа выясняется, как такие выражения следует использовать, чтобы они соответствовали реляционной модели. С целью написания качественного SQL следует применять соответствующие практики, что позволит ощутить преимущества реляционной модели.

2. Определения кортежа и отношения

Пусть *T1,T2,..,Tn* (n≥0) - определённые типы, не обязательно различные. Пусть *Ai* - отличные имена атрибутов, ассоциированные с каждым *Ti*. Каждая из *n* комбинаций «имя атрибута/тип» называется *атрибутом*. С каждым атрибутом ассоциировано *значение атрибута vi* типа *Ti*; каждая из *n* пар атрибут/значение называется компонентом. Множество из всех *n* таких компонент определяют *значение кортежа* (*кортеж*, для краткости) над атрибутами *A1,A2,..,An*. Множество из всех *n* атрибутов называется заголовком кортежа.

Пусть {*H*} будет заголовком кортежей, и пусть *t1, t2, ..., tm* будут различные кортежи заголовка {*H*}. Сочетание {*H*} и множества кортежей {*t1*, *t2*, ..., *tm*} называется *значением отношения* (*отношением*, для краткости), над атрибутами *A1*, *A2*, ..., *An*, где *A1*, *A2*, ..., *An* все атрибуты {*H*}. *Заголовком отношения* будет {*H*}, *телом отношения* будет множество кортежей {*t1*, *t2*, ..., *tm*}.

Выводы из определения:

- В реляционной модели каждый атрибут каждого отношения должен иметь имя (то есть *анонимные атрибуты запрещены*), и такие имена должны быть уникальны в данном отношении (то есть *запрещены дублирующиеся имена атрибутов*). В SQL данные правила уместны не везде: они верны для значений переменных таблиц, определённых с помощью CREATE TABLE и CREATE VIEW, но не для результатов запросов над такими таблицами.
- Отношения *никогда не содержат кортежей-дубликатов*. Это свойство следует из того, что тело отношения определено как множество кортежей, а множество в математике не содержит дублирующихся элементов. Поведение SQL в данном смысле не верно: поскольку таблицы в SQL могут содержать дублирующиеся строки, отсюда они не являются отношениями в общем смысле.
- *Кортежи в отношении не упорядочены сверху вниз.* Это свойство следует опять из того, что тело отношения определено как множество, а в математическом множестве элементы не упорядочены. Конечно, из того факта, что кортежи в отношениях не упорядочены, не следует то, что запросы не могут включать спецификации ORDER BY. Однако это означает, что результат такого запроса не будет отношением. Отсортированные результаты удобны для отображения, однако, такой запрос не является реляционный оператором.
- *Атрибуты в отношении также не упорядочены слева направо*, потому что заголовок отношения - также математическое множество. Поведение SQL в данном контексте неверно: в SQL таблицах колонки упорядочены слева направо.

- *Кортежи никогда не содержат NULL* - поскольку, по определению, для каждого атрибута кортежи содержат значения соответствующего типа.
- *Отношение не содержат NULL* – поскольку тело отношения является множеством кортежей, а кортежи не содержат NULL.

3. Замкнутость реляционной модели. Правила выведения типов отношений и таблиц.

По определению результатом каждого реляционного оператора должно быть отношение. И напротив, любой оператор, результатом которого не является отношение по определению не может быть реляционным оператором. В случае SQL реляционные операторы не могут возвращать результаты, содержащие строки дубликаты, упорядоченные столбцы, NULL, неименованные столбцы, дублирующиеся имена столбцов. Замкнутость реляционной модели крайне важна, оно позволяет нам писать в реляционной модели вложенные выражения. Не используйте любые операторы, нарушающие замыкания, если Вы хотите использовать результат в дальнейших реляционных выражениях.

Реляционная модель использует специальные правила выведения типов отношений, которые предполагают, что, если нам известен заголовок входных отношений в операции, мы можем вывести заголовок выходного отношения этой операции. Преимущество такого метода в избегании сложностей, зависящих от порядкового номера атрибутов.

SQL же поддерживает выведение типов несколько не корректно. Во-первых, в SQL вообще нет понятия реляционных типов, взамен он оперирует типами строк. Во-вторых, он допускает в результирующих таблицах неименованные столбцы. В-третьих, допускает в результирующих таблицах дублирование имен столбцов. Для использования SQL реляционным образом Вам следует применять некоторые правила, как это делают реляционные операторы. *Это предполагает надлежащее именование атрибутов отношений:*

- Для всех атрибутов, представляющих «одну и ту же информацию», по возможности назначать одинаковые имена. И напротив, если два атрибута представляют разную информацию, правильно будет определить для них различные имена. Единственный случай, когда следовать данным рекомендациям невозможно, - если два атрибута одной таблицы предоставляют одинаковую информацию.
- Для каждой базовой таблицы определите представления во избежание возможного переименования атрибутов в будущем.
- Используйте такие представления вместо лежащих в их основе базовых таблиц.

- Используйте спецификацию AS для определения подходящих имен для любых столбцов, которые не имеют имени, или имя их не уникально.

Мы не можем игнорировать факт, что столбцы в SQL остаются упорядоченными даже если нам этого не нужно. Следующий код предполагает упорядоченность атрибутов:

- SELECT *
- JOIN, UNION, INTERSECT и EXCEPT—в особенности если не определена спецификация CORRESPONDING в последних трех случаях.
- В перечислении имен атрибутов, следующем за определением переменной области значений
- В перечислении имен атрибутов, определённых в CREATE VIEW
- INSERT, если не определён список имен атрибутов.
- ALL и ANY сравнения, если размерность операндов больше одного
- В выражениях VALUES

4. Сравнение типов в SQL

Рассмотрим общеизвестный факт, что в реляционной модели два значения могут быть сравнены, только если они одного и того же типа. Основная идея здесь в том, что СУБД должна запрещать выполнять любые реляционные операторы (join, union, что угодно), которые подразумевают явное/неявное сравнение значений различных типов. Было бы полезно, если при попытке пользователя сравнить значения различных типов, СУБД запретит данное сравнение, сообщив об ошибке, и попросит пользователя исправить ошибку перед выполнением. Ни один из известных сегодня SQL продуктов не ведет себя подобным образом. В сегодняшних продуктах в зависимости от того, как Вы настроите базу данных, выполнение запроса прервется или же вернет не верный ответ (точнее не совсем не верный ответ, а правильный ответ на неправильный вопрос).

В SQL же реализована строгая типизация, но в несколько слабой форме. Так сравнение строк и чисел не допустимо, однако сравнение целых и действительных чисел допустимо, поскольку перед выполнением сравнения целое число будет приведено к вещественному типу. Такое приведение типов широко распространено, однако, в контексте SQL его применение к реляционным операциям, таким, как объединение, пересечение и разность, приводит к весьма странным результатам, которые впоследствии содержат строки, не принадлежащие ни одному из операндов. Рассмотрим объединение таблиц T1 {X INTEGER, Y NUMERIC(5,1)} и T2 {X NUMERIC(5,1), Y INTEGER}.

SELECT X , Y FROM T1

UNION

SELECT X , Y FROM T2

T1

X	Y
0	1.0
0	2.0

T2

X	Y
0.0	0
0.0	1
1.0	2

X	Y
0.0	1.0
0.0	2.0
0.0	0.0
1.0	2.0

Таким образом, результат содержит строки не принадлежащие ни таблице T1, ни таблице T2 – весьма странное объединение.

Рекомендация: избегайте приведения типов, где это возможно. Так, в контексте SQL, следует быть уверенным, что колонки с одинаковым именем имеют одинаковый тип. Если избежать этого не удается - используйте явное приведение типов, используя CAST или его эквиваленты.

5. Табличные выражения. Особенности стандарта SQL в сравнении с реляционной моделью

В данной главе мы рассмотрим крайне важное *табличное выражение*, которое возникает в многочисленных контекстах языка SQL. В большинстве случаях, в действительности, его можно рассматривать как вершину синтаксического дерева. Следует отметить, что стандарт ссылается на эту конструкцию как на *выражения запроса* (*query expression*) и приписывает другое (более ограниченное) понятие для табличного выражения (“table expression”). Мы предпочитаем термин *табличное выражение*, поскольку он более точно отражает, что выражение возвращает таблицу (точнее *неименованную* таблицу). Напротив, стандарт предполагает “table expression” просто как частный случай, что мы называем select выражением без SELECT раздела как такового.

5.1 Соединения / JOIN EXPRESSIONS

В общем случае табличные выражение разделены на join и non-join табличные выражения

query-expression-body

::= joined-table | nonjoin-query-expression

Рассмотрим только первый из двух случаев. В общем случае *Join выражение* представляет явное соединение двух таблиц, представленных как *ссылка на*

таблицу (*table reference)*. Более точно, Join выражение это либо *cross join*, либо явно/неявно подразумевает наличие типа соединения (join type).

joined-table::=

table-reference CROSS JOIN table-primary

| table-reference [join-type] JOIN table-primary

[ON search-condition | USING (column-name-list)]

| table-reference NATURAL [join-type] JOIN table-primary

| table-reference UNION JOIN table-primary

Cross Join

"Cross join" - это другое название того, что более точно называть расширенное декартовое произведение (сокращенно просто декартово произведение в контексте SQL). Пусть A и B будут таблицы, представленные как результат вычислений table reference. Тогда результатом выражение "A CROSS JOIN B" будет таблица, содержащая все возможные строки R, которые являются конкатенацией строк из A и B. Из этого следует, что соединение A CROSS JOIN B семантически эквивалентно следующему select-выражению:

(SELECT A.*, B.*

FROM A, B)

Из определения видно, что общие атрибуты таблиц A и B встречаются в результате дважды.

Другие Join выражения

Синтаксис для других join выражений следующий:

table-reference [join-type] JOIN table-primary

[ON search-condition | USING (column-name-list)]

| table-reference NATURAL [join-type] JOIN table-primary

| table-reference UNION JOIN table-primary

Тип соединения (Join type) в свою очередь может быть одним из следующих:

join-type ::=

INNER | { LEFT | RIGHT | FULL } [OUTER]

Следует отметить что:

1. NATURAL и UNION не могут быть одновременно указаны.

2. Если указано NATURAL или UNION – тогда ни ON условие, ни USING условие не могут быть указаны.
3. Если не указано ни NATURAL, ни UNION – тогда либо ON условие, либо USING условие должны быть указаны.
4. Если тип соединение опущен – тогда INNER тип предполагается по умолчанию
5. Слово OUTER в случаях LEFT, RIGHT, и FULL является необязательным и не оказывает никакого эффекта на общий смысл выражений.

Рассмотрим следующие варианты соединения:

1. table-reference JOIN table-primary ON search-condition
2. table-reference JOIN table-primary USING (column-name-list)
3. table-reference NATURAL JOIN table-primary

Пусть A и B будут таблицы, представленные как результат вычисления table reference.

1. “A JOIN B ON cond," (где cond – условное выражение) по определению семантически эквивалентно следующему select выражению:

   (SELECT A .*, B.*

   FROM A, B

   WHERE cond)

   Из определения видно, что общие атрибуты таблиц A и B встречаются в результате дважды.
2. Пусть список атрибутов в условии USING будет определён как *C1, C2 ,..., Cn*, и каждый атрибут C l, C2, ..., Cn должен одновременно быть атрибутом A и B. Тогда соединение семантически эквивалентно *Случаю 1*, в котором ON условие представлено как - ON A.Cl = B.Cl AND A.C2 = B.C2 AND ... AND A.Cn = B.Cn. За исключением того, что общие атрибуты *C1, C2 ,..., Cn* встречаются в результате только один раз, а не дважды. А атрибуты в результирующей таблицы упорядочены следующим образом: сперва идут общие атрибуты (упорядоченные слева на право, как это указано в условии USING); затем остальные атрибуты A

(в порядке каком они указаны в A); затем остальные атрибуты B(в порядке каком они указаны в B).

3. Выражение семантически эквивалентно *Случаю 2,* в котором список атрибутов включает *все* общие атрибуты A и B. Отметим, что если таких общих атрибутов нет, в таком случае выражение A NATURAL JOIN B сводиться к A CROSS JOIN B.

Рассмотрим следующие примеры:

S JOIN SP ON S.SNO = SP.SNO

S JOIN SP USING (SNO)

S NATURAL JOIN SP

Эти три выражения эквивалентны: хотя первое возвращает таблицу, содержащую два одинаковых столбца SNO, второе и третье содержит только один такой столбец.

Отметим, что, хотя результатом вычислений этих выражений является таблица, содержащая множество строк, эти выражения не могут быть использованы как самостоятельный оператор. Необходимо определить *cursor* над этими выражениям и получать строки одну за другой с помощью такого курсора.

Рекомендации:

1. Использование NATURAL JOIN для обозначения соединения более предпочтительно, чем остальные методы - такая формулировка будет наиболее краткой. Однако следует убедиться, что одноимённые столбцы имеют одинаковый тип.
2. Старайтесь избегать использования JOIN ON, поскольку он гарантировано возвращает результат с дублируемыми именами столбцов. Однако, если вам действительно необходимо выполнить некоторый вид экви-соединения, самостоятельно выполните надлежащее именование столбцов над полученным результатом.
3. В использования JOIN USING убедитесь, что столбцы с одинаковым именем имеют одинаковый тип.
4. В случае использования CROSS JOIN убедитесь, что отсутствуют общие имена столбцов.

Замечание по внешнему соединению

Внешние соединения LEFT, RIGHT, FULL and UNION были специально разработаны, чтобы вставлять NULL в таблицы результат, поэтому их

следует рассматривать в контексте влияния NULL на SQL модель данных. Они будут опущены в текущем контексте. Говоря реляционным языком, это некоторый вид принуждения – оно позволяет выполнить объединение таблиц, даже если типы операндов не удовлетворяют требованиям объединения. Внешнее соединение заполняет недостающие столбцы обоих операндов NULL-ами перед выполнением объединения, и таким образом операнды станут удовлетворять требованиям объединения. Но никто не мешает заполнять недостающие столбцы подходящими значениями взамен NULL.

5.2 Ссылки на таблицу / TABLE REFERENCES

Ссылка на таблицу (*table reference*) может ссылаться на некоторую таблицу, не важно именованная это таблица (например, базовая таблица или представление) или неименованная. Ссылки на таблицу используется в SQL для двух целей: они определяют операнды в разделе FROM select выражений и операнды в join выражениях. Что касается синтаксиса, то ссылка на таблицу состоит либо из join выражения (рассмотреные выше), либо из имени таблицы (a), либо из табличного выражения в скобках (b). В *Случае (a)* ссылка на таблицу может выключать необязательную конструкцию AS, целью которой является объявление *переменной области значений (range variable)* для таблицы в запросе, а также указывать имена столбцов для этой таблицы. В *Случае (b)* ссылка на таблицу *должна* включать в себя такую AS конструкцию.

table-reference

::= joined-table

| table [[AS] range-variable [(column-commalist)]

]

| (query-expression-body) [AS] range-variable [(

column-commalist)]

Список column-commalist, если он определен, должен явно указывать имя для каждого атрибута в соответствующей таблице и не может определять одно имя дважды.

Стоит отметить, что стандарт SQL не использует термин “переменные области значений”. Вместо этого он использует термин “correlation name”, но не определяет тип объектов, к которым можно обращаться, используя данное имя. “Переменные области значений” (range variable) - это общепринятый термин. SQL всегда требует, чтобы select-выражение (также как и join-выражения) формулировались в терминах переменных области значений. Если такие переменные не объявлены явно, тогда SQL предполагает наличие

неявных переменных области значений с тем же именем, что и соответствующие таблицы. Нет ничего плохого в том, чтобы объявлять такие перемены области значений там, где они не требуются явным образом, особенно в сложных выражениях они помогут сделать выражение более понятным. Однако будьте внимательны, поскольку правила области видимости имен в SQL сложны в понимании.

Исследуя вопросы выведения типов в SQL, стоит рассмотреть пример:

```
SELECT P.* , S.SNO , S.SNAME , S.STATUS
FROM P , S
WHERE P.CITY = S.CITY
AND P.PNAME > S.SNAME
```

Выражение P.PNAME > S.SNAME весьма необычно, поскольку выражение предполагает применение к результату выражения FROM, и естественно, переменные отношения P и S не являются частью этого результата. В действительности, сложно объяснить как нечто P.PNAME может появиться в выражениях WHERE и SELECT, поскольку всё должно быть описано исключительно в терминах результата выражения FROM. Стандарт SQL разъясняет это, но способом гораздо более сложным, чем описан в реляционной модели.

Область видимости переменных области значений – то есть контекст, в котором на них можно ссылаться и имя этой переменой должно быть уникальным, определён следующим образом:

- Если ссылка на таблицу присутствует в FROM выражении, тогда областью видимости переменной является select-выражение его включающее (разделы SELECT, FROM, WHERE, GROUP BY, HAVING – которые образует данное select-выражение). Исключая любые вложенные select-выражения или join-выражения, в которых другие переменные области значений могут быть объявлены с тем же именем.
- Если ссылка на таблицу присутствует в join-выражении, тогда областью видимости является это join-выражение. Исключая любые вложенные select-выражения или join-выражения, в которых другие переменные области значений могут быть объявлены с тем же именем.

Описанные замечания по поводу области видимости переменных не описывают все возможные случаи. В случае join-выражений вопрос видимости переменных области значений весьма сложный. Например, если join-выражение, содержащее ссылку на таблицу, присутствует в свою очередь

в разделе FROM – тогда область видимости таблицы расширяется до select-выражения, включающей данный раздел FROM:

```
SELECT DISTINCT SP.PNO, S.CITY
FROM SP NATURAL JOIN S
```

Причина вероятно в том, что область видимости в данном случае должна быть такой же, как если бы join выражение заменили бы двумя операндами join, разделёнными запятой.

```
SELECT ...
FROM SP, S
```

Однако одно очень нелогичное следствие из этих правил видимости показано в следующем примере.

```
SELECT DISTINCT SP.*
FROM SP NATURAL JOIN S
```

Результат выражения будет включать столбцы PNO и QTY, но не столбец SNO – потому что в результат join-выражение не содержит столбец SP.SNO (в действительности, использование SP.SNO в SELECT выражении должно было быть синтаксической ошибкой). Вместо этого результат join выражения содержит столбец SNO без всякого уточнения, что является результатом своего рода объединения столбцов S.SNO и SP.SNO.

5.3 Объединение, разность и пересечение

SQL операторы UNION, EXCEPT и INTERSECT основаны на хорошо известных операторах теории множеств: объединение, разность и пересечение. Две таблицы, которые являются операндами объединения, разности и пересечения, должны иметь одинаковую степень (то есть одинаковое количество столбцов), соответствующие столбцы должны иметь *совместимые типы данных*.

UNION и EXCEPT присутствуют в “nonjoin query expressions”, а INTERSECT присутствует в “nonjoin query terms”.

nonjoin-query-expression

    ::=   nonjoin-query-term

| query-expression-body { UNION | EXCEPT } [ALL | DISTINCT]

[CORRESPONDING [BY (column-commalist)]

] query-term

nonjoin-query-term

::= nonjoin-query-primary

| query-term INTERSECT [ALL | DISTINCT]

[CORRESPONDING [BY (column-commalist)]

] query-primary

Пересечение / INTERSECT

Пусть таблицы A и B будут результатом вычислений table-term и table-primary. Возможны следующие варианты пересечений:

1. A INTERSECT [ALL | DISTINCT] CORRESPONDING BY (column-commalist) B
2. A INTERSECT [ALL | DISTINCT] CORRESPONDING B
3. A INTERSECT [ALL | DISTINCT] B

Если не указано спецификатор ALL или DISTINCT, тогда DISTINCT предполагается по умолчанию.

Рассмотрим три случая, которые не содержат опцию ALL.

1. Пусть *Cl, C2 ... Cn* список атрибутов в BY выражении. Каждый атрибут из *Cl, C2 ... Cn* одновременно определяет атрибуты *A* и *B*. Данный случай по определению семантический эквивалентен следующему выражению:

((SELECT Cl, C2, ..., Cn FROM A)

INTERSECT

(SELECT Cl, C2, ..., Cn FROM B))

Другими словами, пусть *AC* будет таблица, полученая из *A* удалением всех столбцов, которые не отмечены в BY выражении; и таблица *BC* получена из *B* аналогичным образом. Тогда результатом пересечения будет таблица из *n* столбцов, строка *R* присутствует в результате тогда и только тогда, когда она присутствует и в *AC,* и в *BC.* Результат выражения не содержит строки дубликаты.

2. Пересечение по определению семантически эквивалентно пересечению в *Случае 1,* в котором список атрибутов включает в себя все атрибуты, общие для *A* и *B*.
3. В данном случае пересечение выполняется не согласно сравнению значений атрибутов с одинаковым именем, как в *Случае 1 и 2*, а сравниваются в *соответствии порядковой позицией данных атрибутов*. То есть, *i-ый* атрибут из *A* сравнивается с *i-ым* атрибут из *B* (для всех *i* от 1 до *n,* где *n* – степень *n;* напоминаем, что *A* и *B* должны иметь одинаковую степень). Результат также будет степени *n*, и некая строка присутствует в результате тогда и только тогда, когда она одновременно присутствует и в *A,* и в *B*. Опять же результат выражения не будет содержать дубликаты.

Если определена опция ALL в указанных трех случаях, то результат будет идентичным, за исключением того, что результат будет содержать строки дубликаты. Точнее, предположим, что строка *R* содержится *m* раз в первом операнде и *n* раз во втором операнде ($m > 0$, $n > 0$) – тогда строка *R* будет присутствовать в результате *p* раз, где *p* меньшее из *m* и *n*.

Объединение / UNION

Для операции объединения множество возможных случаев аналогично операции пересечения:

1. A UNION [ALL | DISTINCT] CORRESPONDING BY (column-commalist) B
2. A UNION [ALL | DISTINCT] CORRESPONDING B
3. A UNION [ALL | DISTINCT] B

Эти случаи определены аналогично случаям пересечения, учитывая, что для операции UNION некая строка содержится в результирующей таблице тогда и только тогда, когда она содержится как минимум в одной из таблиц операнда. По умолчанию, результат не содержит строки дубликата. В случае, если указана опция ALL, и предполагая, что строка *R* содержится *m* раз в первом операнде и *n* раз во втором операнде ($m > 0$, $n > 0$) – тогда строка *R* будет присутствовать в результате *p* раз, где $p = m + n$.

Разность / EXCEPT

Множество возможных случаев аналогично случаям пересечения:

1. A EXCEPT [ALL | DISTINCT] CORRESPONDING BY (column-commalist) B
2. A EXCEPT [ALL | DISTINCT] CORRESPONDING B

3. A EXCEPT [ALL | DISTINCT] B

И снова эти случаи определены аналогично случаям пересечения, учитывая, что для операции EXCEPT некая строка содержится в результирующей таблице тогда и только тогда, когда она содержится в первой таблице операнде и не содержится во второй таблице операнде. По умолчанию результат не содержит строки дубликаты. В случае если указана опция ALL, и предполагая, что строка R содержится m раз в первом операнде и n раз во втором операнде ($m > 0$, $n > 0$) – тогда строка R будет присутствовать в результате p раз, где p больше из $m - n$ и 0.

***Рекомендации*:**

- Убедитесь, что соответствующие столбцы имеют одинаковые имена и типы.
- Всегда определяйте COREESPONDING, где это возможно. Поскольку не все SQL продукты могут поддерживать ее, в таком случае, вам придется самостоятельно позаботиться об упорядочивание столбцов.
- Не используйте опцию BY в спецификации CORRESPONDING
- Никогда не определяйте опцию ALL, и более того, желательно использовать опцию DISTINCT явно. Зачастую пользователи указывают ALL в случаях, если знают об отсутствии дубликатов во входных таблицах, и указывают системе не тратить время, удаляя их.

Замечание по поводу синтаксиса

Дополнительно следует отметить некоторое несоответствие синтаксиса NATURAL JOIN и UNION (INTERSECT и EXCEPT). Предположим, что A и B именованные таблицы, типы которых удовлетворяют требованиям UNION, тогда следующие выражения имеют верный или не верный синтаксис:

```
A NATURAL JOIN B                                  -- верно
A UNION B                                         -- не верно
TABLE A UNION TABLE B                             -- верно
TABLE A NATURAL JOIN TABLE B                      -- не верно
SELECT * FROM A UNION SELECT * FROM B             -- верно
SELECT * FROM A NATURAL JOIN SELECT * FROM B      -- не верно
(TABLE A) UNION (TABLE B)                         -- верно
(TABLE A) NATURAL JOIN (TABLE B)                  -- не верно
(TABLE A) AS AA NATURAL JOIN (TABLE B) AS BB      -- верно
```

5.4 Простая таблица / Simple table

Как было описано ранее, простой запрос (query primary) используется для обозначения второго операнда операции пересечения.

query-primary

::= joined-table | nonjoin-query-primary

nonjoin-query-primary

::= (nonjoin-query-expression) | simple-table

simple-table

::= query-specification

| TABLE table

| table-value-constructor

Ранее мы уже рассмотрели соединения (joined table) и пересечения, объединения и разность (nonjoin query expressions). Сейчас мы остановимся на рассмотрении выражения "TABLE table" и конструкторы таблиц (table-value-constructor). Select выражения (query-specification) будут рассмотрены позже.

- Выражение "TABLE table" (где table – именованная таблица: базовая таблица или представление) по определению семантически эквивалентно выражению:
  (SELECT * FROM table)
- Конструктор таблиц представляет собой список конструкторов строк:
  VALUES row-value-expression-list,

где каждый конструктор строки определяет в точности одну строку, а конструктор таблицы определяет таблицу, которая является своего рода "UNION ALL" таких строк.

Конструктор строк в свою очередь может иметь следующую форму:

[ROW](row-value-constructor-element-conmalist) | (query-expression)

Другими словами, конструктор строк представляет собой или список компонентов строки в скобках, или табличное выражение в скобках:

1. Компонентом строки может быть скалярное выражение или одно из ключевых слов: DEFAULT или NULL. (DEFAULT и NULL допустимы, только если конструктор строки используется в INSERT выражении)

2. Табличное выражение в скобках должно вычисляться как таблица, содержащая в точности одну строку. В этом случае значением конструктора строки будет полученная строка. Стандарт SQL определяет такое выражение как строчное выражение (*row subquery).* Возникнет ошибка времени выполнения, если данное строчное выражение вернет таблицу, содержащую более одной строки. *Однако, если данное строчное выражение вернет таблицу, не содержащую ни одной строки, тогда значением конструктора строки будет строка полностью состоящая из NULL.*

Рекомендации:

- Поскольку столбцы таблицы упорядочены слева направо, и в конструкторе не поддерживается определение значения атрибута по его имени, вам необходимо убедиться, что все строки имеет соответствующий тип – то есть на i-ой позиции строки должно находиться значение i-ого атрибута таблицы.
- Убедитесь, что не описали одну строку дважды.

Пусть T будет именованная таблица, имена и типы атрибутов совпадают с таблицей поставщиков S. Тогда следует отметить, что следующие два INSERT выражения отличаются по смыслу:

1. INSERT INTO T VALUES (SELECT * FROM S WHERE SNO = 'S6')
2. INSERT INTO T (SELECT * FROM S WHERE SNO = 'S6')

Предполагая, что выражение SNO = 'S6' ложно, тогда первое выражение вставляет в таблицу T строку, полностью состоящую из NULL; в то время как второе выражение не вставляет ни одной строки.

5.5 SELECT выражение

В качестве select-выражения может быть рассмотрено табличное выражение, которое не содержит никаких JOIN, UNION, EXCEPT или INTERSECT. Хотя такие операторы могут использоваться в качестве вложенных.

query-specification

```
::=     SELECT [ ALL | DISTINCT ] select-list
        FROM table-reference-commalist
                [ WHERE search-condition ]
                        [ GROUP BY [ALL | DISTINCT] column-commalist ]
                                [ HAVING search-condition]
```

Раздел SELECT

SELECT [ALL | DISTINCT] select-list

1. Если не указан спецификатор ALL или DISTINCT, ALL предполагается по умолчанию.
2. Мы предполагаем, что результатом вычисления разделов FROM, WHERE, GROUP BY и HAVING (не важно, какие из них присутствуют в запросе, а какие нет) будет таблица *T1*.(В действительности *T1*- неименованная таблица).
3. Пусть *T2* будет таблица, полученная из *T1* вычислением списка select-list.
4. Если указано ключевое слово DISTINCT, тогда *T3* будет таблица, полученная из *T2* удалением строк дубликатов. Если DISTINCT не указано, тогда таблица *T3* будет равна *T2*.
5. *T3* будет результат вычисления данного select-выражения.

Рассмотрим список элементов select-list. Возможны два случая, причем второй случай - всего лишь сокращённая форма первого случая.

1. Элемент select-list имеет вид:

scalar-expression [[AS] column]

Скалярное выражение обычно включает один или более атрибутов таблицы *T1*. Для каждой строки таблицы *T1* вычисляется данное скалярное выражение. Список результатов вычисления таких скалярных выражений по строке таблицы *T1* является строка таблицы *T2*. Раздел AS, если он присутствует, определяет имя столбца в результирующий таблице E2 для заданного скалярного выражения. Само ключевое слово AS является необязательным.

2. Элемент select-list имеет вид:

[range-variable .] *

- Если имя переменной области значений опущено (то есть, элемент select-list представлен ‘*’), тогда этот select-item должен быть единственным в разделе SELECT. Такая форма является сокращением для списка всех атрибутов таблицы *T1*, *в порядке слева направо.*
- Если элемент select-list содержит имя переменной области значений, тогда select-item представляет список всех атрибутов указанной переменной *в порядке слева направо.* Как мы помним, в качестве переменной области значений не явно может быть использовано имя таблицы.

- Если данный элемент select-list используется при определении представлений или ограничений, тогда список атрибутов вычисляется в момент определения данных ограничений или представлений. *Следовательно, если впоследствии в соответствующие таблицы будут добавлены новые атрибуты, то это не повлияет на данные представления или ограничения.*

Рекомендации:

- Всегда указывайте опцию DISTINCT, чтобы позаботиться об удалении строк дубликатов.

Раздел FROM

FROM table-reference-commalist

Предположим, список ссылок на таблицу представлен в виде таблиц *A, B , C*. Тогда результатом вычисления раздела FROM будет таблица, которая является декартовым произведением таблиц *A, B , C*. Декартовое произведение от одной таблицы *T* по определению равно *T*.

Раздел WHERE

WHERE search-condition

Пусть *T* будет таблица, полученая после вычисления раздела FROM. Тогда результатом раздела WHERE будет таблица, полученная из *T* удалением всех строк, для которых условное выражение (search-condition) не равно *true*. Если раздел WHERE опущен, результатом будет таблица *T*.

Раздел GROUP BY

GROUP BY [ALL | DISTINCT] column-commalist

Пусть *T* будет результат вычисления разделов FROM и WHERE. Каждый атрибут в списке column-commalist в разделе GROUP BY должен быть атрибутом таблицы *T*. Результатом раздела GROUP BY будет сгруппированная таблица, то есть множество из групп строк, полученное из *T*, где строки в группе имеют одинаковое значение для атрибутов, указанных в разделе GROUP BY. Если не указан спецификатор ALL или DISTINCT, ALL предполагается по умолчанию. Если указано ключевое слово DISTINCT, тогда группы строк не будут содержать дубликатов. Такие группы сортированы внутри множества по количеству строк в группе от большего к меньшему. Отметим, что результат GROUP BY не является правильной таблицей, поэтому раздел GROPU BY никогда не используется без соответствующего

раздела SELECT, задача которого получить правильную таблицу из промежуточного результата.

Если select выражение включает в себя раздел GROUP BY, то дополнительное ограничение накладывается на раздел SELECT. Каждый элемент списка SELECT должен иметь общее значение для группы. Предполагается, что такие элементы не должен включать в себя никакие ссылки на атрибуты из таблицы T, которые не являются общими для группы, если только такие ссылки не являются аргументом (или частью выражения аргумента) для функции агрегации (функции над множеством), которая вычисляет единственное скалярное значение для коллекции скалярных значений.

Раздел HAVING

HAVING search-condition

Пусть *G* будет сгруппированная таблица, полученная после вычисления FROM, WHERE и GROUP BY разделов. Если раздел GROUP BY опущен, тогда *G* представляет собой сгруппированную таблицу, полученную после вычисления разделов FROM и WHERE, которая содержит только одну группу. Таким образом, полагается, что объявлен неявный раздел GROUP BY, который не содержит ни одного атрибута. Результатом раздела HAVING будет сгруппированная таблица, полученная из *G* удалением всех групп, для которых условное выражение (search-condition) не равно *true*. Скалярное выражение в разделе HAVING должно оперировать общими значениями для группы (как скалярные выражения в разделе SELECT, если присутствует раздел GROUP BY). Если раздел HAVING опущен, а раздел GROUP BY присутствует – тогда результатом будет таблица *G*. Если оба раздела HAVING и GROUP BY опущены – тогда результатом будет правильная (несгруппированная) таблица *T*, полученная вычислением разделов FROM и WHERE.

Замечание по поводу избыточности

Следует отметить, что разделы GROUP BY и HAVING являются в действительности избыточными. Имеется ввиду, что для каждого select-выражения, включающего данные разделы, существует эквивалентное по смыслу выражение, которое не будет включать эти разделы.

6. Заключение

В данной статье рассмотрено поведение табличных выражений стандарта SQL в сравнении с реляционными операторами. В реляционной модели понятие отношений гарантирует что: отсутствуют строки дубликаты, отсутствуют дублирующиеся столбцы, отсутствуют неименованные столбцы, отсутствует

упорядоченность столбцов, отсутствует упорядоченность строк, отсутствуют неопределённые значения (NULL). В статье были показаны случаи, в которых операторы SQL ведут себя отличным от реляционной модели образом, то есть возвращают таблицы, которые не являются отношениями. Также были описаны практики, которые позволяют вам избежать таких случаев и использовать SQL исключительно реляционным образом.

Использование SQL реляционным образом возможно в большинстве случаев. Однако, поскольку существующие реализации далеки от идеальных, временами вы обнаружите, что это приводит к потери производительности. Если в таких случаях вы склонитесь к не «истинно реляционному» использованию, тогда следует понимать, что вы идете на такие компромиссы с позиций концептуальной целостности. Всегда следует понимать правильную в теории ситуацию, и вы должны иметь веские основания уклониться от неё. Вам также следует описать эти причины, для того чтобы отказаться от них в будущем (к примеру, если новая версия продукта, который вы используете, лучше решает интересующую вас задачу), тогда вы сможете вернуться к оригинальным идеям.

7. Список литературы

[1]. C.J. Date. Hugh Darwen. A gude to the SQL standard. 4th edition. Addison-Wesley. (1997).

[2]. C.J. Date. SQL and Relational Theory: How to Write Accurate SQL Code. O'Reilly Media, Inc. (2009).

[3]. C. J. Date. Database in Depth: Relational Theory for Practitioners. Sebastopol, Calif.: O'Reilly Media, Inc. (2005).

[4]. C. J. Date, Hugh Darwen. "Foundation for Future Database Systems: The Third Manifesto", Addison-Wesley Pub Co; 2nd edition (2000).

[5]. 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.

[6]. К. Дж. Дейт. Введение в системы баз данных. Изд. 8-е./ Перев. с англ. – М.: Вильямс, 2005.

[7]. С.Д. Кузнецов. Базы данных: языки и модели. – М.: ООО «Бином-Пресс», 2008 .

Features of SQL table expressions and their compliance with the concepts of relational model.

Ivan. V. Bludov
ivan.bludov@gmail.com
MSU, CMC Department, Moscow, Russia

Abstract. The paper analyses a behavior of the SQL standard's table expressions in comparison with relational equivalents. The concept of relation of relational model presumes nonexistence of duplicate tuples, attributes of the same name, ordering of attributes, ordering of tuples, no-typed NULLs. The paper demonstrate cases where behavior of constructions of the SQL-standard violates the rules of the relational model: these constructions produce tables that are not relations (join expressions, table references, set-theoretical operators, simple tables, and select). The paper also describes practices that allow to avoid such cases and to use SQL in truly relational manner. Relational use of SQL is possible nearly always. However, existing implementations of SQL are far from ideal, and sometimes such use leads a poor performance. Non-relational use of SQL means a tradeoff between performance and conceptual consistence. It is always useful to understand a theoretically perfect variant of use and to have solid reasons to abandon it. It is also useful to document these reasons to return to the perfect variant with some new version of SQL-based DBMS that does not demonstrates problems of performance.

Keywords: SQL, the relational model, SQL query, a table expression

References

[1]. C.J. Date. Hugh Darwen. A gude to the SQL standard. 4th edition. Addison-Wesley. (1997).
[2]. C.J. Date. SQL and Relational Theory: How to Write Accurate SQL Code. O'Reilly Media, Inc. (2009).
[3]. C. J. Date. Database in Depth: Relational Theory for Practitioners. Sebastopol, Calif.: O'Reilly Media, Inc. (2005).
[4]. C. J. Date, Hugh Darwen. Foundation for Future Database Systems: The Third Manifesto, Addison-Wesley Pub Co; 2nd edition (2000).
[5]. C.J. Date. Vvedenie v sistemy baz dannyh [An Introduction to Database Systems]. Izd. 8-e./ Perev. s angl. – M.: Vil'jams, 2005 (in Russian).
[6]. S.D. Kuznetsov. Bazy dannyh: jazyki i modeli [Databases: languages and models]. – M.: OOO «Binom-Press», 2008 (in Russian).

Роль предыстории при оценке сложного объекта в управлении по прецедентам[1]

Л. Е. Карпов, В. Н. Юдин

Аннотация. Построение системы управления для нестационарных объектов – это дальнейшее развитие подхода к управлению объектами по прецедентам, разработанного в рамках более ранних работ авторов, применительно к нестационарным объектам с неполным описанием. Подход к управлению на основе прецедентов в виде цикла из трёх составляющих (оценка состояния объекта, выработка управляющего воздействия, оценка состояния после воздействия) для нестационарного объекта потребовал существенной доработки из-за неоднозначности в оценке объекта. За время оценки и выработки адекватного управляющего воздействия состояние объекта может самопроизвольно измениться, и заранее подготовленное воздействие станет непригодным. На нынешнем этапе упор сделан на разработку методики, позволяющей снизить неоднозначность оценки объекта до и после воздействия. Источником дополнительной информации может служить предыдущее поведение объекта.

Ключевые слова: система управления, вывод по прецедентам, добыча данных, база прецедентов, нестационарный объект, поведение объекта, мера близости, неполнота описания, дифференциальный ряд состояний.

Введение

Новая версия исследовательской системы поддержки решений, создаваемой в ИСП РАН (см. [1-15]), и существующей на её базе системы поддержки врачебных решений «Спутник Врача», связана с решением задачи управления нестационарными, не поддающимися формализации, объектами с неполным описанием. Основным примером таких объектов является организм человека, состояния которого в силу ограничений по времени и ресурсам могут описываться не полностью. Для такого рода объектов не существует и вряд ли

[1] Работа поддержана грантами Российского фонда фундаментальных исследований № 12-01-00780, № 12-07-00214

будет создано в обозримом будущем адекватное описание в виде математической модели.

При попытках управления такими объектами возникает ситуация, принципиально отличающаяся от “классической”. Вместо точного вида математической модели объекта можно опираться только на доступную информацию о состояниях объекта управления, управляющих воздействиях на него и результатах воздействий. В теории вывода по прецедентам это соответствует трём составляющим понятия “случай” (прецедент) – описанию проблемы, применённому решению и результату применения этого решения. Совокупность таких случаев образует так называемую “базу прецедентов”.

Построение системы управления на данном этапе – это дальнейшее развитие подхода к управлению по прецедентам, разработанного авторами ранее, применительно к нестационарным объектам с неполным описанием. Подход к управлению на основе прецедентов в виде цикла из трёх составляющих (оценка состояния объекта, выработка управляющего воздействия, оценка состояния после воздействия) для нестационарного объекта потребовал существенной доработки из-за неоднозначности в оценке объекта.

Значительная часть подходов к управлению, в частности, управление физическими объектами, строится на предположении о том, что оценка состояния объекта и выработка управляющего воздействия на него – одномоментные события. Однако есть ряд приложений, например, медицина, где оценка объекта и подготовка управляющего воздействия могут занимать значительное время, за которое состояние объекта может самопроизвольно измениться, что сделает подготовленное воздействие непригодным. Причина трансформации объекта во времени – в наличии как внутренних, так и внешних (не зависящих от процесса управления) факторов.

Если принять за основу признаковое описание объекта, когда объект описывается набором своих характеристик, то к моменту применения воздействия объект будет иметь, вообще говоря, отличный от предыдущего набор признаков. Такая ситуация может возникать при управлении сложными объектами любой природы, но мы приведём пример из медицины. При остром аппендиците боли могут возникать в правой подвздошной области, но по мере перехода воспаления с аппендикса на брюшину и развития гангренозной стадии, занимающего несколько часов, боли могут уменьшиться или пройти, создавая впечатление “мнимого благополучия”. В целом, набор симптомов резко изменяется. Кроме того, некоторые важные признаки, которые являются существенными для позиционирования (правильной оценки) объекта, из-за нехватки средств или времени могут отсутствовать.

Особенность новой версии системы – адекватная подготовка управляющего воздействия для нестационарных объектов с неполным описанием. Упор был сделан на разработку методики, позволяющей снизить неоднозначность оценки объекта до и после воздействия, вызванную вполне естественной причиной: неполнотой описания объекта. Такая ситуация встречается на

практике во многих приложениях, в частности, в медицине, чаще всего (но не только) из-за нехватки времени и ресурсов. Основная идея предложенного метода заключается в том, что предыдущее поведение объекта может служить источником дополнительной информации для оценки объекта в текущем состоянии. Это утверждение вызвало интерес и понимание в кругу врачей разного профиля (всегда учитывавших предысторию болезни и лечения пациента, так называемый анамнез), когда обсуждалась очередная версия программной системы поддержки врачебных решений "Спутник врача", в которой реализуются эти принципы.

Метод

Основная проблема в выводе по прецедентам – выбор наиболее подходящего прецедента, который упирается в оценку схожести прецедента и текущего случая. В наших более ранних работах был введён метод оценки схожести в условиях неполноты описания объектов, основанный на разбиении базы прецедентов на классы состояний.

При оценке текущего случая точка, соответствующая ему, сравнивается с проекциями классов на пространство его признаков. Близкими считаются "аналоги" – прецеденты, принадлежащие классу, в который попадает случай. Если случай при его оценке попал в область пересечения классов, то близкими к нему будут аналоги из всех соответствующих классов, также находящиеся в этой области пересечения. В конечном счёте, аналоги самого высокого ранга находятся в области пересечения всех классов, которые образуют так называемый "дифференциальный ряд" случая. В зависимости от сложности пересечения, мы можем разделить все аналоги на группы (*рис. 1*). Приведём определение меры близости:

Расстояние между текущим случаем и прецедентом равно разности количества классов, куда попал текущий случай, и количества классов из этого числа, в котором находится прецедент.

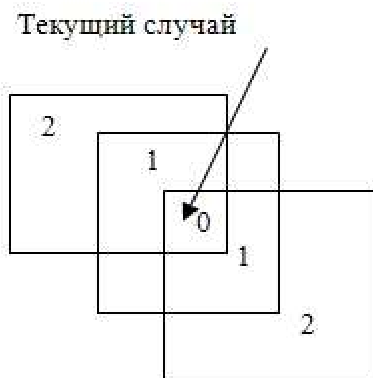


Рис. 1. Мера близости для текущего случая. Дифференциальный ряд.

Для оценки неполностью описанных объектов введено понятие “дифференциального ряда” состояний объекта – совокупности проекций классов в указанном пространстве, в которые попал текущий случай.

Если нельзя однозначно оценить состояние объекта, можно попытаться расширить его признаковое пространство, дифференцировав классы, в которые оно входит, добавлением нового признака. Однако это не всегда возможно, в частности, из-за дефицита времени, средств или оборудования.

Понятие цели управления не всегда отождествляется с достижением конкретного состояния. Целью может быть управляемое поведение, учитывающее переходы объекта из одного класса состояний в другой. Так, при лечении хронических заболеваний, задача восстановления больного органа – невыполнима. Тогда целью управления может стать замедление процесса дегенерации рабочей ткани. Поэтому, говоря о цели, мы имеем в виду не какое-то одно конкретное состояние, а оптимальное поведение объекта, выражающееся в принадлежности объекта в заданный конкретный момент времени к тому или иному классу. Требуется найти алгоритм управления, обеспечивающий достижение этой цели за конечное число управляющих воздействий.

Чтобы подобрать воздействие для текущего случая, в базе прецедентов отыскивается прецедент со схожим исходным состоянием C_{i-1} (первая составляющая понятия “прецедент”), схожим конечным состоянием C_i (третья составляющая понятия “прецедент”), затем из него заимствуется воздействие (вторая составляющая), которое предположительно должно перевести наш объект в нужное состояние (*рис. 2*).

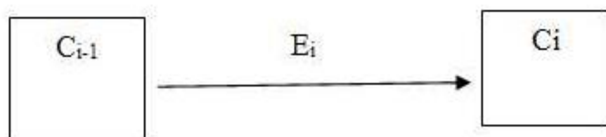


Рис. 2. Структура прецедента.

При выборе прецедентов и их управляющих воздействий могут встретиться случаи, наборы признаков которых в начальном состоянии совпадают, но для которых одно и то же воздействие приводит к разным конечным состояниям (*рис. 3*). Наличие таких случаев в одной базе прецедентов связано с тем, что состояние объекта не всегда описывается достаточно полно, и в этих случаях отсутствует неучтенный разделяющий признак. При наличии такого признака, исходные состояния при занесении их в базу прецедентов трактовались бы как разные. По этой же причине, для не полностью описанных случаев

невозможно гарантировать, что воздействие, заимствованное у прецедента, приведёт объект в нужное состояние.

Неоднозначность в оценке объекта, естественно, отражается и на выборе управляющих воздействий. Так как состояние текущего объекта описывается дифференциальным рядом, то для каждого из классов этого ряда в базе прецедентов существует, вообще говоря, своё воздействие, переводящее объект из этого класса в класс, соответствующий цели управления на текущем шаге. На самом же деле, исходя из предполагаемой оценки случая, выбирается одно воздействие. Таким образом, при отсутствии разделяющего признака, нет полной уверенности, что объект правильно идентифицирован, а выбранное воздействие приведёт его именно в то состояние, которое необходимо (*рис. 4*).

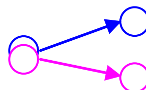


Рис. 3. Совпадение состояний из-за неполноты описания случая.

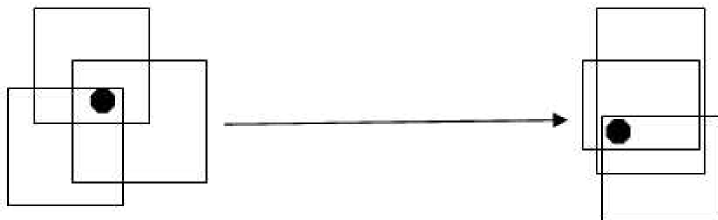


Рис. 4. Элементарный шаг управления.

Так, в медицине практикующему врачу приходится действовать на свой страх и риск, и при неполной идентификации случая заболевания назначать лечение, которое, вообще говоря, может оказаться неадекватным. Но даже в этом случае перевод объекта в состояние, которое не соответствует цели лечения, может оказаться информативным, так как сопоставление полученного состояния с базой прецедентов может прояснить исходное состояние случая.

Чем больше прецедентов в базе, тем больше спектр их возможных значений, тем выше вероятность найти "наиболее подходящий" прецедент, и выше качество принимаемого решения.

Оценивать объект, основываясь только на его текущем состоянии – это значит, не использовать всех возможностей для оценки. Принятие во внимание предыстории объекта – его состояний и реакций на воздействия – может снизить степень неоднозначности оценки текущего состояния и, следовательно, ограничить выбор воздействий (достаточно вспомнить понятие "анамнез" в медицине).

Особенность выполняемого авторами исследовательского проекта и новой версии системы управления в том, что управление объектом (оценка состояния и выработка управляющего воздействия) основывается не только на текущем состоянии, но и на всей предыстории поведения объекта.

Чтобы прояснить метод, рассмотрим снова элементарный шаг управления (*рис. 5*).

Предыдущее состояние ($i-1$) Текущее состояние (i)

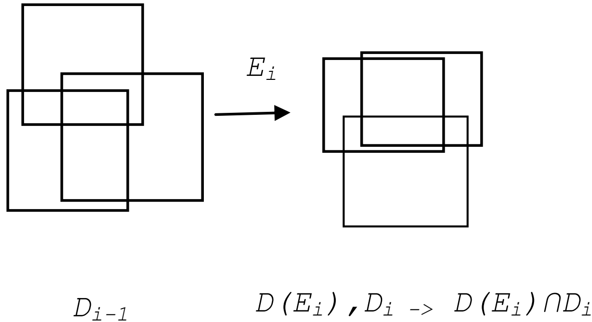


Рис. 5. Формирование дифференциального ряда.

Введём ряд обозначений:

Пусть текущее состояние объекта – i, а предыдущее – $i-1$:

- D_i – дифференциальный ряд, основанный (только лишь) на признаках i-го состояния объекта,
- D_{i-1} – дифференциальный ряд, основанный на признаках $i-1$ -го (предыдущего) состояния объекта,
- E_i – воздействие, которое перевело объект в i -е состояние.

Если представить вместо D_i дифференциальный ряд, построенный виртуальным применением оптимальных воздействий для каждого из классов дифференциального ряда D_{i-1} из базы прецедентов, то это будет, вообще говоря, другой ряд, назовём его «предполагаемый» дифференциальный ряд i-го состояния и обозначим $D(E_i)$.

До воздействия мы имеем только предполагаемый ряд $D(E_i)$, после воздействия – ряд D_i, построенный по признакам i-го состояния. Очевидно, что искомый класс для i-го состояния должен находиться в пересечении множеств $D(E_i) \cap D_i$.

Если такую процедуру делать в каждом состоянии объекта, начиная с первого, то вместо рядов D_k ($k= 1, \dots i$) мы будем иметь ряды $D(E_k) \cap D_k$, каждый из

которых меньше, либо равен D_k (если это не так, значит, база прецедентов недостаточно полна). Это позволит на каждом шаге управления объектом сохранять информацию обо всех предшествующих шагах.

Разработанная методика ограничения дифференциального ряда используется в новой версии системы поддержки решений, а также в её медицинском варианте – программе поддержки врачебных решений в диагностике и выборе лечения “Спутник Врача”.

Заключение

Сложность реальных объектов, дополняемая неопределенностью их функционирования и неполнотой описания, затрудняет задачу построения адекватной модели поведения, что увеличивает значимость интеллектуальных методов управления. Основным примером таких объектов является организм человека, состояния которого в силу ограничений по времени и ресурсам могут описываться не полностью.

Для управления такими объектами авторы предложили использовать нетрадиционный и оригинальный подход, основанный на сочетании методов вывода по прецедентам (Case-Based Reasoning) и методов добычи данных (Data Mining).

По мере развития и совершенствования системы подход к управлению на основе прецедентов в виде цикла из трёх составляющих для нестационарных объектов потребовал существенной доработки из-за неоднозначности в оценке объекта. Разработана методика ограничения дифференциального ряда состояний объекта, позволяющая, начиная с первого шага управления, подобно голограмме, отображать на текущем шаге всё предыдущее состояние объекта.

Большой интерес к системам интеллектуального управления подтверждается огромным количеством публикаций и проводимых международных конференций, посвященных этим проблемам. Результаты, полученные авторами при реализации ранее поддержанных проектов, докладывались на представительных международных научных конференциях. Результаты обсуждений и интерес, вызванный докладами, показывает, что ведущаяся работа находится на современном мировом уровне.

Литература

[1]. Л. Е. Карпов, В. Н. Юдин, «Методы добычи данных при построении локальной метрики в системах вывода по прецедентам», Препринт Института системного программирования РАН, № 18, 2006, стр 1-42, *http://citforum.ru/consulting/BI/data_mining/*

[2]. Л. Е. Карпов, В. Н. Юдин, «Адаптивное управление по прецедентам, основанное на классификации состояний управляемых объектов», Труды Института системного программирования РАН (ИСП РАН), т. 13, № 2, Институт системного программирования РАН, 2007, стр. 37-57, ISBN 5-89823-026-2. ISSN 2220-6426

(Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2007/13/2/isp_2007_13_2_37.pdf*, *http://www.citforum.ru/consulting/BI/karpov/*

[3]. Л. Е. Карпов, В. Н. Юдин, «Интеграция методов добычи данных и вывода по прецедентам в медицинской диагностике и выборе лечения», Математические методы распознавания образов. Сборник докладов 13-й Всероссийской конференции, октябрь 2007, МАКС Пресс, 2007, стр. 589-591, ISBN 978-5-317-02060-6, *http://www.mmro.ru/files/mmro13.pdf*

[4]. В. Н. Юдин, Л. Е. Карпов, А. В. Ватазин, «Процесс лечения как адаптивное управление человеческим организмом в программной системе "Спутник врача"», Альманах клинической медицины, т. 17, № 1, МОНИКИ, 2008, стр. 262-265, ISBN 978-5-98511-032-6 (Т. XVII, ч. 1), ISBN 5-9900012-1-5, *http://www.isan.troitsk.ru/win/block1.pdf*

[5]. В. Н. Юдин, Л. Е. Карпов, А. В. Ватазин, «Методы интеллектуального анализа данных и вывода по прецедентам в программной системе поддержки врачебных решений», Альманах клинической медицины, т. 17, № 1, МОНИКИ, 2008, стр. 266-269, ISBN 978-5-98511-032-6 (Т. XVII, ч. 1), ISBN 5-9900012-1-5, *http://www.isan.troitsk.ru/win/block1.pdf*

[6]. Л. Е. Карпов, А. Н. Томилин, В. Н. Юдин, «Репликация и валидация в распределенной системе поддержки врачебных решений», Труды Всероссийской научной конференции "Научный сервис в сети Интернет: решение больших задач", МГУ, 2008, стр. 387-392, ISBN 978-5-211-05616-9, *http://agora.guru.ru/abrau2008/pdf/043.pdf*

[7]. Л. Е. Карпов, В. Н. Юдин, А. В. Ватазин, «Виртуальная интеграция и консолидация знаний в распределенной системе поддержки врачебных решений», Научно-практическая конференция ЦФО РФ «Актуальные вопросы гемафереза, хирургической детоксикации и диализа», МОНИКИ, 2009, стр. 36. ISBN 978-5-98511-054-8.

[8]. А. В. Ватазин, Л. Е. Карпов, В. Н. Юдин, «Виртуальная интеграция и консолидация знаний в распределенной системе поддержки врачебных решений», Альманах клинической медицины, т. 20, 2009, стр. 83-86. ISSN 2072-0505.

[9]. А. В. Ватазин, Л. Е. Карпов, В. Н. Юдин, «Многопараметрическое управление сложным объектом в программной системе поддержки врачебных решений», III Евразийский конгресс по медицинской физике и инженерии "Медицинская физика – 2010", 21-25 июня 2010 г., т. 4, МОНИКИ, 2010, стр. 415-417.

[10]. А. В. Ватазин, В. Н. Юдин, Л. Е. Карпов, «Многопараметрическое управление сложным объектом в программной системе поддержки врачебных решений», Ежегодная научно-практическая конференция Центрального Федерального округа РФ "Актуальные вопросы заместительной почечной терапии, гемафереза и трансплантационной координации", МОНИКИ, 2010, стр. 8. ISBN 978-5-98511-091-3.

[11]. Leonid Karpov, Valery Yudin, «The Case-Based Software System for Physician's Decision Support», Sami Khari, Lenka Lhotska, Nadia Pisanti (eds.), "Information Technology in Bio- and Medical Informatics, ITBAM 2010", Proceedings of the First International Conference, Bilbao, Spain. Lecture Notes in Computer Science Sublibrary: SL 3, Springer Verlag, Berlin, Heidelberg, 2010, pp. 78-85. ISSN 0302-9743.

[12]. L. E. Karpov, V. N. Yudin, A. V. Vatazin, «Multi-Parametric Control of Complex Object in the Program System for Physician's Decision Support», Proceedings of the 12-th International Workshop on Computer Science and Information Technologies

(CSIT'2010), Russia, Moscow – St. Petersburg, September 13-19, v. 1, Ufa State Aviation Technical University, 2010, pp. 28-30.
[13]. Л. Е. Карпов, В. Н. Юдин, «Обмен данными в распределённой системе поддержки решений», Труды Института системного программирования, т. 19, Институт системного программирования РАН, 2010, стр. 71-80, ISBN 978-0-543-57630-9, ISBN 978-5-4221-0085-9, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2010/19/isp_19_2010_71.pdf*
[14]. Л. Е. Карпов, В. Н. Юдин, «Многопараметрическое управление на основе прецедентов», Труды Института системного программирования, т. 19, Институт системного программирования РАН, 2010, стр. 81-93, ISBN 978-0-543-57630-9, ISBN 978-5-4221-0085-9, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2010/19/isp_19_2010_81.pdf*
[15]. А. В. Ватазин, Л. Е. Карпов, Ю. Г. Сметанин, В. Н. Юдин, «Программная система поддержки врачебных решений с гибридной архитектурой на основе правил и прецедентов», V Троицкая конференция "Медицинская физика и инновации в медицине (ТКМФ-5)", Сборник материалов, том 2, стр. 425-427. 2012, РАН, Троицкий Научный Центр, ISBN 978-5-89513-272-2

State prehistory for complex object estimation in a control system based on cases

L. E. Karpov, V. N. Yudin
ISP RAS, Moscow, Russia

Abstract. Building a control system for non-stationary objects is a subsequent step in developing an approach to controlling not fully described objects using Case-Based Reasoning. Initially this approach was offered in some earlier projects of the authors. The Case-Based Reasoning style of controlling complex objects shows us the cycle with three constituent elementary steps: estimating the object state, generating a controlling action, estimating the object state after that action. For non-stationary objects this traditional approach should be dramatically improved because now the state estimation cannot be done unambiguously. While we are performing these steps the object state may be changed spontaneously. In this case the action that was prepared beforehand becomes inapplicable. In this project the authors are trying to develop a method that should decrease the level of ambiguity in estimating the state object before and after the action. The previous object behavior is the source of additional information - that is the main idea of the method offered here. The authors have developed the technique of limiting the differential set. This technique allows us to fully represent all the previous states of the object starting from the very first step of control process.

Keywords: control system, Case-Based Reasoning, Data Mining, case base, non-stationary process, object behavior, measure of closeness, incompleteness of description, differential set of states, state prehistory.

References

[1]. L. E. Karpov, V. N. Yudin. Metody dobychi dannykh pri postroenii lokal'noj metriki v sistemakh vyvoda po pretsedentam [Data Mining methods in building local metrics for Case-Based Reasoning systems], Preprint ISP RAN [Preprint of ISP RAS], no. 18, 2006, pp. 1-42, http://citforum.ru/consulting/BI/data_mining/ (in Russian).

[2]. L. E. Karpov, V. N. Yudin. Adaptivnoe upravlenie po pretsedentam, osnovannoe na klassifikatsii sostoyanij upravlyaemykh ob"ektov [Case-Based Reasoning adaptive control with classification of states of objects under control], Trudy ISP RAN [The Proceedings of ISP RAS], vol. 13, no. 2, 2007, pp. 37-57, ISBN 5-89823-026-2. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), http://www.ispras.ru/ru/proceedings/docs/2007/13/2/isp_2007_13_2_37.pdf, http://www.citforum.ru/consulting/BI/karpov/ (in Russian).

[3]. L. E. Karpov, V. N. Yudin. Integratsiya metodov dobychi dannykh i vyvoda po pretsedentam v meditsinskoj diagnostike i vybore lecheniya [Integration of data Mining and Case-Based Reasoning methods in medical diagnostics and treatment choosing], Sbornik dokladov 13-j Vserossijskoj konferentsii Matematicheskie metody raspoznavaniya obrazov [Proc. of 13-th All-Russian conference Math. methods of pattern recognition], October 2007, MAKS Press, 2007, pp. 589-591, ISBN 978-5-317-02060-6, http://www.mmro.ru/files/mmro13.pdf (in Russian).

[4]. V. N. Yudin, L. E. Karpov, A. V. Vatazin. Protsess lecheniya kak adaptivnoe upravlenie chelovecheskim organizmom v programmnoj sisteme "Sputnik vracha" [Process of patient treatment as an adaptive control of human being organism in software system "Doctor's Partner"], Al'manakh klinicheskoj meditsiny [Almanac of Clinical Medicine], vol. 17, no. 1, МОНИКИ [Moscow Regional Scientific Research Clinical Institute], 2008, pp. 262-265, ISSN 2072-0505, ISBN 978-5-98511-032-6, ISBN 5-9900012-1-5, http://www.isan.troitsk.ru/win/block1.pdf (in Russian).

[5]. V. N. Yudin, L. E. Karpov, A. V. Vatazin. Metody intellektual'nogo analiza dannykh i vyvoda po pretsedentam v programmnoj sisteme podderzhki vrachebnykh reshenij [Application of Data Mining and Case-Based Reasoning in software system for physician's decision support], Al'manakh klinicheskoj meditsiny [Almanac of Clinical Medicine], vol. 17, no. 1, MONIKI [Moscow Regional Scientific Research Clinical Institute], 2008, pp. 266-269, ISSN 2072-0505, ISBN 978-5-98511-032-6, ISBN 5-9900012-1-5, http://www.isan.troitsk.ru/win/block1.pdf (in Russian).

[6]. L. E. Karpov, A. N. Tomilin, V. N. Yudin. Replikatsiya i validatsiya v raspredelennoj sisteme podderzhki vrachebnykh reshenij [Data replication and validation in distributed software system for physician's decision support], Trudy Vserossijskoj nauchnoj konferentsii "Nauchnyj servis v seti Internet: reshenie bol'shikh zadach" [Proc. All-Russian scientific conference "Scientific Service in Internet: solving of huge problems"], MGU [Moscow State University, 2008, pp. 387-392, ISBN 978-5-211-05616-9, http://agora.guru.ru/abrau2008/pdf/043.pdf (in Russian).

[7]. L. E. Karpov, A. V. Vatazin, V. N. Yudin. Virtual'naya integratsiya i konsolidatsiya znanij v raspredelennoj sisteme podderzhki vrachebnykh reshenij [Virtual knowledge integration and consolidation in software system for physician's decision support], Trudy Nauchno-prakticheskaya konferentsiya TSFO RF «Aktual'nye voprosy gemafereza, khirurgicheskoj detoksikatsii i dializa [Proc. of research and practical conference 'Actual problems of hemapheresis, surgery detoxication and dialysis'], MONIKI [Moscow Regional Scientific Research Clinical Institute], 2009, pp. 36. ISBN 978-5-98511-054-8 (in Russian).

[8]. A. V. Vatazin, L. E. Karpov, V. N. Yudin. Virtual'naya integratsiya i konsolidatsiya znanij v raspredelennoj sisteme podderzhki vrachebnykh reshenij [Virtual knowledge integration and consolidation in software system for physician's decision support], Al'manakh klinicheskoj meditsiny [Almanac of Clinical Medicine], vol. 20, 2009, pp. 83-86. ISSN 2072-0505 (in Russian).

[9]. A. V. Vatazin, L. E. Karpov, V. N. Yudin. Mnogoparametricheskoe upravlenie slozhnym ob"ektom v programmnoj sisteme podderzhki vrachebnykh reshenij [Multiparametric object control in software system for physician's decision support], III Evrazijskij kongress po meditsinskoj fizike i inzhenerii "Meditsinskaya fizika – 2010" [Third Euro-Asia congress for medical physics], 21-25 of June 2010, vol. 4, MONIKI [Moscow Regional Scientific Research Clinical Institute], 2010, pp. 415-417 (in Russian).

[10]. A. V. Vatazin, L. E. Karpov, V. N. Yudin. Mnogoparametricheskoe upravlenie slozhnym ob"ektom v programmnoj sisteme podderzhki vrachebnykh reshenij [Multiparametric object control in software system for physician's decision support], Ezhegodnaya nauchno-prakticheskaya konferentsiya TSentral'nogo Federal'nogo okruga RF "Aktual'nye voprosy zamestitel'noj pochechnoj terapii, gemafereza i transplantatsionnoj koordinatsii" [Proc. of Annual research and practical conference 'Actual problems of replacement therapy, hemapheresis, and transplantation

coordination'], MONIKI [Moscow Regional Scientific Research Clinical Institute], 2010, стр. 8. ISBN 978-5-98511-091-3. (in Russian).

[11]. Leonid Karpov, Valery Yudin. The Case-Based Software System for Physician's Decision Support. Sami Khari, Lenka Lhotska, Nadia Pisanti (eds.), "Information Technology in Bio- and Medical Informatics, ITBAM 2010", Proceedings of the First International Conference, Bilbao, Spain. Lecture Notes in Computer Science Sublibrary: SL 3, Springer Verlag, Berlin, Heidelberg, 2010, pp. 78-85. ISSN 0302-9743.

[12]. L. E. Karpov, V. N. Yudin, A. V. Vatazin. Multi-Parametric Control of Complex Object in the Program System for Physician's Decision Support, Proceedings of the 12-th International Workshop on Computer Science and Information Technologies (CSIT'2010), Russia, Moscow – St. Petersburg, September 13-19, v. 1, Ufa State Aviation Technical University, 2010, pp. 28-30.

[13]. L. E. Karpov, V. N. Yudin. Obmen dannymi v raspredelyonnoj sisteme podderzhki reshenij [Data exchange in distributed software system for decision support], Trudy ISP RAN [The Proceedings of ISP RAS], vol. 19, 2010, pp. 71-80, ISBN 978-0-543-57630-9, ISBN 978-5-4221-0085-9, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), http://www.ispras.ru/ru/proceedings/docs/2010/19/isp_19_2010_71.pdf (in Russian).

[14]. L. E. Karpov, V. N. Yudin. Case-based multi-parametric object control, Trudy ISP RAN [The Proceedings of ISP RAS], vol. 19, 2010, pp. 81-93, ISBN 978-0-543-57630-9, ISBN 978-5-4221-0085-9, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), http://www.ispras.ru/ru/proceedings/docs/2010/19/isp_19_2010_81.pdf (in Russian).

[15]. A. V. Vatazin, L. E. Karpov, Y. G. Smetanin, V. N. Yudin. Programmnaya sistema podderzhki vrachebnykh reshenij s gibridnoj arkhitekturoj na osnove pravil i pretsedentov [Software system for physician's decision support with architecture based on rules and cases], V Troitskaya konferentsiya "Meditsinskaya fizika i innovatsii v meditsine (TKMF-5)", Sbornik materialov [Proc. of Fifth conference 'Medical physics and innovations in medicine'], vol. 2, pp. 425-427. 2012, RAS, Troitsk Scienfic Centre, ISBN 978-5-89513-272-2 (in Russian).

Гибридный подход к построению систем поддержки решений[1]

В. Н. Юдин, Л. Е. Карпов

Аннотация. Системы поддержки принятия решений, в которых результаты вывода по правилам дополняют результаты рассуждений по прецедентам – очередной шаг вперед по сравнению с моделями, поддерживавшими только одну из парадигм знаний. В существующих на данный момент гибридных системах основным инструментом вывода являются порождающие правила, а прецеденты используются лишь для обработки исключений. В описываемом подходе к поддержке решений для второй очереди исследовательской системы ИСП РАН и созданной на ее основе системе поддержки врачебных решений «Спутник Врача», рассуждения по правилам и прецедентам дополняют друг друга в условиях неоднозначной оценки неполностью описанного случая. Отсутствие существенного признака для одного из правил зачастую делает непригодным применение самих порождающих правил. На текущем этапе проекта разработана методика двухэтапной оценки случая, где сначала используется прецедентный подход с целью получить представление о возможной принадлежности случая тому или иному классу (так называемый дифференциальный ряд состояний случая). На основе этой информации, на втором этапе, уже по правилам вывода проводится обратный логический вывод от возможного заключения, как гипотезы, – к фактам, подтверждающим эту гипотезу. Это снижает затраты на поиск недостающих признаков и существенно ограничивает перебор ветвей при движении по цепи правил.

Ключевые слова: система поддержки принятия решения, вывод по прецедентам, добыча данных, вывод на основе правил, база прецедентов, мера близости, классы эквивалентности, пространство признаков, неполнота описания, дифференциальный ряд состояний, обратный логический вывод.

Введение

В институте системного программирования несколько лет проводятся исследования различных подходов к построению систем поддержки принятия решений при работе со сложными объектами, описания которых плохо формализуются и часто остаются неполными, не описывающими все

---

[1] Работа поддержана грантами Российского фонда фундаментальных исследований № 12-01-00780, № 12-07-00214

особенности поведения объектов с необходимым уровнем детализации. Эта работа неоднократно поддерживалась Российским фондом фундаментальных исследований (см. [1-15]). В рамках общего исследования авторами строится пример системы поддержки решений, в которой в качестве объекта рассматривается организм человека. Организм человека – объект действительно сложный для понимания, хотя его исследования продолжаются уже не одну тысячу лет. Однако знаний, необходимых для проведения точной диагностики и исправления выявленных отклонений от нормы, пока явно недостаточно.

Первые варианты создаваемой системы строились на основе сочетания методов добычи данных (Data Mining) и логического вывода на основе прецедентов (Case-Based reasoning). В последнее время в рассмотрение попали также методы логического вывода знаний на основе правил, что в ещё большей степени позволяет называть создаваемую систему гибридной. Часто методы вывода по прецедентам и правилам противопоставляют друг другу, однако их сочетание позволяет получить в гибридной системе дополнительные преимущества.

Вывод, основанный на правилах (продукциях), позволяет интегрировать знания в систему с помощью правил описательной логики. Очень часто вывод на основе правил используется в реактивных системах, то есть в системах, которые отслеживают в одном или нескольких приложениях факты возникновения интересующих их событий, в частности, событий, сигнализирующих о критических условиях. Фиксация события приводит к выполнению действия, управляющего ситуацией. Для этого правила в программных системах обычно записываются в виде пар “событие-действие”. В некоторых моделях правило может содержать ещё и условие, то есть логический предикат над параметрами события, вычисляемый при обнаружении события. Предикат определяет, нужно ли выполнять указанное действие. В таких случаях говорят, что модель следует парадигме “событие-условие-действие” (event-condition-action – ECA):

«ПРИ <событие> ЕСЛИ (условие) ТО (заключение)»

Правила записываются в некотором порядке, который облегчает их понимание, но реально между ними нет отношения упорядоченности.

Модели, основанные на правилах, гораздо менее структурированы и меньше отражают порядок в общем потоке действий, чем многие другие подходы. Они больше подходят для тех ситуаций, в которых имеется не очень много ограничений на выполняемые действия, и где, следовательно, небольшое число правил может определять всю схему взаимодействий составных частей комплексной системы. Правила позволяют также моделировать асинхронные события, то есть события, которые могут произойти на любой стадии процесса управления, что делает их вполне пригодными для определения логики управления исключительными ситуациями, которые по своей природе асинхронны.

В самом общем виде данные, используемые при записи и обработке правил, можно разделить на данные, связанные с отдельными приложениями, и управляющие данные. Прикладные данные – это, например, параметры, посылаемые или получаемые в сообщениях, или извлекаемые из приборов при проведении измерений. Управляющие данные используются для вычисления условий перехода, а в общем случае это те данные, которые используются при проведении технологического процесса (например, при медицинской диагностике или управлении производственным процессом). В большинстве систем управляющих данных немного, их типы ограничены строковыми, целыми или вещественными типами, хотя иногда могут использоваться и составные типы – массивы или структуры. Значения управляющих данных обычно непосредственно извлекаются из сообщений и измерительных приборов.

Прикладные данные более сложны и разнообразны. Один из подходов для работы с ними заключается в их трактовке как чёрных ящиков, которые можно только передавать от одного вида деятельности другому. Другой подход пытается сделать все данные явными, вставляя соответствующие определения данных в состав описания правил.

Чёрные ящики имеют свои преимущества. Одно из них – в том, что модель может игнорировать обмены сложными данными между видами деятельности. Ранние системы из прагматических соображений использовали именно этот подход.

Условия в записи правил представляют собой посылки правил и состоят из одной или нескольких пар “атрибут-значение”, соединённых логическими связками “И”, “ИЛИ”, “НЕ”. Заключение выражает либо некоторый факт, либо указание на определённое действие, подлежащее исполнению. Механизм логического вывода отыскивает правила, в состав которых входят введённые факты, и актуализирует те, которым эти факты соответствуют. Правило срабатывает, если имеет место совпадение представленного факта с условием правила, при этом заключение сработавшего правила также становится фактом.

По мере срабатывания правил может быть подтверждено или опровергнуто конечное заключение. Метод рассуждений от фактов – к заключениям носит название прямой логический вывод. При обратном логическом выводе рассуждение идёт от заключения, как гипотезы – к фактам, подтверждающим эту гипотезу.

Уже давно получили популярность при описании принятия решений в медицине деревья решений [16]. Деревья, имеющие одну точку входа, называемую корнем, можно считать частным случаем правил вывода.

Вывод на основе прецедентов представляет собой метод принятия решений, моделирующий человеческие рассуждения. Метод использует знания о предыдущих ситуациях или случаях (прецедентах), которыми могут быть встречавшиеся ранее проблемы или типичные случаи, а также принятые в

связи с ними решения. При рассмотрении новой проблемы (текущего случая) находится похожий прецедент в качестве аналога. Вместо того чтобы искать решение каждый раз сначала, можно попытаться использовать его решение, возможно, адаптировав к текущему случаю.

Накопленная совокупность прецедентов, наполняемая как смоделированными случаями, так и случаями из практики, образует так называемую «базу прецедентов». Система, построенная по такому принципу, является самообучаемой: чем больше прецедентов содержится в базе, тем больше спектр их возможных значений, тем выше вероятность найти "наиболее подходящий" прецедент, следовательно, выше качество принимаемого решения.

Каждая из моделей обладает как преимуществами, так и недостатками. Идея вывода по правилам позволяет получить решение, не требующее доказательств, но она подразумевает наличие хорошо формализованной задачи. Основные достоинства таких систем – лёгкость восприятия каждого отдельного правила, несложность процесса внесения изменений, простота логического вывода, модульность представления знаний, независимость правил друг от друга. К недостаткам таких систем можно отнести отличие от структур знаний, свойственных человеку, неясность взаимных отношений правил, трудность понимания логики процессов, описанных большим числом правил. Такая модель представляет один из наиболее редких подходов к решению проблемы.

Основное достоинство систем на основе прецедентов – простота и лёгкость реализации, но они не создают моделей и правил, обобщающих предыдущий опыт. Такие системы эффективно работают только при наличии большой базы прецедентов. Одна из основных проблем – выбор подходящих прецедентов, который упирается в оценку схожести прецедента и текущего случая.

Использование признакового описания, когда объект определяется набором своих характеристик, недостаточно характеризует его в системах подобного рода. При оценке объекта набор его признаков мало что даёт для исследователя до тех пор, пока этот объект не будет сравнен с множеством других подобных объектов на обладание общими, либо разными характеристиками. Для этого используют дополнительные знания о проблемной области, или, как их еще называют, фоновые знания. Один из способов оценки сходства или различия между объектами – разбиение их на классы эквивалентности, внутри которых объекты считаются равными. Классы могут быть построены различными способами: с помощью экспертного знания, на основе обучающей выборки, или путём кластеризации базы прецедентов. Возможны также сочетания этих способов. Разбиение на кластеры можно считать частным случаем разбиения на классы, когда не требуется этап предварительного обучения.

В обоих подходах к принятию решений задачу поддержки принятия решений в том или ином виде можно свести к решению задачи классификации, где на

входе системы – набор признаков объекта, а на выходе – принадлежность к классу.

Метод

На очередном шаге проводимых в ИСП РАН исследований осуществлён переход к изучению гибридных систем поддержки принятия решений, в которых методика использования порождающих правил сочетается с методикой использования прецедентов. Такие системы можно рассматривать как очередной шаг вперед по сравнению с простейшей оболочкой, поддерживающей единственную парадигму представления знаний. Существующие прикладные системы с гибридной архитектурой ориентированы в основном на использование правил, а прецеденты используются лишь для обработки исключений. Новизна предлагаемого подхода заключается в том, что в механизме выработки решения результаты вывода по правилам и результаты рассуждений по прецедентам стали взаимно дополнять друг друга. Имея накопленную базу прецедентов, дополняя знания, полученные на основе изучения ранее встречавшихся случаев, действиями в ситуациях, заранее предусмотренных правилами поведения, можно решать различные задачи, связанные, в частности, с дифференциальной диагностикой, выбором лечения, оценками исхода заболеваний.

Большая часть существующих подходов к построению систем, основанных на прецедентах, сосредоточена на отборе прецедентов. В основе подходов к отбору лежит оценка схожести прецедента и текущего случая. Существующие подходы к такой оценке предполагают, что в основе описаний объектов лежит общий набор признаков. В реальных условиях, особенно в медицине, это не выполняется. Необходим метод оценки в условиях, когда объект исследования не полностью описан и оценивается неоднозначно.

Для осуществления оценки в пространстве всех признаков обычно вводится некоторая метрика. В этом же пространстве определяется точка, соответствующая текущему случаю, и на основе выбранной метрики находится ближайшая точка, представляющая прецедент. Однако в некоторых случаях ввести метрику не удаётся. В этих случаях вместо метрики используется так называемая мера близости. При разработке системы упор был сделан на работу в условиях неопределённости, когда объект (текущий случай) не полностью описан и попадает в смешение различных понятий. В медицине такая ситуация возникает, если в условиях дефицита времени и ресурсов выявлен недостаток информации о пациенте. Авторами был предложен алгоритм поиска наиболее подходящего прецедента для неполностью описанных объектов, базирующийся на разбиении базы прецедентов на классы эквивалентности.

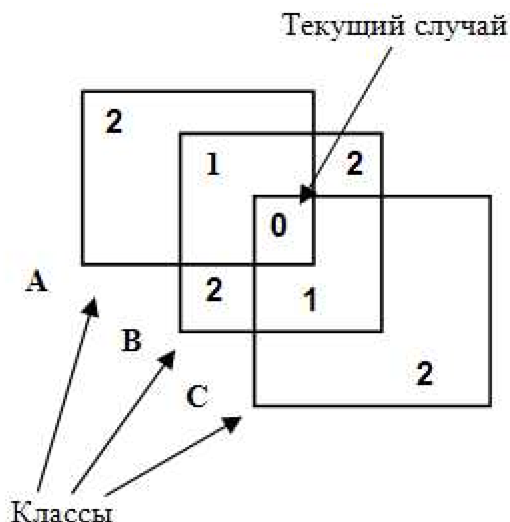


Рис. 1. *Оценка близости в системе.*

Отношения между текущим случаем и классами выявляются в проекциях классов на пространство признаков случая. Недостаточно описанный случай может попасть в проекцию класса, к которому он не принадлежит, только потому, что у него не хватает признака, который дифференцировал бы его от этого класса. Наибольшую информацию об отсутствующих признаках могут дать аналоги – прецеденты, которые в признаковом пространстве случая ведут себя идентично по принадлежности к классам, то есть попадают в ту же область пересечения. Расстояние между текущим случаем и прецедентом определяется как разность количества классов, куда попал текущий случай (эти классы образую так называемый дифференциальный ряд случая), и количества классов из этого числа, в котором находится прецедент (*рис. 1*).

Как указывалось, задачу поддержки принятия решений можно свести к решению задачи классификации, где на выходе – принадлежность объекта классу. Перед реализацией гибридного метода была поставлена основная задача – на основании набора признаков объекта определить его принадлежность классу, возможно доопределив существенные признаки.

В системе, реализованной в ИСП РАН (прецедентный подход) для неполностью описанных объектов искомая принадлежность это – дифференциальный ряд классов возможной принадлежности объекта (*рис. 1*). Задача оценки (распознавания) объекта включает подзадачу – поиск признака, который дифференцировал бы его от классов дифференциального ряда, к которым он не принадлежит.

Медицина – прецедентная наука, но такой механизм явно не применяется при описании случаев в медицинской литературе, хотя понятие “дифференциальная диагностика” и послужила исходным пунктом для

разработки описанного метода. Несмотря на это, примеры в медицине часто описываются с помощью правил “если-то”, а в последнее время также иллюстрируются деревьями решений. На самом деле деревья решений представляют собой частные случаи продукционных правил. Задача оценки при использовании деревьев решений не меняется, а остаётся всё той же: начиная с корня дерева, проверяя узлы как существенные признаки объекта, дойти до одной из листовой вершин, представляющих класс объекта.

Вывод по прецедентам, в том модифицированном виде, как он здесь описан, оптимально подходит для неполностью описанных объектов, так как позволяет сразу оценить классы возможной принадлежности объекта. Остается только дифференцировать эти классы. В свою очередь, вывод по правилам для таких объектов может оказаться неприменим. Отсутствие существенного признака (используемого в узле) может сделать невозможным как вход в набор порождающих правил, так и прохождение указанного узла, делая непригодным весь механизм.

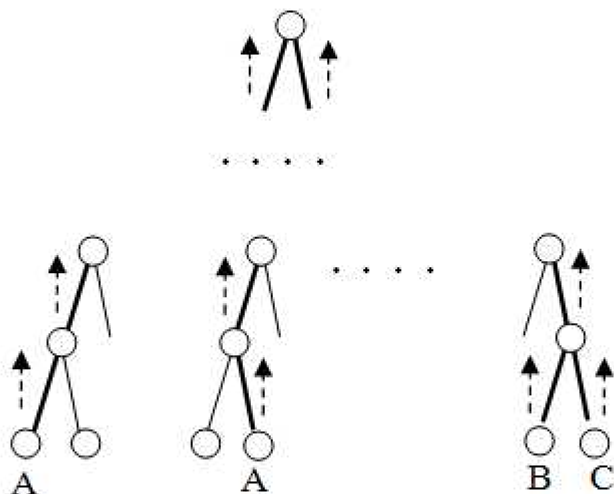


Рис. 2. Обратный логический вывод в деревьях решений.

На текущем этапе разработки системы реализована методика двухэтапной оценки случая, объединяющих оба механизма принятия решений, где вначале используется прецедентный подход с целью получить представление о возможной принадлежности случая тому или иному классу (дифференциальный ряд). На втором этапе, на основе этой информации, но уже по правилам вывода, проводится *обратный логический вывод* от возможного заключения, как гипотезы, – к фактам, подтверждающим эту гипотезу. В качестве заключений берутся классы дифференциального ряда. Если представить этот процесс в дереве решений, то вывод идёт от листовых вершин одного или нескольких деревьев (где листья помечены

предполагаемыми классами) к корням (*рис. 2*). Это снижает затраты на дополнительные исследования недостающих признаков и существенно ограничивает выбор ветвей при движении по цепи правил.

Казалось бы, вывод по прецедентам и по правилам – самодостаточные методы, каждый из которых может решить задачу оценки объекта от признаков к классам. Но в условиях неоднозначной оценки объекта вывод по правилам дополняет рассуждения по прецедентам в механизме выработки решения. В новой версии системы правила вывода могут формироваться как обобщение базы прецедентов, а также на основе экспертного знания, а арсенал предметной области может быть пополнен ассоциативными правилами и деревьями решений.

Заключение

Гибридные системы можно рассматривать как очередной шаг вперед по сравнению с простейшими моделями, поддерживающими только одну парадигму представления знаний. Однако в существующих на данный момент гибридных системах основным инструментом вывода являются порождающие правила, а прецеденты используются лишь для обработки исключений из правил.

В работе представлен подход, объединяющий оба механизма принятия решений, в попытке преодолеть или ослабить недостатки каждого из механизмов, пользуясь знаниями, добываемыми в предметной области методами добычи данных.

Предлагаемый подход позволит интегрировать имеющиеся в предметной области знания в механизм выработки решения таким образом, чтобы результаты вывода по правилам и результаты рассуждений по прецедентам взаимно дополняли друг друга.

Подход пригоден при работе с не полностью описанными объектами в условиях, когда накладываются ограничения по времени и ресурсам. Ранее авторами уже были разработаны алгоритмы отбора прецедентов на основе предложенной меры схожести, в которых была изначально учтена специфика работы с нефиксированным набором показателей, что особенно характерно для медицинских приложений, но этими приложениями не могут и не должны ограничиваться.

Авторам представляется, что такой подход, где методика использования порождающих правил сочетается с методикой использования прецедентов, востребован и актуален.

Литература

[1]. Л. Е. Карпов, В. Н. Юдин, «Методы добычи данных при построении локальной метрики в системах вывода по прецедентам», Препринт Института системного программирования РАН, № 18, 2006, стр 1-42, *http://citforum.ru/consulting/BI/data_mining/*

[2]. Л. Е. Карпов, В. Н. Юдин, «Адаптивное управление по прецедентам, основанное на классификации состояний управляемых объектов», Труды Института системного программирования РАН (ИСП РАН), т. 13, № 2, Институт системного программирования РАН, 2007, стр. 37-57, ISBN 5-89823-026-2. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2007/13/2/isp_2007_13_2_37.pdf, http://www.citforum.ru/consulting/BI/karpov/*

[3]. Л. Е. Карпов, В. Н. Юдин, «Интеграция методов добычи данных и вывода по прецедентам в медицинской диагностике и выборе лечения», Математические методы распознавания образов. Сборник докладов 13-й Всероссийской конференции, октябрь 2007, МАКС Пресс, 2007, стр. 589-591, ISBN 978-5-317-02060-6, *http://www.mmro.ru/files/mmro13.pdf*

[4]. В. Н. Юдин, Л. Е. Карпов, А. В. Ватазин, «Процесс лечения как адаптивное управление человеческим организмом в программной системе "Спутник врача"», Альманах клинической медицины, т. 17, № 1, МОНИКИ, 2008, стр. 262-265, ISBN 978-5-98511-032-6 (Т. XVII, ч. 1), ISBN 5-9900012-1-5, *http://www.isan.troitsk.ru/win/block1.pdf*

[5]. В. Н. Юдин, Л. Е. Карпов, А. В. Ватазин, «Методы интеллектуального анализа данных и вывода по прецедентам в программной системе поддержки врачебных решений», Альманах клинической медицины, т. 17, № 1, МОНИКИ, 2008, стр. 266-269, ISBN 978-5-98511-032-6 (Т. XVII, ч. 1), ISBN 5-9900012-1-5, *http://www.isan.troitsk.ru/win/block1.pdf*

[6]. Л. Е. Карпов, А. Н. Томилин, В. Н. Юдин, «Репликация и валидация в распределенной системе поддержки врачебных решений», Труды Всероссийской научной конференции "Научный сервис в сети Интернет: решение больших задач", МГУ, 2008, стр. 387-392, ISBN 978-5-211-05616-9, *http://agora.guru.ru/abrau2008/pdf/043.pdf*

[7]. Л. Е. Карпов, В. Н. Юдин, А. В. Ватазин, «Виртуальная интеграция и консолидация знаний в распределенной системе поддержки врачебных решений», Научно-практическая конференция ЦФО РФ «Актуальные вопросы гемафереза, хирургической детоксикации и диализа», МОНИКИ, 2009, стр. 36. ISBN 978-5-98511-054-8.

[8]. А. В. Ватазин, Л. Е. Карпов, В. Н. Юдин, «Виртуальная интеграция и консолидация знаний в распределенной системе поддержки врачебных решений», Альманах клинической медицины, т. 20, 2009, стр. 83-86. ISSN 2072-0505.

[9]. А. В. Ватазин, Л. Е. Карпов, В. Н. Юдин, «Многопараметрическое управление сложным объектом в программной системе поддержки врачебных решений», III Евразийский конгресс по медицинской физике и инженерии "Медицинская физика – 2010", 21-25 июня 2010 г., т. 4, МОНИКИ, 2010, стр. 415-417.

[10]. А. В. Ватазин, В. Н. Юдин, Л. Е. Карпов, «Многопараметрическое управление сложным объектом в программной системе поддержки врачебных решений», Ежегодная научно-практическая конференция Центрального Федерального округа

РФ "Актуальные вопросы заместительной почечной терапии, гемафереза и трансплантационной координации", МОНИКИ, 2010, стр. 8. ISBN 978-5-98511-091-3.

[11]. Leonid Karpov, Valery Yudin, «The Case-Based Software System for Physician's Decision Support», Sami Khari, Lenka Lhotska, Nadia Pisanti (eds.), "Information Technology in Bio- and Medical Informatics, ITBAM 2010", Proceedings of the First International Conference, Bilbao, Spain. Lecture Notes in Computer Science Sublibrary: SL 3, Springer Verlag, Berlin, Heidelberg, 2010, pp. 78-85. ISSN 0302-9743.

[12]. L. E. Karpov, V. N. Yudin, A. V. Vatazin, «Multi-Parametric Control of Complex Object in the Program System for Physician's Decision Support», Proceedings of the 12-th International Workshop on Computer Science and Information Technologies (CSIT'2010), Russia, Moscow – St. Petersburg, September 13-19, v. 1, Ufa State Aviation Technical University, 2010, pp. 28-30.

[13]. Л. Е. Карпов, В. Н. Юдин, «Обмен данными в распределённой системе поддержки решений», Труды Института системного программирования, т. 19, Институт системного программирования РАН, 2010, стр. 71-80, ISBN 978-0-543-57630-9, ISBN 978-5-4221-0085-9, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2010/19/isp_19_2010_71.pdf*

[14]. Л. Е. Карпов, В. Н. Юдин, «Многопараметрическое управление на основе прецедентов», Труды Института системного программирования, т. 19, Институт системного программирования РАН, 2010, стр. 81-93, ISBN 978-0-543-57630-9, ISBN 978-5-4221-0085-9, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2010/19/isp_19_2010_81.pdf*

[15]. А. В. Ватазин, Л. Е. Карпов, Ю. Г. Сметанин, В. Н. Юдин, «Программная система поддержки врачебных решений с гибридной архитектурой на основе правил и прецедентов», V Троицкая конференция "Медицинская физика и инновации в медицине (ТКМФ-5)", Сборник материалов, том 2, стр. 425-427. 2012, РАН, Троицкий Научный Центр, ISBN 978-5-89513-272-2.

[16]. Hillary Don, «Decision making in critical care», University of California School of Medicine San Francisco, California, B. C. Decker Inc., The C. V. Mosby company, 1985, есть русский перевод: Х. Дон «Принятие решения в интенсивной терапии», М.: Медицина, 1995, 224 с., ISBN 5-225-00489-X, ISBN 0-941158-35-7.

Hybrid approach to building decision support system

V. N. Yudin, L. E. Karpov
ISP RAS, Moscow, Russia
This work is supported by Russian Fund for Basic Research
№ 12-01-00780, № 12-07-00214

Abstract. Decision support systems where the results of deduction on rules are supplemented with Case-based Reasoning results are another step forward in comparison with models which are supporting only single knowledge paradigm. The main deduction instruments in up-to-date hybrid systems are producing rules. The precedents are used only for exception processing. The approach described here which was designed and developed for the second release of system in the Institute for System Programming of Russian Academy of Sciences. On the basis of this research system the medical decision support system "Doctor's Partner" is being developed. Now the rule and precedent reasoning are complementing each other while the conditions of ambiguity of not fully described case still exist. The absence of significant feature for one of the rules often prevents from using the rule-oriented method at all. During the current project stage the method of two-phase case estimation is used. The first phase consists of using the Case-based Reasoning method in order to get some knowledge about the possible case class (so called differential set). The next phase will use this information to produce the reverse logical inference treating the possible conclusion as a hypothesis and moving to the facts that are able to confirm it. The approach shortly described here may decrease the cost of additional rule investigation and is significantly limiting the branch choosing while moving along the rules chain.

Keywords: decision support system, Case-Based Reasoning, Data Mining, Rule-Based deduction, case base, measure of closeness, equivalence classes, feature space, incompleteness of description, differential set of states, reverse logical inference.

References

[1]. L. E. Karpov, V. N. Yudin. Metody dobychi dannykh pri postroenii lokal'noj metriki v sistemakh vyvoda po pretsedentam [Data Mining methods in building local metrics for Case-Based Reasoning systems], Preprint ISP RAN [Preprint of ISP RAS], no. 18, 2006, pp. 1-42, http://citforum.ru/consulting/BI/data_mining/ (in Russian).

[2]. L. E. Karpov, V. N. Yudin. Adaptivnoe upravlenie po pretsedentam, osnovannoe na klassifikatsii sostoyanij upravlyaemykh ob"ektov [Case-Based Reasoning adaptive control with classification of states of objects under control], Trudy ISP RAN [The Proceedings of ISP RAS], vol. 13, no. 2, 2007, pp. 37-57, ISBN 5-89823-026-2. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), http://www.ispras.ru/ru/proceedings/docs/2007/13/2/isp_2007_13_2_37.pdf, http://www.citforum.ru/consulting/BI/karpov/ (in Russian).

[3]. L. E. Karpov, V. N. Yudin. Integratsiya metodov dobychi dannykh i vyvoda po pretsedentam v meditsinskoj diagnostike i vybore lecheniya [Integration of data Mining and Case-Based Reasoning methods in medical diagnostics and treatment choosing], Sbornik dokladov 13-j Vserossijskoj konferentsii Matematicheskie metody

raspoznavaniya obrazov [Proc. of 13-th All-Russian conference Math. methods of pattern recognition], October 2007, MAKS Press, 2007, pp. 589-591, ISBN 978-5-317-02060-6, http://www.mmro.ru/files/mmro13.pdf (in Russian).

[4]. V. N. Yudin, L. E. Karpov, A. V. Vatazin. Protsess lecheniya kak adaptivnoe upravlenie chelovecheskim organizmom v programmnoj sisteme "Sputnik vracha" [Process of patient treatment as an adaptive control of human being organism in software system "Doctor's Partner"], Al'manakh klinicheskoj meditsiny [Almanac of Clinical Medicine], vol. 17, no. 1, МОНИКИ [Moscow Regional Scientific Research Clinical Institute], 2008, pp. 262-265, ISSN 2072-0505, ISBN 978-5-98511-032-6, ISBN 5-9900012-1-5, http://www.isan.troitsk.ru/win/block1.pdf (in Russian).

[5]. V. N. Yudin, L. E. Karpov, A. V. Vatazin. Metody intellektual'nogo analiza dannykh i vyvoda po pretsedentam v programmnoj sisteme podderzhki vrachebnykh reshenij [Application of Data Mining and Case-Based Reasoning in software system for physician's decision support], Al'manakh klinicheskoj meditsiny [Almanac of Clinical Medicine], vol. 17, no. 1, MONIKI [Moscow Regional Scientific Research Clinical Institute], 2008, pp. 266-269, ISSN 2072-0505, ISBN 978-5-98511-032-6, ISBN 5-9900012-1-5, http://www.isan.troitsk.ru/win/block1.pdf (in Russian).

[6]. L. E. Karpov, A. N. Tomilin, V. N. Yudin. Replikatsiya i validatsiya v raspredelennoj sisteme podderzhki vrachebnykh reshenij [Data replication and validation in distributed software system for physician's decision support], Trudy Vserossijskoj nauchnoj konferentsii "Nauchnyj servis v seti Internet: reshenie bol'shikh zadach" [Proc. All-Russian scientific conference “Scientific Service in Internet: solving of huge problems”], MGU [Moscow State University, 2008, pp. 387-392, ISBN 978-5-211-05616-9, http://agora.guru.ru/abrau2008/pdf/043.pdf (in Russian).

[7]. L. E. Karpov, A. V. Vatazin, V. N. Yudin. Virtual'naya integratsiya i konsolidatsiya znanij v raspredelennoj sisteme podderzhki vrachebnykh reshenij [Virtual knowledge integration and consolidation in software system for physician's decision support], Trudy Nauchno-prakticheskaya konferentsiya TSFO RF «Aktual'nye voprosy gemafereza, khirurgicheskoj detoksikatsii i dializa [Proc. of research and practical conference ‘Actual problems of hemapheresis, surgery detoxication and dialysis’], MONIKI [Moscow Regional Scientific Research Clinical Institute], 2009, pp. 36. ISBN 978-5-98511-054-8 (in Russian).

[8]. A. V. Vatazin, L. E. Karpov, V. N. Yudin. Virtual'naya integratsiya i konsolidatsiya znanij v raspredelennoj sisteme podderzhki vrachebnykh reshenij [Virtual knowledge integration and consolidation in software system for physician's decision support], Al'manakh klinicheskoj meditsiny [Almanac of Clinical Medicine], vol. 20, 2009, pp. 83-86. ISSN 2072-0505 (in Russian).

[9]. A. V. Vatazin, L. E. Karpov, V. N. Yudin. Mnogoparametricheskoe upravlenie slozhnym ob"ektom v programmnoj sisteme podderzhki vrachebnykh reshenij [Multiparametric object control in software system for physician's decision support], III Evrazijskij kongress po meditsinskoj fizike i inzhenerii "Meditsinskaya fizika – 2010" [Third Euro-Asia congress for medical physics], 21-25 of June 2010, vol. 4, MONIKI [Moscow Regional Scientific Research Clinical Institute], 2010, pp. 415-417 (in Russian).

[10]. A. V. Vatazin, L. E. Karpov, V. N. Yudin. Mnogoparametricheskoe upravlenie slozhnym ob"ektom v programmnoj sisteme podderzhki vrachebnykh reshenij [Multiparametric object control in software system for physician's decision support], Ezhegodnaya nauchno-prakticheskaya konferentsiya TSentral'nogo Federal'nogo okruga RF "Aktual'nye voprosy zamestitel'noj pochechnoj terapii, gemafereza i

transplantatsionnoj koordinatsii" [Proc. of Annual research and practical conference 'Actual problems of replacement therapy, hemapheresis, and transplantation coordination'], MONIKI [Moscow Regional Scientific Research Clinical Institute], 2010, стр. 8. ISBN 978-5-98511-091-3. (in Russian).

[11]. Leonid Karpov, Valery Yudin. The Case-Based Software System for Physician's Decision Support. Sami Khari, Lenka Lhotska, Nadia Pisanti (eds.), "Information Technology in Bio- and Medical Informatics, ITBAM 2010", Proceedings of the First International Conference, Bilbao, Spain. Lecture Notes in Computer Science Sublibrary: SL 3, Springer Verlag, Berlin, Heidelberg, 2010, pp. 78-85. ISSN 0302-9743.

[12]. L. E. Karpov, V. N. Yudin, A. V. Vatazin. Multi-Parametric Control of Complex Object in the Program System for Physician's Decision Support, Proceedings of the 12-th International Workshop on Computer Science and Information Technologies (CSIT'2010), Russia, Moscow – St. Petersburg, September 13-19, v. 1, Ufa State Aviation Technical University, 2010, pp. 28-30.

[13]. L. E. Karpov, V. N. Yudin. Obmen dannymi v raspredelyonnoj sisteme podderzhki reshenij [Data exchange in distributed software system for decision support], Trudy ISP RAN [The Proceedings of ISP RAS], vol. 19, 2010, pp. 71-80, ISBN 978-0-543-57630-9, ISBN 978-5-4221-0085-9, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), http://www.ispras.ru/ru/proceedings/docs/2010/19/isp_19_2010_71.pdf (in Russian).

[14]. L. E. Karpov, V. N. Yudin. Case-based multi-parametric object control, Trudy ISP RAN [The Proceedings of ISP RAS], vol. 19, 2010, pp. 81-93, ISBN 978-0-543-57630-9, ISBN 978-5-4221-0085-9, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), http://www.ispras.ru/ru/proceedings/docs/2010/19/isp_19_2010_81.pdf (in Russian).

[15]. A. V. Vatazin, L. E. Karpov, Y. G. Smetanin, V. N. Yudin. Programmnaya sistema podderzhki vrachebnykh reshenij s gibridnoj arkhitekturoj na osnove pravil i pretsedentov [Software system for physician's decision support with architecture based on rules and cases], V Troitskaya konferentsiya "Meditsinskaya fizika i innovatsii v meditsine (TKMF-5)", Sbornik materialov [Proc. of Fifth conference 'Medical physics and innovations in medicine'], vol. 2, pp. 425-427. 2012, RAS, Troitsk Scienfic Centre, ISBN 978-5-89513-272-2 (in Russian).

[16]. Hillary Don. Decision making in critical care. University of California School of Medicine San Francisco, California, B. C. Decker Inc., The C. V. Mosby company, 1985.

Вероятностный анализ нового алгоритма упаковки прямоугольников в полосу

М.А. Трушников, ctgy@yandex.ru

Аннотация. В 1993 году Коффман и Шор предложили онлайновый алгоритм упаковки прямоугольников в полосу с оценкой $O(N^{2/3})$ для математического ожидания незаполненной площади полосы в стандартной вероятностной модели. С тех пор вопрос о возможности улучшения этой оценки в классе онлайновых алгоритмов оставался открытым. В данной работе проанализирован принципиально новый онлайновый алгоритм упаковки, предложенный ранее автором. Для него доказана оценка $O(N^{1/2}(\log N)^{3/2})$ для математического ожидания незаполненной площади полосы.

Ключевые слова: онлайновый алгоритм упаковки, вероятностный анализ качества упаковки.

Введение

Задача упаковки в полосу (в англоязычной литературе Strip packing problem) состоит в размещении множества открытых прямоугольников внутри полубесконечной вертикальной полосы единичной ширины, при этом стороны прямоугольников должны быть параллельны сторонам полосы (вращения запрещены) и прямоугольники не должны пересекаться. Нужно минимизировать «высоту упаковки» --- расстояние от основания полосы до верхней грани верхнего прямоугольника в упаковке [1,2].

Задача имеет следующую естественную интерпретацию. Каждый прямоугольник --- вычислительная задача, ширина прямоугольника соответствует количеству процессоров необходимых для вычисления задачи, высота --- времени. Эффективное размещение прямоугольников внутри полосы требуется также в задачах разработки СБИС и раскройки материалов. Частным случаем упаковки в полосу при равенстве высот всех прямоугольников является NP-трудная задача упаковки в

контейнеры [3]. Поэтому для общей задачи упаковки в полосу интерес представляют приближенные полиномиальные алгоритмы.

Особый интерес представляют «онлайновые» алгоритмы, размещающие прямоугольники в полосе по мере их поступления, без знания параметров всех последующих прямоугольников. При анализе **в среднем** минимизируемой функцией является математическое ожидание незаполненной прямоугольниками площади полосы от основания полосы до верхней грани самого верхнего прямоугольника в упаковке. При этом в стандартной вероятностной модели ширины и высоты всех прямоугольников являются независимыми в совокупности равномерно распределенными на $(0,1]$ случайными величинами [4].

Известно, что математическое ожидание незаполненной площади полосы у оптимальной упаковки есть $O(N^{1/2})$, где N --- число прямоугольников. В 1993 году Коффман и Шор предложили «офлайновый» (информация о всех прямоугольниках известна заранее) алгоритм [4], для которого по порядку достигается оценка $O(N^{1/2})$. В той же работе они предложили онлайновый алгоритм упаковки с оценкой $O(N^{2/3})$ для математического ожидания незаполненной площади полосы.

В 2010 году [5] был предложен новый онлайновый алгоритм с той же оценкой $O(N^{2/3})$, но не требующий знания числа прямоугольников заранее. Тем не менее вопрос о возможности улучшения оценки $O(N^{2/3})$ в классе онлайновых алгоритмов оставался открытым. Известно было, в частности, что в классе шельфовых алгоритмов [6], [7] эту оценку улучшить нельзя.

В данной работе дан положительный ответ на этот вопрос. Для принципиально нового онлайнового алгоритма упаковки из [8] доказана оценка $O(N^{1/2}(\log N)^{3/2})$ для математического ожидания незаполненной площади полосы.

Постановка задачи

1 *Вход:* N --- число прямоугольников; $w_i, h_i, i = 1,\ldots,N$ --- ширины и высоты прямоугольников, являющиеся значениями независимых в совокупности равномерно распределенных на $(0,1]$ случайных величин;

Выход: $x_i, y_i, i = 1,\ldots,N$ -- координаты центров прямоугольников, удовлетворяющие условию: прямоугольники без вращений и пересечений размещены внутри полубесконечной полосы единичной

ширины. Основание полосы совпадает с отрезком $[(0,0),(1,0)]$ в R^2, а боковые стороны полосы параллельны оси y.

Высотой упаковки назовем величину

$$H = max_i\left(y_i + \frac{h_i}{2}\right).$$

Требуется минимизировать величину

$$S = H - \sum_{i=1}^{N} w_i * h_i,$$

равную незаполненной площади полосы.

Алгоритм

В [8] предложен новый онлайновый алгоритм для рассматриваемой задачи, который мы будем анализировать в данной работе. Нпомним некоторые особенности алгоритма.

Рассмотрим следующие величины:

$$d = \left\lfloor \frac{N/4}{\sqrt{N}} \right\rfloor,\ U = \frac{N/4}{d} = \sqrt{N} + O(1), \delta = \frac{1}{d}$$

В основании полосы выделается $d+1$ горизонтальная область, каждая высоты U, причем i-я область имеет ширину $i\delta$. Таким образом получаем две одинаковые пирамиды (одна из них перевернутая). Каждый четный прямоугольник будем упаковывать в одну пирамиду, каждый нечетный --- в другую.

Прямоугольники, из которых состоит пирамида будем называть **контейнерами**. Пронумеруем контейнеры внутри пирамиды числами от 1 до d так, что i-й имеет ширину $i\delta$. Прямоугольники внутри контейнеров будут упаковываться просто друг над другом: первый упаковываемый прямоугольник кладется на дно контейнера, следующий --- поверх первого и так далее.

Пусть некоторое количество прямоугольников упаковано в пирамиду и следующим для упаковки в данную пирамиду приходит прямоугольник ширины w.

- Найдем такое i, что $(i-1)\delta < w \le i\delta$. Будем говорить, что этот прямоугольник **назначен** в i-й контейнер (тем не менее он не обязательно будет упакован именно в i-й контейнер).
- Далее ищем минимальное такое j, что $i \le j \le d$ и в j-ом контейнере достаточно места, чтобы поместить туда данный прямоугольник.

- Если такое j существует --- помещаем данный прямоугольник в j-й контейнер.
- Если нет --- просто кладем прямоугольник сверху текущей упаковки. Такие прямоугольники, которым не нашлось места ни в одном из контейнеров, в которые они помещаются по ширине будем называть **выпавшими**.

Анализ алгоритма

Корректность и онлайновость алгоритма очевидны. Оценим математическое ожидание незаполненной площади полосы.

Теорема2. *Математическое ожидание незаполненной площади полосы упаковки, полученной алгоритмом из [8] есть* $O(N^{1/2}(\log N)^{3/2})$.

План доказательства теоремы. Обозначим за S суммарную площадь N прямоугольников. Ясно, что $\mathsf{E}S = N/4$. Высота выделенной части полосы есть

$$(d+1)U = N/4\left(\frac{d+1}{d}\right) = N/4 + \frac{N}{4\left\lfloor \frac{N/4}{\sqrt{N}} \right\rfloor} = N/4 + O(N^{1/2}).$$

Будем рассматривать только одну из двух пирамид и соответственно только $\lfloor N/2 \rfloor$ прямоугольников упакованных в эту пирамиду. Пронумеруем эти $\lfloor N/2 \rfloor$ прямоугольников числами от 1 до $\lfloor N/2 \rfloor$ в соответствии с порядком поступления прямоугольников на вход алгоритму.

Обозначим за $\mathsf{M}\{n_1, n_2\}$ математическое ожидание числа **выпавших** (смотри определение в описании алгоритма) прямоугольников при упаковке прямоугольников с номерами из диапазона $[n_1, n_2]$ в данную пирамиду.

Таким образом, для доказательства теоремы требуется доказать, что

$$\mathsf{M}\{1, \lfloor N/2 \rfloor\} = O(N^{1/2}(\log N)^{3/2}) .$$

Определим 2 целых числа k_0 и k_1 :

$$k_0 = \lfloor N/2 \rfloor - \lfloor N^{3/4}\sqrt{\log N} \rfloor, k_1 = \lfloor N/2 \rfloor - \lfloor N^{1/2} \rfloor.$$

$$\mathsf{M}\{1, \lfloor N/2 \rfloor\} = \mathsf{M}\{1, k_0\} + \mathsf{M}\{k_0+1, k_1\} + \mathsf{M}\{k_1+1, \lfloor N/2 \rfloor\}$$

Далее будут доказаны несколько лемм, из которых следует утверждение теоремы.

Лемма 1.

$$\mathsf{M}\{k_1+1, \lfloor N/2 \rfloor\} = O(N^{1/2}).$$

Очевидно, так как $k_1 = \lfloor N/2 \rfloor - \lfloor N^{1/2} \rfloor$ и следовательно всего прямоугольников в этом диапазоне $O(N^{1/2})$.

Для любого прямоугольника вероятность быть назначенным (смотри определение в описании алгоритма) в каждый конкретный контейнер есть $1/d$, так как всего d контейнеров. Пусть в пирамиду было упаковано k прямоугольников. Зафиксируем любой контейнер пирамиды. Через $X_i, 1 \le i \le k$ обозначим случайную величину, равную высоте i-го прямоугольника в случае, если i-й прямоугольник назначается в данный контейнер, и 0 иначе. Можно записать $X_i = \xi_i \eta_i$, где ξ_i --- случайная величина, равная 1 с вероятностью p и 0 и вероятностью $1-p$, а η_i --- равномерно распределенная на отрезке $(0,1]$ случайная величина. Введем $X = X_1 + X_2 + \ldots + X_k$ --- случайную величину, равную суммарной высоте прямоугольников, назначенных в данный контейнер, где k --- количество упакованных прямоугольников. Далее нам потребуется следующий вариант неравенства больших уклонений.

Лемма 2.

Пусть случайная величина $X = X_1 + X_2 + \ldots + X_k$, где $X_i = \xi_i \eta_i$, ξ_i принимает значение 1 с вероятностью p и 0 с вероятностью $1-p$, η_i --- равномерно распределенная на отрезке $(0,1]$ случайная величина, причем все случайные величины $\xi_i, \eta_i, i = 1, \ldots, k$ независимы в совокупности. Тогда для любого α из интервала $(0,1)$ выполняется неравенство

$$\mathsf{P}\{X > (1+\alpha)\mathsf{E}X\} \le e^{-\frac{5}{9}\alpha^2 \mathsf{E}X}.$$

Лемма 3.

$$\mathsf{M}\{1, k_0\} \to 0, \quad N \to \infty$$

Доказательство. Оценим вероятность переполнения любого контейнера с помощью леммы 2. Для $k = k_0 = \lfloor N/2 \rfloor - \lfloor N^{3/4}\sqrt{\log N} \rfloor$

$$\mathrm{E}(X)=\frac{1}{2}\left(\frac{k_0}{d}\right)=\frac{1}{2}\left(\frac{\lfloor N/2\rfloor-\lfloor N^{3/4}\sqrt{\log N}\rfloor}{\frac{\sqrt{N}}{4}+O(1)}\right)=\sqrt{N}-2N^{1/4}\sqrt{\log N}+O(1).$$

Возьмем α , удовлетворяющее равенству $U-1=(1+\alpha)\mathrm{E}X$.

$$\alpha=\frac{U-1-\mathrm{E}X}{\mathrm{E}X}=\frac{2N^{1/4}\sqrt{\log N}}{\sqrt{N}-2N^{1/4}\sqrt{\log N}}+O(1)$$

Следовательно, по лемме 2

$$\mathrm{P}\{X>U-1\}\le exp\left(-(5/9)\frac{(2N^{1/4}\sqrt{\log N}+O(1))^2}{N^{1/2}-2N^{1/4}\sqrt{\log N}+O(1)}\right)$$

$$\le exp\left(-(5/9)\frac{4N^{1/2}\log N}{N^{1/2}-2N^{1/4}\sqrt{\log N}+O(1)}\right).$$

Для достаточно больших N

$$\mathrm{P}\{X>U-1\}<\frac{1}{N^{2.1}}.$$

Обозначим событие, соответствующее переполнению хотя бы одного любого контейнера за A, через A_i --- событие переполнения i-го контейнера,

$$A=\bigcup_{i=1}^{d}A_i$$

$$\mathrm{P}(A)\le\sum_{i=1}^{d}\mathrm{P}(A_i)<d\frac{1}{N^2}<\frac{1}{N^{1.1}}$$

Следовательно

$$\mathrm{M}\{1,k_0\}\to 0,\ \ N\to\infty.$$

Лемма 4.

Пусть алгоритмом было упаковано $\lfloor N/2\rfloor-\lfloor N^{1/2+\beta}\rfloor$ прямоугольников в одну пирамиду и $0<\beta<1/4$. Зафиксируем любое γ такое, что

$$1/2-2\beta+\frac{\ln(5\ln N)}{\ln N}\le\gamma<1/2.$$

Обозначим через C --- событие, заключающееся в том, что хотя бы один из $\lceil N^{\gamma}\rceil$ ***нижних*** *контейнеров пирамиды заполнен не более чем на $U-1$. Тогда для достаточно больших N*

$$P\{C\} \geq 1 - \frac{1}{N^{1.1}}$$

Доказательство. Рассмотрим **любые** $\lceil N^{\gamma} \rceil$ контейнеров пирамиды. Для $k = \lfloor N/2 \rfloor - \lfloor N^{1/2+\beta} \rfloor$ рассмотрим

$$X = X_1 + X_2 + \ldots + X_k .$$

Где X_i --- высота i-го прямоугольника в случае, если он назначен в один из $\lceil N^{\gamma} \rceil$ рассматриваемых контейнеров пирамиды и 0 иначе.

$$\mathsf{E}X = \frac{1}{2} \cdot \frac{\left(\lfloor N/2 \rfloor - \lfloor N^{1/2+\beta} \rfloor\right)\lceil N^{\gamma} \rceil}{d}$$

$$\mathsf{E}X \leq \frac{2\left(N/2 - N^{1/2+\beta} + 1\right)N^{\gamma}}{N^{1/2} - 1}$$

Докажем сначала для произвольного множества из $\lceil N^{\gamma} \rceil$ контейнеров пирамиды, что

$$P\left\{X > (U-1)\lceil N^{\gamma} \rceil\right\} < \frac{1}{N^{2.1}}.$$

Найдем α, которое удовлетворяет равенству

$$(U-1)\lceil N^{\gamma} \rceil = (1+\alpha)\mathrm{E}X.$$

$$\alpha = \frac{(U-1)\lceil N^{\gamma} \rceil - \mathrm{E}X}{\mathrm{E}X} = \frac{U - 1 - \dfrac{\lfloor N/2 \rfloor - \lfloor N^{1/2+\beta} \rfloor}{2d}}{\dfrac{\lfloor N/2 \rfloor - \lfloor N^{1/2+\beta} \rfloor}{2d}} = \frac{N/2 - 2d - \lfloor N/2 \rfloor + \lfloor N^{1/2+\beta} \rfloor}{\lfloor N/2 \rfloor - \lfloor N^{1/2+\beta} \rfloor}$$

$$\alpha \leq \frac{N^{1/2+\beta} - \dfrac{1}{2}N^{1/2} + 1}{\dfrac{N}{2} - N^{1/2+\beta} - 1}$$

Оценим вероятность отклонения случайной величины X от $\mathsf{E}X$ на $\alpha\mathsf{E}X$ с помощью леммы 2.

$$P\left\{X > (U-1)\lceil N^{\gamma} \rceil\right\} = P\left\{X > (1+\alpha)\mathrm{E}X\right\}$$

$$P\left\{X > (1+\alpha)\mathrm{E}X\right\} < e^{-(5/9)\alpha^2 \mathrm{E}X} \le$$

$$\le exp\left(-\frac{5}{9}\cdot\frac{\left(N^{1/2+\beta}-\frac{1}{2}N^{1/2}+1\right)^2\left(2\left(N/2-N^{1/2+\beta}+1\right)N^{\gamma}\right)}{\left(\frac{N}{2}-N^{1/2+\beta}-1\right)^2\left(N^{1/2}-1\right)}\right)$$

Для достаточно больших N

$$P\left\{X > (1+\alpha)\mathrm{E}X\right\} \le exp\left(-\frac{4}{9}\cdot\frac{N^{2+2\beta+\gamma}}{N^{2+1/2}}\right)$$

По условию

$$\gamma \ge 1/2 - 2\beta + \frac{\ln\left(5\ln N\right)}{\ln N}.$$

Следовательно для достаточно больших N

$$P\left\{X > (U-1)\lceil N^{\gamma}\rceil\right\} \le exp\left(-\frac{4}{9} * N^{\frac{\ln(5\ln N)}{\ln N}}\right) \le \frac{1}{N^{2.1}}.$$

Оценим теперь вероятность того, что хотя бы в одну последовательность $\lceil N^{\gamma}\rceil$ **подряд идущих** контейнеров будет назначено множество прямоугольников с суммарной высотой превосходящей $(U-1)\lceil N^{\gamma}\rceil$ (обозначим это событие за B). Последовательность подряд идущих контейнеров с номерами с i-го по $(i+\lceil N^{\gamma}\rceil-1)$-й назовем i-ой группой контейнеров (всего имеется $d-\lceil N^{\gamma}\rceil+1$ групп). За B_i обозначим событие назначения прямоугольников с суммарной высотой, превосходящей $(U-1)\lceil N^{\gamma}\rceil$ в контейнеры i-ой группы,

$$B = \bigcup_{i=1}^{d-\lceil N^{\gamma}\rceil+1} B_i.$$

$$\mathsf{P}(B) \le \sum_{i=1}^{d-\lceil N^{\gamma}\rceil+1} \mathsf{P}(B_i) < \left(d-\lceil N^{\gamma}\rceil+1\right)\frac{1}{N^2} < \frac{1}{N^{1.1}}$$

Покажем теперь, что если событие B не произошло, то в процессе упаковки прямоугольников в пирамиду по предложенному алгоритму хотя бы в одном из нижних $\lceil N^{\gamma}\rceil$ контейнеров достаточно места для упаковки прямоугольника высоты 1 (одновременно нижние $\lceil N^{\gamma}\rceil$ контейнеров не могут быть заполнены на $U-1$).

Пусть это не так и каждый из $\lceil N^{\gamma} \rceil$ нижних контейнеров заполнен более, чем на $U-1$. Вспомним, что j-й контейнер имеет ширину $j\alpha$.

Найдем минимальный номер контейнера i такой, что $i \le d - \lceil N^{\gamma} \rceil$ и все контейнеры с i-го по d-й заполнены более, чем на $U-1$. Тогда утверждается, что в i-ую группу контейнеров (с номерами от i до $i + \lceil N^{\gamma} \rceil - 1$) были назначены прямоугольники с суммарной высотой превосходящей $(U-1)\lceil N^{\gamma} \rceil$. Действительно, в i-ую группу контейнеров по определению индекса i упаковано множество прямоугольников с суммарной высотой превосходящей $(U-1)\lceil N^{\gamma} \rceil$, а так как либо $i=1$, либо в $(i-1)$-й контейнер можно упаковать прямоугольник высоты 1, то все контейнеры упакованные в i-ую группу были в нее и назначены. Получено противоречие.

Лемма 5.

$$\mathsf{M}\{k_0+1, k_1\} = O\left(N^{\frac{1}{2}} (\log N)^{3/2}\right)$$

Доказательство. Разделим отрезок $[0,1/4]$ на

$$n = \left\lfloor \frac{\ln N}{6\ln\left(5\ln N\right)} \right\rfloor$$

одинаковых частей. Введем обозначение M_i,

$$\mathrm{M}_i = \mathrm{M}\left\{ \lfloor N/2 \rfloor - \lfloor N^{\frac{1}{2}+\frac{1}{4n}i} \rfloor, \lfloor N/2 \rfloor - \lfloor N^{\frac{1}{2}+\frac{1}{4n}(i-1)} \rfloor \right\}.$$

$$\mathrm{M}\{k_0+1, k_1\} = \sum_{i=1}^{n} \mathrm{M}_i = \sum_{i=1}^{n} \mathrm{M}\left\{ \lfloor N/2 \rfloor - \lfloor N^{\frac{1}{2}+\frac{1}{4n}i} \rfloor, \lfloor N/2 \rfloor - \lfloor N^{\frac{1}{2}+\frac{1}{4n}(i-1)} \rfloor \right\}$$

Оценим M_i для $2 \le i \le n$.

Применим лемму 4 для параметров

$$\beta = \frac{i-1}{4n},$$

$$\gamma = \frac{1}{2} - \frac{i-1}{3n}.$$

Очевидно, что

$$\gamma \ge \frac{1}{2} - 2\beta + \frac{\ln\left(5\ln N\right)}{\ln N}$$

и алгоритмом было упаковано $\lfloor \frac{N}{2} \rfloor - \lfloor N^{1/2+\beta} \rfloor$ прямоугольников (условия леммы 4 выполняются). Следовательно нижние $\lceil N^{\gamma} \rceil$ контейнеров пирамиды к моменту упаковки $\lfloor \frac{N}{2} \rfloor - \lfloor N^{1/2+\beta} \rfloor$ прямоугольников не могут быть все заняты более чем на $U-1$ и следовательно прямоугольник может выпасть только если будет назначен в один из нижних $\lceil N^{\gamma} \rceil$ контейнеров пирамиды. Всего в пирамиде d контейнеров, то есть вероятность того, что каждый прямоугольник из рассматриваемого диапазона может выпасть не превосходит

$$\frac{\lceil N^{\gamma} \rceil}{d}.$$

Всего прямоугольников в диапазоне

$$\left(\lfloor N/2 \rfloor - \lfloor N^{\frac{1}{2}+\frac{1}{4n}i} \rfloor, \lfloor N/2 \rfloor - \lfloor N^{\frac{1}{2}+\frac{1}{4n}(i-1)} \rfloor \right)$$

не более $N^{\frac{1}{2}+\frac{i}{4n}}$. Следовательно

$$\mathrm{M}_i = \mathrm{M}\left\{ \lfloor N/2 \rfloor - \lfloor N^{\frac{1}{2}+\frac{1}{4n}i} \rfloor, \lfloor N/2 \rfloor - \lfloor N^{\frac{1}{2}+\frac{1}{4n}(i-1)} \rfloor \right\} =$$

$$= O\left(\frac{N^{\frac{1}{2}+\frac{i}{4n}} \cdot N^{\frac{1}{2}-\frac{i-1}{3n}}}{N^{1/2}} \right) = O\left(N^{\frac{1}{2}+\frac{-i+4}{12n}} \right).$$

Для $i \geq 5$ $\quad \mathrm{M}_i = O(N^{\frac{1}{2}-\frac{1}{12n}})$.

$$\sum_{i=5}^{n} \mathrm{M}_i = O\left(nN^{\frac{1}{2}-\frac{1}{12n}} \right) = O\left(\log N \frac{N^{\frac{1}{2}}}{\log\log N} \right) = O(N^{\frac{1}{2}} \log N).$$

Кроме того

$$\sum_{i=2}^{4} \mathrm{M}_i = O(3 \cdot M_2) = O\left(N^{\frac{1}{2}+\frac{1}{6n}} \right) = O\left(N^{\frac{1}{2}+\frac{\ln 5 \ln N}{\ln N}} \right) = O\left(N^{\frac{1}{2}} \log N \right).$$

Для $i = 1$ количество прямоугольников в отрезке

$$\left(\lfloor N/2 \rfloor - \lfloor N^{\frac{1}{2}+\frac{1}{4n}} \rfloor, \lfloor N/2 \rfloor - \lfloor N^{\frac{1}{2}} \rfloor \right)$$

есть

$$O\left(N^{\frac{1}{2}+\frac{1}{4n}} \right) = O\left(N^{\frac{1}{2}} (\log N)^{3/2} \right)$$

Итого

$$\mathrm{M}\left\{k_0+1,k_1\right\}=\sum_{i=1}^{n}\mathrm{M}_i=O\left(N^{\frac{1}{2}}(\log N)^{3/2}\right)$$

Лемма 5 доказана а вместе с ней и теорема.

Список литературы

[1]. Baker B. S., Coffman E. J., Rivest R. L. Orthogonal packings in two dimensions. SIAM J. Computing. 1980. V. 9. 4. P. 846-855.

[2]. Baker B. S., Schwartz J. S. Shelf algorithms for two dimensional packing problems. SIAM J. Computing. 1983. V. 6. 2. P. 508-525.

[3]. Shor P. W. The average-case analysis of some on-line algorithms for bin packing. Combinatorica. 1986. V. 6. 2. P. 179-200.

[4]. Coffman E. G., Jr, Shor P. W. Packing in two dimensions: Asymptotic average-case analysis of algorithms. Algorithmica. 1993. V. 9. 3. P. 253-277.

[5]. Кузюрин Н. Н., Поспелов А. И. Вероятностный анализ нового класса алгоритмов упаковки прямоугольников в полосу. ЖВМиМФ. 2011. Т. 51, N 10, с. 1931-1936.

[6]. Кузюрин Н. Н., Поспелов А. И. Вероятностный анализ шельфовых алгоритмов упаковки прямоугольников в полосу. Дискретная математика. 2006. Т. 18. 1. С. 76-90.

[7]. Csirik J., Woegenger G.J. Shelf algorithm for on-line strip packing, Inf. Process. Letters, 1997, v. 63, N 4, P. 171-175.

[8]. Трушников М.А., Об одной задаче Коффмана-Шора, связанной с упаковкой прямоугольников в полосу, Труды ИСП РАН, 2012 г., т. 22, с. 456-462.

Probabilistic analysis of a new strip packing algorithm

Mikhail Trushnikov
(Microsoft Corporation, Redmond, Washington, 98052, USA)

Abstract. In 1993 Coffman and Shor proposed on-line strip packing algorithm with expected unpacked area (waste area) of order O(n^{2/3}) in a standard probabilistic model, where n is the number of rectangles. In the standard probabilistic model the width and height of each rectangle are independent random variables with uniform distribution in [0,1]. It is well-known that optimal packing has expected waste of order O(n^{1/2}) and there exists off-line polynomial algorithm that provides this bound. During more than 10 years the question concerning the possibility to obtain similar upper bound in the class of on-line packing algorithms was open. It was also known that in the class of popular so-called shelf algorithms the bound O(n^{2/3}) cannot be improved. In this paper we give positive answer to this question. We analyze new packing algorithm proposed recently by the author and prove new upper bound of unpacked area of order O(n^{1/2} (log n)^{3/2}) which is almost optimal up to logarithmic factor. The only restriction is the following: we must know the number n of rectangles in advance (exactly as in algorithm of Coffman and Shor). In a popular terminology concerning on-line algorithms this means that our algorithm is closed-end on-line algorithm.

Keywords: on-line strip packing algorithm, probabilistic analysis of packing quality

References

[1]. Baker B. S., Coffman E. J., Rivest R. L. Orthogonal packings in two dimensions. SIAM J. Computing. 1980. V. 9. N 4. P. 846-855.

[2]. Baker B. S., Schwartz J. S. Shelf algorithms for two dimensional packing problems. SIAM J. Computing. 1983. V. 6. N 2. P. 508-525.

[3]. Shor P. W. The average-case analysis of some on-line algorithms for bin packing. Combinatorica. 1986. V. 6. N 2. P. 179-200.

[4]. Coffman E. G., Jr, Shor P. W. Pack ing in two dimensions: Asymptotic average-case analysis of algorithms. Algorithmica. 1993. V. 9. N 3. P. 253-277.

[5]. Kuzyurin N.N., Pospelov A. I. Veroyatrnostnyi analiz novogo klassa algoritmov upakovki pryamougolnikov v polosu [Probabilistic analysis of a new class of strip packing algorithms]. JVMiMF [Journal of computational mathematics and mathematicak physics]. 2011. V. 51, N 10, p. 1931-1936. (in Russian)

[6]. Kuzyurin N.N., Pospelov A. I.. Veroyatnostnyi analiz phelfovyh algoritmov upakovki pryamougolnikov v polosu [Probabilistic analysis of shelf strip packintg algorithms]. Discretnaya matematika [Discrete mathematics]. 2006. V. 18. N 1. P. 76-90. (in Russian)

[7]. Csirik J., Woegenger G.J. Shelf algorithm for on-line strip packing, Inf. Process. Letters, 1997, v. 63, N 4, P. 171-175.

[8]. Trushnikov M. A., Ob odnoi zadache Koffmana-Shora, svyazannoi s urakovkoi pryamougolnikov v pollosu [On Coffman-Shor problem related to packing rectangles into a strip], Trudy ISP RAN [Proceedings of ISP RAS], 2012 , v. 22, p. 456-462. (in Russian)