ИСП Институт Системного Программирования
Российской Академии наук

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

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том 27, выпуск 6

Volume 27, issue 6

Москва 2015

УДК004.45

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

Труды Института системного программирования РАН

Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областям системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge.

Proceedings of ISP RAS is abstracted and/or indexed in:

Содержание

Table of Contents

Предисловие

Этот выпуск «Трудов ИСП РАН» содержит избранные статьи, представленные по итогам работы открытой конференции по компиляторным технологиям, посвященной методам оптимизации и генерации кода, статическому и динамическому анализу программ, а также 6-й международной конференции «Облачные вычисления. Образование. Исследования. Разработка».

Мероприятия прошли 2-3 декабря 2015 года и были организованы ИСП РАН при поддержки Российской академии наук, Российского фонда фундаментальных исследований и компаний-партнеров (Samsung, HP, Dell, NVIDIA и д.р.), совместно с которыми реализуется ряд проектов и программ в области компиляторных технологий и параллельных и распределенных вычислений: "Университетский кластер" (http://www.unicluster.ru), открытая лаборатория по технологиям больших данных (BigDataOpenLab - http://www.bigdataopenlab.ru), исследовательский центр CUDA (CUDA Research Center). Информационным партнером конференции является издательский дом "Открытые системы".

Академик РАН В.П. Иванников

Preface

This issue of "Proceedings of ISA RAS" contains selected papers presented on the results of the open conference on compiler technology, dedicated to techniques of optimization of code generation, static and dynamic analysis of programs, as well as the 6-th International Conference "Cloud computing. Education. Research. Development."

The events took place on 2-3 December 2015 and were organized with the support of ISP RAS, Russian Academy of Sciences, the Russian Foundation for Basic Research and the partner companies (Samsung, HP, Dell, NVIDIA, etc.), in conjunction with which a number of projects and programs in compiler technology and parallel and distributed computing are being developed: "University cluster" (http://www.unicluster.ru), an open laboratory for Big Data technologies (BigDataOpenLab -http://www.bigdataopenlab.ru), CUDA Research center). Media partner of the conference is the publishing house "Open Systems".

Academician V.P. Ivannikov

Метод инструментирования кода на этапе компиляции для направленной отладки оптимизирующих преобразований

Д. А. Максименков <shark@mcst.ru>
ПАО «МЦСТ», РФ, г. Москва, Ленинский пр-т, д. 51

Аннотация. В статье рассматривается проблемы отладки оптимизирующих компиляторов. В качестве эффективного способа повышения надежности производимых компилятором оптимизирующих преобразований автором предлагается новый метод инструментирования кода программы на этапе компиляции. Особенностью описываемого в статье метода является то, что он предназначен в первую очередь для отладки конкретных проблемных оптимизаций, а не самих тестов, и позволяет верифицировать корректность формируемого оптимизацией кода для произвольных входных данных запускаемой задачи. Предлагаемый метод был успешно использован для поиска и выявления нерегулярно проявляющихся ошибок в программах с асинхронно работающим кодом.

Ключевые слова: тестирование, инструментирование кода, отладка оптимизирующих компиляторов.

DOI: 10.15514/ISPRAS-2015-27(6)-1

Для цитирования: Максименков Д.А. Метод инструментирования кода на этапе компиляции для направленной отладки оптимизирующих преобразований. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 7-20. DOI: 10.15514/ISPRAS-2015-27(6)-1.

1. Введение

Одной из проблем при отладке оптимизирующего компилятора является выявление факта того, что в процессе работы скомпилированной задачи была допущена ошибка. Так, некоторые ошибки, допущенные компилятором, могут не приводить к аварийному завершению задачи, а влиять лишь на вычисляемое конечное значение в программе. Если получаемый результат вычислений отличается от эталонного незначительно в пределах допустимой погрешности, то задача считается отработавшей корректно. Таким образом, например, устроены пакеты тестовых задач spec[1]. В этом случае, обнаружить, что в процессе работы компилятора была допущена ошибка, оказывается довольно затруднительно. Зачастую, преимущественно в

больших задачах некоторые неправильно произведенные вычисления могут вообще не влиять на конечный результат работы программы. С точки зрения пользователя, такие ошибки компилятора не являются критичными, т.к. не оказывают какого-либо влияния на конечный результат вычисления конкретной задачи. Но для целей отладки оптимизирующего компилятора выявление некорректных преобразований, произведенных компилятором, являются важной задачей. Ведь такие оптимизации при компиляции других программ могут привести к их неправильной работе или даже слому. Из-за ограниченности набора тестовой базы, вычислительных мощностей, человеческих и временных ресурсов возможность выявления любых ошибок оптимизатора (а не только тех, которые приводят к явному слому или неправильной работе задачи) является эффективным средством повышения надежности оптимизирующего компилятора.

В случае, когда ошибка, допущенная компилятором, выявляется самой задачей в процессе ее работы (падение или различие в печатаемых результатах вычислений), также возникает необходимость в локализации момента появления ошибки в коде. Как правило, неправильно произведенные вычисления приводят к падению задачи далеко от момента самой ошибки или отражаются только в конечном подсчитанном значении после завершения работы всей задачи, что сильно затруднит анализ проблемы.

Одним из широко используемых методов поиска ошибок в компиляторе является метод инструментирования кода. Например, проекты Valgrind[2], AddressSanitizer[3] предназначены для отладки программ. В качестве отлаживаемой программы можно использовать, в том числе, и сам оптимизирующий компилятор. При этом в бинарный код оптимизирующего компилятора встраиваются различные проверки, предназначенные для выявления ошибок определенного класса, таких как выход за границу массива, использование неинициализированных данных, обращение по невалидному адресу в коде самого компилятора. Подавая такому компилятору различные входные данные – т.е. разнообразные тесты, можно находить контексты для работы большинства анализов и запуска целого ряда оптимизаций, проверяя тем самым корректность работы написанного кода компилятора. Т.е. данные методы позволяют отлаживать компилятор как пользовательскую задачу. При этом не проверяется логика работы самих оптимизаций, проводимых над кодом задачи. Существуют методы [4], которые позволяют на уровне исходных кодов инструментировать тесты, используемые в качестве входных данных для оптимизирующего компилятора. Тесты с такими встроенными внутренними самопроверками во время своей работы в определенных местах программы проверяют корректность производимых промежуточных вычислений на соответствие эталонным, заранее подсчитанным константам. Инструментированные таким образом задачи способны эффективно выявлять ошибки в различных оптимизациях компилятора, если код теста в результате произведенных

компилятором преобразований оказался неэквивалентным исходному и стал работать по-другому. Однако в результате такого инструментирования изменяется сам исходный код задачи, что в конечном итоге может повлиять на результаты анализов и на применение различных оптимизаций. В результате возможна ситуация, когда ошибка проявляется при компиляции исходного кода теста и перестает проявляться после инструментирования этого же кода дополнительными проверками. Помимо этого, встроенные проверки рассчитаны на фиксированный набор входных данных. Если запускаемый тест предполагает возможность использования различных исходных данных (например, [1]), то для каждого входного набора данных инструментирование кода придется проводить заново.

Предлагаемый в статье метод производит инструментирование кода задачи в момент применения оптимизирующего преобразования, т.е. на этапе компиляции теста. Поэтому исходный код задачи никак не модифицируется, а встраиваемые проверки не зависят от подаваемых входных для теста данных. Особенностью описываемого в статье метода является то, что он предназначен для отладки отдельных проблемных оптимизаций и позволяет верифицировать корректность формируемого оптимизацией кода для произвольных входных данных запускаемой задачи. Встраиваемые в код самопроверки не влияют на результаты анализов и, как следствие, на применимость отлаживаемой оптимизации. Иначе говоря, они не вносят возмущение в работу компилятора, т.к. реализуются непосредственно после формирования кода производимой оптимизации и лишь дополняют его. Так же, как и другие методы, данный метод позволяет оперативно выявлять ошибки и локализовывать место и контекст их проявления в работающем коде программы. Метод применим для широкого класса оптимизаций и способен находить ошибки в оптимизирующих преобразованиях даже в тех случаях, когда сам тест проблем не выявляет, а известные способы инструментирования программ не способны их обнаруживать в силу ограниченности класса выявляемых ими проблем.

2. Ошибки, возникающие при работе оптимизаций

Для возможности создания эффективно исполняемого кода компилятор преобразует задачи, написанные на разных высокоуровневых языках программирования, в команды, представляющие собой внутренний абстрактный язык компилятора — промежуточное представление, удобное для дальнейшей работы оптимизаций. Оптимизации последовательно применяются к операциям промежуточного представления, тем самым постепенно модифицируя код программы в терминах внутреннего языка, делая его более эффективным. В конечном итоге после применения всей цепочки оптимизаций полученное финальное промежуточное представление оптимизированной программы преобразуется в ассемблер целевой

архитектуры. Далее, уже с помощью ассемблера и линковщика, формируется объектный и исполняемый код.

Количество применяемых оптимизаций над командами промежуточного представления может быть довольно большим и зависеть как от режима компиляции (от подданных программистом опций компиляции), так и от самой задачи (компилятор может динамически, в процессе своей работы принимать решение о использовании тех или иных оптимизаций). Оптимизации применяются пофазно: от одного промежуточного состояния представления к другому. Одна фаза может содержать в себе запуск сразу несколько оптимизаций, работающих сообща. Количество таких последовательно выполняемых фаз в оптимизирующем компиляторе, как правило, достигает нескольких сотен.

Оптимизирующие преобразования производятся над одной или несколькими операциями промежуточного представления. Компилятор в процессе оптимизаций заменяет их на другие, более эффективные команды. Получаемая последовательность операций должна быть эквивалентна преобразуемой, изначальной группе команд. В противном случае оптимизированный код будет содержать в себе ошибки и может работать неверно.

Ошибки в оптимизациях, проявляющиеся на широком диапазоне входных данных задачи, как правило, находятся довольно легко. Например, если выражение

```
a = b * c;
```

в процессе оптимизаций компилятором было ошибочно заменено на

```
a = b + c;
```

то практически при любых входных значениях переменных b и c результат преобразования окажется неверным. Такую ошибку в программе будет обнаружить несложно — выражение будет получать неверный результат практически при любых входных данных. Причем вычисляемое значение будет отличаться от ожидаемого значительно.

Однако существует другой класс ошибок, проявляющийся только при определенном наборе входных данных. Для получения более эффективный код часть оптимизаций производит преобразование, которое будет эквивалентным лишь при выполнении определенного ряда условий, например, при ограниченном наборе входных аргументов операций, к которым применяется данная оптимизация.

Так, выражение

```
f2 = f1 * 0.0;
```

где f1 и f2 — это вещественные числа,

можно заменить на эквивалентное выражение

```
f2 = 0.0;
```

Однако такое преобразование будет эквивалентным не для любых входных значений f1. Например, при f1 = nan (не число), согласно стандарту работы вещественной арифметики IEEE-754 [5] значение f2 также должно принимать значение nan. Производить описанное выше преобразование можно лишь в том случае, когда компилятору достоверно известно, что в вещественных вычислениях участвуют только нормальные числа (т.е. среди них нет nan, qnan, inf и т.п.). В данном случае такую информацию может сообщить компилятору программист с помощью специализированной опции компиляции, указав оптимизациям, что такие преобразования разрешены.

Большинство оптимизаций применяются как раз в предположении выполнения ряда условий, определяемых компилятором статически. Т.е. решение о выполнение условий, для которых применяемое преобразование будет эквивалентным, принимается в процессе работы компилятора. И по результатам этого решения одни операции заменяются на другие, эквивалентные изначальным, но при выполнении определенных условий. Выявлять ошибки в таких преобразованиях бывает довольно сложно, т.к. для их проявления необходимо появление контекста, не учтенного оптимизацией. Т.е. для некоторых входных данных задачи построенный оптимизатором код будет некорректным. И чем меньше диапазон входных данных, для которых сформированный оптимизацией код работает неверно, тем реже проявляется ошибка и тем сложнее будет ее обнаружить.

3. Описание метода инструментирования кода при работе оптимизаций

Покажем суть метода на простом примере. Допустим, выражение

```
b = a / 2;
```

в процессе работы одной из оптимизаций заменили на

```
b = a >> 1;
```

Данная замена будет эквивалентной для всех положительных значений a, а также для a = 0 и для ряда отрицательных значений a (а именно, четных). Для нечетных отрицательных a преобразование даст другой результат, отличающийся на 1. В данном случае диапазон значений a, при которых оптимизация работает корректно, оказывается в несколько раз больше, чем тот набор значений, при которых будет возникать ошибка. Поэтому очень важным является гарантия выполнения условий применимости оптимизации перед использованием их компилятором. В данном примере нужно гарантировать, что a — неотрицательное или четное число.

Рассматриваемый в работе метод позволяет обнаруживать преобразования, некорректно произведенные оптимизирующим компилятором, в т.ч. и для тех

случаев, когда ошибка проявляется на ограниченном контексте входных условий, а значит, имеет редкое (а в некоторых случаях и недетерминированное) проявление, т.е. когда компилятор на основе статического анализа производит замену одной группы операций на другую (эквивалентную, например, только для определенного контекста). В таких случаях возможна ошибка как в самом преобразовании, так и в корректности определении контекста для применяемого преобразования.

Для проверки эквивалентности замены одной группы операций на другую было предложено не удалять первую группу операций, а оставить ее в коде программы и использовать для получения эталонного значения с целью последующего сравнения его с результатом, получаемым от новой группы операций. В случае несличения подсчитанных значений от двух групп операций работа теста аварийно завершается, что оперативно сигнализирует о найденной ошибке. Попутно с этим, предложено сохранять информацию о контексте (например, значение аргумента для модифицируемой операции), который привел к проявлению ошибки. Т.е. на рассмотренном выше примере, если до применения оптимизации имелось выражение

```
b = a / 2;
```

то после преобразования изначальный код дополняется динамической проверкой:

```
b1 = a / 2;
b = a >> 1;
if(b != b1) Error(a);
```

Наравне с новым, более эффективным способом вычисления переменной b (с помощью побитового сдвига вправо) в коде сохраняется оригинальный способ вычисления выражения (с помощью операции целочисленного деления) с последующим сохранением результата во временную переменную b1. Далее идет сличение значений b и b1 и в случае их неравенства происходит аварийное завершение работы теста. Если случится, что переменная a примет отрицательное нечетное значение, при котором производимая оптимизация является некорректной, то об этом сразу же станет известно из-за аварийного завершения программы. Досрочное, аварийное завершение программы позволяет выявлять и локализовать ошибки, допущенные компилятором при построении оптимизированного кода программы, даже если они не влияют на конечный результат работы теста.

С помощью описанного метода можно проверять корректность работы широкого класса простых правил замены одних операций на другие (оптимизации peephole [6]), которых в оптимизирующем компиляторе, как правило, насчитывается несколько сотен. Сохранение изначальной цепочки операций для получения эталонного значения и сравнение его с результатом

вычисления сформированных оптимизацией операций гарантирует выявление ошибки в случае, если компилятор построил неэквивалентный оптимизированный код для подаваемых входных данных.

Еще одной областью применения данного метода в оптимизирующем компиляторе является проверка корректности работы анализов разрыва зависимости между операциями обращения в память. Например, это могут быть операции записи (store) и чтения (load) данных или операции, неявно работающие с памятью, например, команды вызова функций (call) и системных вызовов (syscall). К сожалению, далеко не всегда удается корректно определить независимость операций обращения в память. Адресные аргументы операций load и store, как правило, статически неизвестны или динамически изменяются, например, в цикловых конструкциях. А возможность модификации функциями конкретных ячеек памяти зачастую неочевидна. В компиляторе используются несколько десятков интерфейсов определения независимости для заданных пар операций обращения в память, использующих различные анализы (например, [7], [8], [9]). Для определения независимости операций в коде программы могут также применяться различные подсказки (pragma, restrict), добавляемые, как правило, автором задачи для возможности более эффективной работы оптимизатора, и даже пользовательские опции (-frestrict-params, -frestrict-all, -fstrict-aliasing). Ошибка в любой из перечисленных компонент приведет к ложному признанию независимости для операций чтения и записи в память с совпадающими или пересекающимися адресами.

Цель оптимизирующего компилятора – сформировать как можно более быстро работающий код. Для этого операции в коде переставляются таким образом, чтобы загрузить данные из памяти на внутренние регистры процессора как можно раньше. Операции load могут работать довольно долго в зависимости от того, где именно находятся запрашиваемые данные (в L1, L2, L3-кэш памяти процессора или в ОЗУ). Именно поэтому многие анализаторы стремятся доказать независимость операции load с другими командами (store, call), чтобы разорвать зависимости между конфликтующими операциями. При отсутствии таких зависимостей команды чтения данных из памяти поднимаются и планируются выше по коду, чем операции store и call.

Поясним вышесказанное на примере (рис.1).

Операции в коде до применения оптимизации:	Операции в коде после применения оптимизации:
store [mem1] ← var1	load [mem2] → var2
…	…
load [mem2] → var2	store [mem1] ← var1
	…
	load’ [mem2] → var3
	if (var2 != var3) Error(mem2);

Рис.1

До применения оптимизации, использующей статический анализ адресов mem1 и mem2, в коде промежуточного представления была операция store, пишущая по адресу mem1 значение переменной var1, и стоящая ниже по коду операция load, читающая в переменную var2 данные из памяти mem2. Для случая, когда один из анализов дал положительный ответ о различии адресов mem1 и mem2, оптимизация поднимает операцию load выше store. Чтобы проконтролировать корректность принятого анализом решения, в коде на месте изначальной операции load создают другую, похожую операцию load', работающую по тому же адресу mem2, но сохраняющую значение во временной переменной var3. Такую операцию помечают во внутреннем представлении как заведомо конфликтующую с операцией store, чтобы впоследствии к ней уже нельзя было применить похожее преобразование повторно. Далее результаты прочитанных значений в переменных var2 и var3 сравнивается, и в случае их несоответствия происходит аварийное завершение работы программы с фиксацией факта ошибки в работе оптимизации для адреса mem2.

По имеющейся многолетней статистике ошибок надежности, выявленных в процессе отладки оптимизирующих компиляторов, ложное признание независимости конфликтующих операций load и store — одна из самых часто встречающихся ошибок оптимизирующего компилятора. Применение описываемого метода позволяет выявить ошибки, связанные с некорректным разрывом зависимости между конфликтующими операциями обращения к пересекающимся ячейкам памяти и последующей их перестановкой.

4. Применение метода на практике

Предлагаемый в работе метод был использован для поиска ошибок и отладки оптимизирующего компилятора для процессора Эльбрус [10] при работе с асинхронным механизмом предварительной подкачки данных из массивов, находящихся в памяти. Асинхронность работы предварительной подкачки данных из памяти приводила к случайному проявлению ошибок в разных местах программы (например, на разных итерациях цикла). Для такой ситуации было крайне важным в случае проявления ошибки идентифицировать момент возникновения проблемы и контекст, при котором проявлялась ошибка. Асинхронная программа запускается в определенный момент работы основного кода программы и далее работает параллельно ему. Ошибка могла возникать (или не проявляться) в зависимости от того, в какой именно момент происходит считывание данных из памяти. Предварительная подкачка данных для массивов из памяти устроена следующим образом. Вначале компилятором определяется место в коде основной программы, начиная с которого возможен запуск асинхронной подпрограммы — момент начала предварительной подкачки (операция bap). Перед запуском самой подпрограммы предварительной подкачки настраиваются специализированные регистры, определяющие параметры считываемых

данных: начальный адрес памяти, по которому будут загружаться данные, расстояние между последовательными чтениями, направление изменение адреса считывания и т.д. После запуска асинхронной подпрограммы в отдельном потоке считываемые данные из памяти попадают в буфер предварительной подкачки. Все это происходит параллельно работе основного кода. В самом коде программы чтения данных из памяти с помощью обычных операций load заменяются на специализированные операции mova – пересылки уже прочитанных ранее значений из буфера предварительной подкачки в переменные программы. Время доступа при обращении к такому буферу — минимально и постоянно в отличие от операций load. Рассматриваемая оптимизация apb – Array Prefetch Buffer[11] состоит в выявлении регулярных цикловых чтений, построении асинхронного кода и синхронного кода подготовки и запуска асинхронной программы предварительной подкачки; при этом операции load заменяются на mova. Однако далеко не любая операция load может быть заменена на mova. Определение того, какие именно данные можно прочитать заранее, т.е. для каких ячеек памяти нет конфликтующих операций записи (store), и определение момента в коде программы, начиная с которого можно запускать асинхронную подпрограмму, является условием применимости данной оптимизации. Кроме того, ошибка в настройке специализированных регистров, может влиять на корректность работы считываемых значений программой предварительной подкачки.

В данном случае эти условия определяются статически, на этапе компиляции. Т.е. компилятор в процессе своей работы определяет те операции load, которые не конфликтуют с другими операциями, и заменяет их на операции mova. Если независимость операций была определена некорректно (между моментом запуска bap асинхронной программы и операцией mova имеется модификация читаемой ячейки памяти), то в зависимости от того, когда именно асинхронная подпрограмма прочтет данные из памяти (до (область A) или после (область B) конфликтующей операции store) зависит корректность почитаемого значения (см рис.2). Отсюда и появляется недетерминированное поведение оптимизированной программы: от запуска к запуску данные, получаемые с помощью операции mova, могут содержать в себе корректные или неверные значения.

Код до применения оптимизации apb:

```
store [mem1] ← %reg1
…
load [mem2] → %reg2
```

Код после применения оптимизации apb:

```
bap
// область A
store [mem1] ← %reg1
// область B
mova → %reg2
```

Рис.2

При равенстве адресов mem1 и mem2 (или даже их частичном пересечении, с учетом формата операций load и store) получим некорректный код, который может загружать различные значение в регистр %reg2.

Чтобы можно было оперативно выявлять данный класс ошибок и идентифицировать условия, при которых оптимизация apb отрабатывает некорректно, в компиляторе, на фазе apb был реализован отладочный режим генерации кода с динамической проверкой результата. По отладочной опции компилятора оптимизация apb не удаляет операцию load, к которой применяется преобразование, и в дополнение к новой созданной операции mova добавляет код самопроверки (см рис.3).

Код до применения оптимизации apb:	Код после применения оптимизации apb:

store [mem1] ← %reg1
…
load [mem2] → %reg2
	bap
	// область A
	store [mem1] ← %reg1
	// область B
	mova → %reg2
	load [mem2] → %reg3
	cmp %reg2, %reg3 → %reg3
	div mem2, %reg3

Рис.3

В данном случае, результаты операций load и mova сохраняются в регистрах %reg2 и %reg3 соответственно. Регистр %reg3 при этом используется как временный регистр для хранения промежуточных значений. Если асинхронная подпрограмма предварительной подкачки загрузила данные в буфер до модификации памяти операцией store, то в случае mem1 = mem2 результат операции сравнения cmp двух регистров %reg2 и %reg3, содержащих разные значения, будет ложным (т.е. равным нулю). Далее нулевой %reg3 используется в качестве делителя целочисленной операции div. Деление на ноль приведет к исключительной ситуации и аварийному завершению работы программы. В случае, когда прочитанные значения операциями load и mova совпадают, результат операции cmp будет истинным (отличным от нуля). В таком случае, операция целочисленного деления завершится без побочных эффектов. В качестве первого аргумента у операции div используется значение адреса mem2. При возникновении целочисленного деления на ноль доступность (например, под отладчиком) информации о первом аргументе сломавшейся операции даст информацию о условиях проявления ошибки. В данном случае это значение адреса mem2, обращение к которому компилятором было ошибочно признано независимым с другими операциями программы, находящимися после команды bap.

Возможны различные вариации реализации данного алгоритма. Вместо адреса mem2 в качестве делимого можно сохранять другую информацию, например, счетчик цикла, чтобы узнать на какой именно итерации цикла произошла ошибка в программе. Вместо операции целочисленного деления можно использовать другие команды, например, операцию обращения в память, формируя в ней при помощи регистра %reg3 нулевой или корректный адрес[4].

Также нужно отметить компактность встраиваемой динамической проверки, линейность полученного кода и отсутствие операций вызова различных функций (exit, abort и т.д), используемых для аварийного завершения работы программы. В этом случае вносится минимальное возмущение в оптимизируемый код, что благоприятно способствует дальнейшему применению других оптимизаций к модифицируемой цепочки операций. Так, например, операции вызова функций являются полюсом для применения целого ряда оптимизирующих преобразований с ограничением возможности переноса других команд через точку вызова. Код, построенный в виде динамической самопроверки, не препятствует использованию результата из регистра %reg2 в дальнейших вычислениях программы и может работать параллельно с ним.

5. Ограничения метода

Данный метод встраивания в код динамической проверки корректности оптимизаций можно использовать при отладке довольно широкого класса оптимизаций, таких как peephole и перестроение операций, на основе различных анализов зависимостей. Однако для некоторых видов преобразований описанный выше алгоритм неприменим или применим с дополнительными ограничениями. Метод нельзя применять, если среди модифицируемых операций присутствуют вызовы некоторого класса функций, меняющих глобальный контекст, например, при наличии функций печати (printf). Если такой код исполнить два раза (оптимизированный и оригинальный вариант), то функция печати отработает дважды, что нарушит логику программы. В некоторых случаях, данный метод можно применять, но с дополнительными модификациями. Если входные данные для фрагмента кода изменяются в процессе его же работы, то повторное их использование уже будет некорректным. В таком случае нужно предварительно скопировать модифицируемые входные во временные регистры для возможности их повторного использования.

6. Результаты

Как уже было сказано, описанный метод был успешно реализован в оптимизирующем компиляторе для процессоров Эльбрус и использовался при поиске и выявления нерегулярно проявляющихся ошибок в оптимизации apb. По результатам тестирования в течение 8 месяцев отладки было выявлено

порядка двух десятков ошибок надежности в данной оптимизации и несколько ошибок в других оптимизациях, также работающих с операциями предварительной подкачки. Время жизни большинства найденных ошибок оказалось довольно большим (составило несколько лет). После исправления всех выявленных проблем нерегулярные падения оптимизированных кодов больше проявлялись. Помимо этого, тесты со встроенными динамическими самопроверками для операций предварительной подкачки позволили выявить и исправить аппаратные ошибки в механизме предварительной подкачки в процессоре Эльбрус во время его верификации на прототипе, а также успешно используются в настоящее время для отладки и поиска ошибок на прототипах новых процессоров.

Список литературы

[1]. Standart Perfomance Evaluation Corporation. The SPEC Benchmark Suites http://www.spec.org
[2]. Valgrind http://www.valgrind.org
[3]. AddressSanitizer (ASan) http://www.chromium.org/developers/testing/addresssanitizer
[4]. Максименков Д.А., Рогов Р.Ю. Применение метода инструментирования тестовых программ при отладке оптимизирующих компиляторов. Вопросы радиоэлектроники, 2010, вып. 3, стр. 50-61
[5]. Standard for Binary Floating-Point Arithmetic IEEE-754. http://www.ieee.org
[6]. D.A. Lamb, Construction of a Peephole Optimizer. Software – Practice & Experience, 11/ 639-647 – 1981.
[7]. Michael Jind Pointer analysis: Haven't we solved this proplem yet http://www.cs.cornell.edu/courses/cs711/2005fa/papers/hind-paste01.pdf
[8]. Amer Divan, Kathryn S.McKinley, J.Eliot B.Moss. Type-Based Alias Analysis http://web.cs.ucla.edu/~palsberg/tba/papers/diwan-mckinley-moss-pldi98.pdf
[9]. Дроздов А.Ю, Владиславлев В.Е. Межпроцедурный анализ указателей. Информационные технологии, 2005, приложение No 2, стр 35-42
[10]. Волконский В.Ю. Оптимизирующие компиляторы для архитектуры с явным параллелизмом команд и аппаратной поддержкой двоичной совместимости. Информационные технологии и вычислительные системы - 2004 — Вып.3
[11]. Галазин А.Б, Степаненков, А.М., Ступаченко, Е.В. Программная предварительная подкачка кода для микропроцессора Эльбрус-3М. Информационные технологии, 2007, вып. 11.

Compile the Code Instrumentation Technique for Selective Debugging of Optimizing Transformations

D. Maksimenkov <shark@mcst.ru>
PAO «MCST», 51 Leninsky, Moscow, Russia

Abstract. The paper addresses the problem of an optimizing compiler debugging. A new method for compile-time instrumentation is presented as an efficient approach to improve reliability of optimizing transformations implemented in the compiler. The principal feature of the method is that it aims at debugging the transformations itself rather than debugging the tests, and therefore it allows correctness of the resulting code to be verified on any input data fed into the executed program, i.e. the proposed self-checking instrumentation is orthogonal to particular input data in that sense it is able to detect the bugs on arbitrary data flow the program exhibits during its invocation.

The method can be applied to a wide set of optimizing transormations. And the cases are known when it reveals faults in transformations while the non-instrumented test itself remains fully functional (and other somewhat similar but more limited instrumentations reveal no bugs, too). Among other its features are the compactness of the embedded dynamic checkers, the linearity of code bloat, and also no assumptions on any standard libraries availability is made (eg. no functions like exit(), abort() etc are used to terminate the program if any checker is triggered). It all ensures that the optimized code is influenced minimally so the original sequence of optimizing transormations implemented in the compiler remains applicable (if compared to non-instrumented code).

The described method has demonstrated successfully its usability for detecting and identifying volatile bugs in the optimizing compilers of the Elbrus microprocessor family when they were used for building software with asynchronous control flow. Moreover it allowed to increase the reliability of not only the compiler itself but also the software being compiled as well as it shown its use in complex testing of the microprocessor prototypes.

Keywords: testing, instrumentation code, debugging optimizing compilers

DOI: 10.15514/ISPRAS-2015-27(6)-1

For citation: D. Maksimenkov. Compile the Code Instrumentation Technique for Selective Debugging of Optimizing Transformations. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 7-20 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-1

References

[1]. Standart Perfomance Evaluation Corporation. The SPEC Benchmark Suites http://www.spec.org
[2]. Valgrind http://www.valgrind.org
[3]. AddressSanitizer (ASan) http://www.chromium.org/developers/testing/addresssanitizer
[4]. Maksimenkov D.A., Rogov R.Y. Primenenie metoda instrumentirovaniya testovyh program pri otladke optimiziruyushcih kompilyatorov [Application of instrumentation

test program debugging optimizing compilers]. Voprosy radioelektroniki [Questions electronics], 2010, no. 3, pp. 50-61 (in Russian).

[5]. Standard for Binary Floating-Point Arithmetic IEEE-754. http://www.ieee.org

[6]. D.A. Lamb, Construction of a Peephole Optimizer. Software – Practice & Experience, 11/ 639-647 – 1981.

[7]. Michael Jind Pointer analysis: Haven't we solved this proplem yet http://www.cs.cornell.edu/courses/cs711/2005fa/papers/hind-paste01.pdf

[8]. Amer Divan, Kathryn S.McKinley, J.Eliot B.Moss. Type-Based Alias Analysis http://web.cs.ucla.edu/~palsberg/tba/papers/diwan-mckinley-moss-pldi98.pdf

[9]. Drozdov A.Y., Vladislavlev V.E. Mezhprocedurny analiz ukazatelej [Interprocedural analysis pointer]. Informacionnie tehnologii [Information technology], 2005. att No 2, pp 35-42 (in Russian).

[10]. Volkonsky V.Y. Optimiziruyuschie kompilyatory dly arhitektur s yavnim parallelizmom komand I apparatnoj podderzhkoj dvoichnoj sovmestimosti [Optimizing compilers for architectures with explicitly parallel instruction and hardware support for binary compatibility] Informacionnie tehnologii i vychislitelnie sistemy [Information technology and computer systems], 2004, no..3 (in Russian).

[11]. Galazin A.B., Stepanenkov A.M., Stupachenko E.V. Programmnaya predvaritelnaya podkachka koda dlya mikroprocessora Elbrus-3M [Spooling software code for the microprocessor Elbrus-3M]. Informacionnie tehnologii [Information technology], 2007, no. 11 (in Russian).

Approaches to Optimizing V8 JavaScript Engine

Dmitry Botcharnikov <dmitry.b@samsung.com>
LLC Samsung R&D Institute Rus, 12, ul. Dvintsev, housing 1, office #1500, Moscow, 127018, Russian Federation

Abstract. JavaScript is one of the most popular programming languages in the world. Started as a simple scripting language for web browsers it now becomes language of choice for millions of engineers in the web, mobile and server-side development. However its interpretational nature doesn't always provide adequate performance. To speed up execution of JavaScript programs there were developed several optimization techniques in recent years. One example of modern high-performing JavaScript engine is a V8 engine used in Google Chrome browser and node.js web server among others. This is an open source project which implemented some advanced optimization methods including Just-in-Time compilation, Polymorphic Inline Caches, optimized recompilation of hot code regions, On Stack Replacement &c. In previous year we were involved in project of optimizing performance of V8 JavaScript engine on major benchmark suites including Octane, SunSpider and Kraken. The project was quite time limited, however we achieved about 10% total performance improvement compared to open source version. We have decided to focus on following approaches to achieve the project's goal: optimized build of V8 itself, because total running time is shared between compilation and execution; tuning of V8 runtime options which default values may not be always optimal; implementation of additional scalar optimizations. All of these approaches have made contribution to final result.

Ключевые слова: JavaScript; optimizations; V8; common subexpression eimination

DOI: 10.15514/ISPRAS-2015-27(6)-2

For citation: Botcharnikov Dmitry. Approaches to Optimizing V8 JavaScript Engine. *Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015,* pp. 21-32 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-2

1. Introduction

JavaScript is one of the most popular programming languages in the world [1]. Started as a simple scripting language for web browsers it now becomes language of choice for millions of engineers in the web, mobile and server-side development. However its interpretational nature doesn't always provide adequate performance.

To speed up execution of JavaScript programs there were developed several optimization techniques in recent years. One example of modern high-performing JavaScript engine is a V8 engine [2] used in Google Chrome browser and node.js web server among others. This is an open source project which implemented some advanced optimization methods including Just-in-Time compilation [3], Polymorphic Inline Caches [4], optimized recompilation of hot code regions, On Stack Replacement [5] &c.

In previous year we were involved in project of optimizing performance of V8 JavaScript engine on major benchmark suites including Octane [6], SunSpider [7] and Kraken [8]. The project was quite time limited, however we achieved about 10% total performance improvement compared to open source version.

The rest of paper is organized as follow: in Section 2 there is an architectural overview of V8, in Section 3 we enumerate and reason our approaches with more detailed discussion in Sections 4, 5, and 6. We conclude in Section 7.

2. V8 engine architecture

In contrast to other JavaScript engines V8 implements compilation to native code from the beginning. It consists of two JIT compilers: the first (called Full code generator) performs fast non-optimized compilation for every encountered JavaScript function, while the second one (called Crankshaft) compiles and optimizes only those functions (and loops) which already ran some amount of time and are likely to run further.

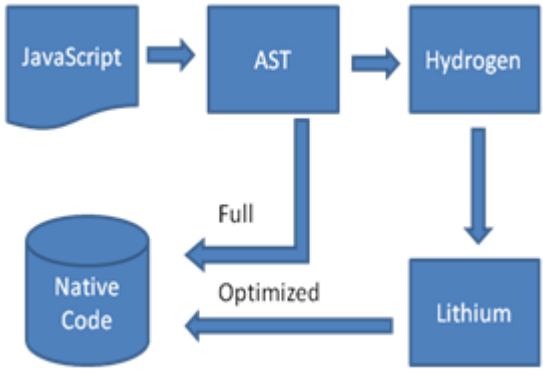


Fig. 1 V8 Engine Architecture

The overall work of V8 engine is as follows (Fig.1):

- Every new script is preliminary scanned to separate each individual function.
- The function that should run is compiled into Abstract Syntax Tree (AST) form.
- AST is compiled into native machine code instrumented with counters for

function calls and loop back edges.

- Also on method call sites V8 inserts special dispatch structure called Polymorphic Inline Cache (PIC). This cache is initialized with call to generic dispatch routine. After each invocation PIC is populated with direct call to type specific receiver up to some predefined limit. In such way PICs collect runtime type information of objects.
- The result code then runs.
- When instrumentation counters reach some predefined threshold, “hot” function or loop is selected for optimized recompilation.
- For this purpose V8 one more time recompiles selected function in AST form. But in this case it also performs optimizations.
- It compiles AST into Static Single Assignment (SSA) form (called Hydrogen) and propagates type information collected by PICs along SSA edges.
- Then it performs several optimizations on this SSA form using type information.
- After that it generates low level representation (called Lithium), does Register Allocation and generates optimized native code which then runs.

Note that V8 optimizing compiler performs much less transformational passes than common ahead-of-time compilers (e.g. gcc, clang/llvm). The reasons behind this we further discuss in Section 6.

3. Approaches to speed up V8 engine

To investigate possible areas of V8 optimization we have performed V8 engine profiling on ARM platform with three different profiling tools: Perf [9], ARM Streamline [10] and Gprof [11]. Each of those has advantages and disadvantages over others but results are very close: V8 JavaScript engine has no ‘hot’ functions in itself that need to be optimized. Different methods show different functions in order of share to total execution time. This is clear evidence that individual function’s contribution is very small compared to precision of measurement. Thus optimization of individual functions can’t achieve much increase in performance.

In following table object identified as perf-2549.map is a code generated by V8 engine.

Overhead	Shared Object	Symbol
65.11%	perf-2549.map	0x5aba4000
0.76%	d8	v8::internal::Scanner::ScanIdentifierOrKeyword()
0.75%	d8	v8::internal::IncrementalMarking::Step (int,v8::internal::IncrementalMarking::CompletionA

		ction)
0.66%	[kernel.kallsyms]	_raw_spin_unlock_irqrestore
0.65%	libc-2.17.so	memchr
0.52%	d8	v8::internal::Heap::DoScavenge(v8::internal::ObjectVisitor*, unsigned char*)
0.51%	libc-2.17.so	0x0004f3ac
0.50%	d8	int v8::internal::FlexibleBodyVisitor<v8::internal::NewSpaceScavenger, v8::internal::JSObject::BodyDescriptor, int>::VisitSpecialized<20> (v8::internal::Map*, v8::internal::HeapObject*)
0.44%	d8	void v8::internal::ScavengingVisitor<(v8::internal::MarksHandling)1, (v8::internal::LoggingAndProfiling)0>::EvacuateObject <(v8::internal::ScavengingVisitor<(v8::internal::MarksHandling)1, (v8::internal::LoggingAndProfiling)0>::ObjectContents)1, 4>(v8::internal::Map*, v8::internal::HeapObject**, v8::internal::HeapObject*, int)
0.43%	d8	v8::internal::ScavengeWeakObjectRetainer::RetainAs(v8::internal::Object*)
0.43%	d8	v8::internal::Scanner::Scan()
0.37%	[kernel.kallsyms]	__memzero

Fig. 2 Several top entries from detailed profile of Octane benchmark by V8 on Linux.

We have decided to focus on following approaches to achieve the project's goal:

- Optimized build of V8 itself, because total running time is shared between compilation and execution.
- Tuning of V8 runtime options which default values may not be always optimal.
- Implementation of additional scalar optimizations.

All of these approaches have made contribution to final result.

4. Optimized build

We have decided to investigate Link Time Optimization [12] and platform options tuning [13]. The latter gave us small outcome (~0.5%) while former have decreased performance.

We have made investigation on Arndale ARM (Samsung Exynos 5250 CPU) development board running Linux with Linaro gcc 4.7 toolchain for the first investigation and the same board running Android 4.4 with Android NDK 9 Linaro toolchain for the second one.

We have specified the following platform options:

- -O3 for highest optimization level
- -mcpu=cortex-a15 for target CPU.

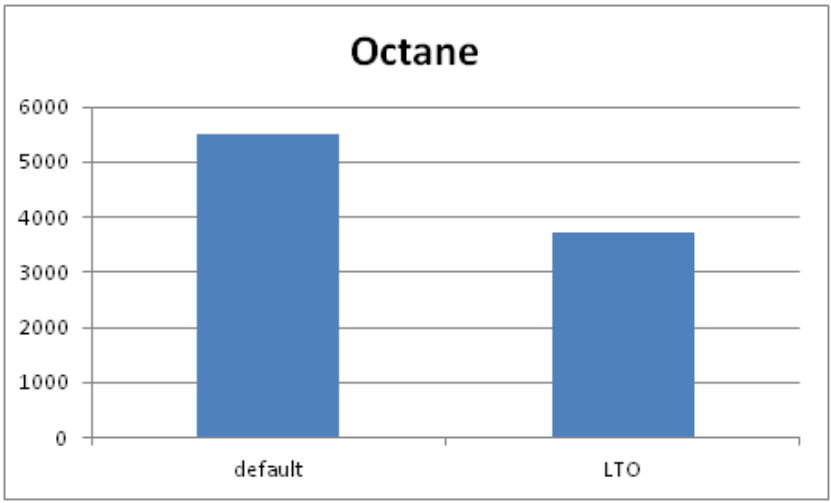


Fig. 3 Effect of LTO

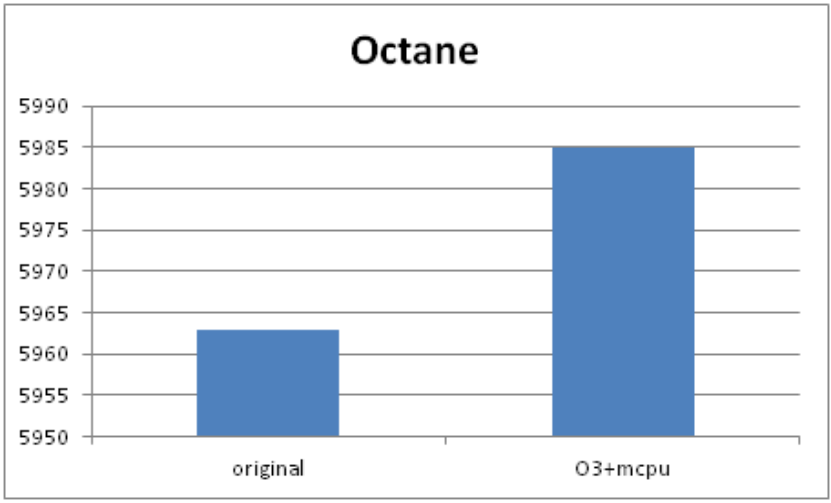


Fig. 4 Effect of platform options tuning

5. Runtime parameters tuning

V8 engine has quite large set of parameters which guides JIT compilation and execution of JavaScript programs. We have found that their default values are not adequate in all cases, e.g. we have found that disabling lazy compilation can substantially improve performance.

As noted in Section 2 V8 performs preliminary parsing of each new script source to separate each individual function. However when we specify parameter '--no-lazy', it instead compiles all functions at once in given script.

Enabling this mode has various impacts on different benchmark. We can see big degradation of CodeLoad test score by about 40% while in the same time huge increase 2.5 times of MandreelLatency test score. The overall increase about 5% was also reproduced on Galaxy Note 3 devices running Android 4.4.

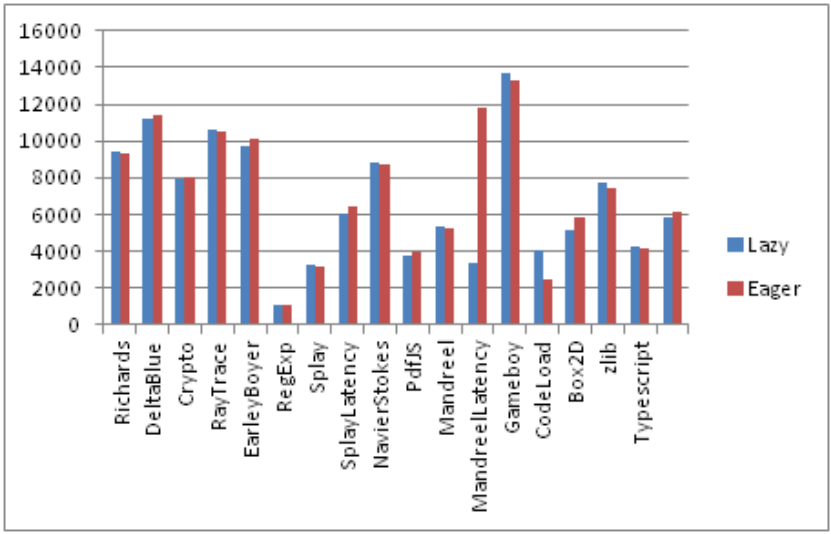


Fig. 5 Effect of eager compilation on Octane benchmarks.

6. Scalar optimizations

We have tried to implement several well-known scalar optimizations in V8 however with varying success. In contrast to ahead of time compilers for classic imperative languages such as C/C++, Pascal, Ada &c., just-in-time compiler has to share time among analysis, optimization and execution. That's why sophisticated optimizations which require thorough analysis don't necessarily lead to increasing performance in such case.

As noted in Section 2 the V8 engine performs optimized compilation of 'hot' regions similar to off-line compiles did. At this stage PICs already collected type

information so we can apply well-known scalar optimization techniques in AST and SSA representations.

The platform used in benchmark was Samsung Galaxy Note 3 with Qualcomm Snapdragon (N9005) CPU. Devices run Android 4.4.2 (KitKat). Octane benchmark suite used in tests was Version 9 download from corresponding repository. For development we use Android NDK r9c on Linux x86_64 Ububtu 12.04 TLS

6.1 Algebraic Simplification

The Algebraic Simplification uses algebraic identities like *a - 0 = a* to simplify expressions. This transformation was implemented in V8 parser when it builds AST representation for Crankshaft.

As was noted above at this point we have collected type information so we can safely optimize algebraic expression given that operands are numeric.

Despite the large amount of optimized expressions in Octane benchmark suite the final result was very small.

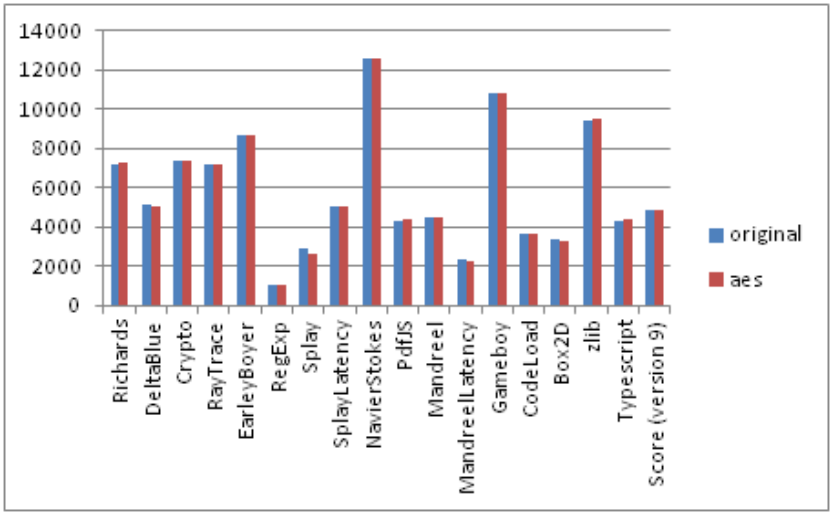


Fig. 6 Effect of Algebraic Expression Simplification

6.2 Common Subexpression Elimination

V8 engine already has implemented Global Value Numbering optimization which eliminates redundant code. However there are related but not identical optimizations such as Constant Propagation and Common Subexpression Elimination. For their differences see [14].

Because V8 already has some kind of Constant Propagation we decided to implement Global Common Subexpression Elimination in SSA form.

We have found that running this optimization before and after Global Value Numbering gives net effect about 2% performance improvement.

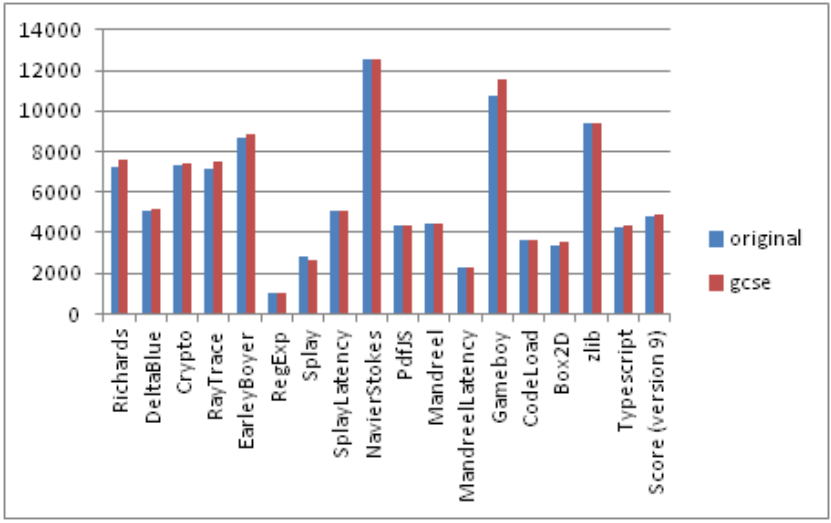


Fig. 7 Effect of Global Common Subexpression Elimination

6.3 Fast call frame for ARM

In our investigations we also have found interesting instruction sequence that speeds up call frame management on original ARMv7 CPUs.

To support EABI [15] compiler typically generate the following prologue and epilogue in each function.

Prologue:

```
func:
    stmdb    sp!, {r4-r5, fp, lr}
    add      fp, sp, #N
```

Epilogue:

```
    mov      sp, fp
    ldmia    sp!, {r4-r5, fp, lr}
    bx       lr
```

We have found however that the following sequences of instruction while provide the same functionality are executed faster on ARMv7 CPUs:

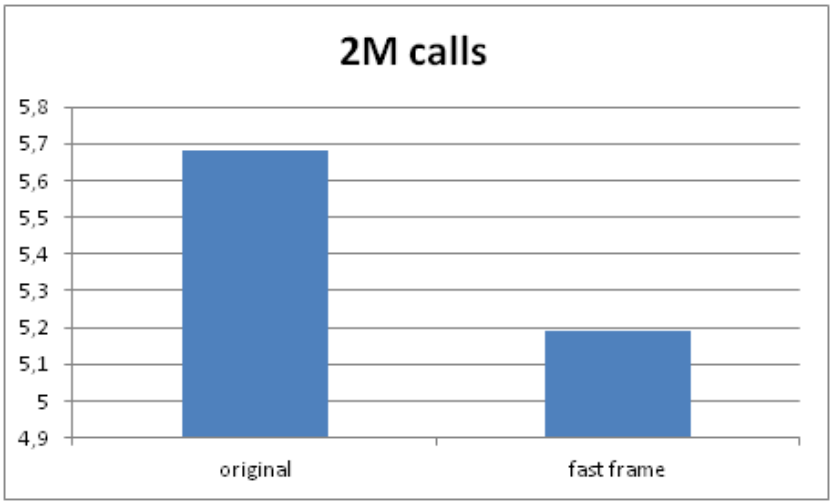
Prologue:

```
func:
    sub sp, sp, #16
    stm sp, {r4,r5,fp, lr}
    add      fp, sp, #N
```

Epilogue:

```
mov       sp, fp
ldm sp, {r4, r5, fp, lr}
add sp, sp, #16
bx        lr
```

The results on synthetic benchmark (~2 million calls, sec):



It is interesting however, that such results are not reproduced on Qualcomm Snapdragon 800 CPU.

7. Conclusion

We have found that even in the presence of type information in V8 optimizing compiler application of traditional scalar optimizations in JavaScript gives diminishing returns.

On the other hand successful application of optimized build gives us evidence that there is a space for optimizations in JavaScript engines.

References

[1]. TIOBE Index for October 2015 (http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html).
[2]. Chrome V8, September 10, 2015 (https://developers.google.com/v8/?hl=en)
[3]. Just-in-time compilation, Wikipedia, October 17, 2015 (https://en.wikipedia.org/wiki/Just-in-time_compilation)
[4]. Hölzle U., Chambers C., Ungar D. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, ECOOP ‘91 proceedings, Springer Verlag Lecture Notes in Computer Science 512, July, 1991

[5]. Wingo A., On-stack replacement in V8, June 20, 2011 (https://wingolog.org/archives/2011/06/20/on-stack-replacement-in-v8)
[6]. Octane 2.0 (https://chromium.github.io/octane)
[7]. SunSpider 1.0.2 JavaScript Benchmark (https://www.webkit.org/perf/sunspider/sunspider.html)
[8]. Kraken JavaScript Benchmark (version 1.1) (http://krakenbenchmark.mozilla.org)
[9]. perf (Linux), Wikipedia, (https://en.wikipedia.org/wiki/Perf_%28Linux%29)
[10]. Streamline Performance Analyzer, (http://ds.arm.com/ds-5/optimize)
[11]. Gprof, Wikipedia, (http://en.wikipedia.org/wiki/Gprof)
[12]. Interprocedural optimization, Wikipedia (https://en.wikipedia.org/wiki/Interprocedural_optimization)
[13]. GCC ARM options (https://gcc.gnu.org/onlinedocs/gcc-4.8.5/gcc/ARM-Options.html#ARM-Options)
[14]. Muchnik S., Advanced Compiler Design and Implementation, Morgan Kauffmann Publishers, San Francisco, USA, 1997, 856p
[15]. Application Binary Interface for the ARM Architecture v2.09 (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0036b/index.html)

Подходы к оптимизации движка JavaScript V8

Дмитрий Бочарников <dmitry.b@samsung.com>
Московский исследовательский центр Самсунг,
Москва, ул. Двинцев, 12, корп. 1

Аннотация. JavaScript является одним из наиболее распространенных языков программирования. Однако производительность движков JavaScript не всегда удовлетворительна. Автором разработаны подходы, позволяющие повысить производительность движка V8 на 10% на основных тестовых наборах.

Ключевые слова: JavaScript, оптимизации, V8, исключение общих подвыражений

DOI: 10.15514/ISPRAS-2015-27(6)-2

Для цитирования: Бочарников Дмитрий. Подходы к оптимизации движка JavaScript V8. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 21-32. DOI: 10.15514/ISPRAS-2015-27(6)-2.

Литература

[1]. TIOBE Index for October 2015 (http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html).
[2]. Chrome V8, September 10, 2015 (https://developers.google.com/v8/?hl=en)
[3]. Just-in-time compilation, Wikipedia, October 17, 2015 (https://en.wikipedia.org/wiki/Just-in-time_compilation)

[4]. Hölzle U., Chambers C., Ungar D. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, ECOOP '91 proceedings, Springer Verlag Lecture Notes in Computer Science 512, July, 1991
[5]. Wingo A., On-stack replacement in V8, June 20, 2011 (https://wingolog.org/archives/2011/06/20/on-stack-replacement-in-v8)
[6]. Octane 2.0 (https://chromium.github.io/octane)
[7]. SunSpider 1.0.2 JavaScript Benchmark (https://www.webkit.org/perf/sunspider/sunspider.html)
[8]. Kraken JavaScript Benchmark (version 1.1) (http://krakenbenchmark.mozilla.org)
[9]. perf (Linux), Wikipedia, (https://en.wikipedia.org/wiki/Perf_%28Linux%29)
[10]. Streamline Performance Analyzer, (http://ds.arm.com/ds-5/optimize)
[11]. Gprof, Wikipedia, (http://en.wikipedia.org/wiki/Gprof)
[12]. Interprocedural optimization, Wikipedia (https://en.wikipedia.org/wiki/Interprocedural_optimization)
[13]. GCC ARM options (https://gcc.gnu.org/onlinedocs/gcc-4.8.5/gcc/ARM-Options.html#ARM-Options)
[14]. Muchnik S., Advanced Compiler Design and Implementation, Morgan Kauffmann Publishers, San Francisco, USA, 1997, 856p
[15]. Application Binary Interface for the ARM Architecture v2.09 (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0036b/index.html)

Динамическая компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM

[1]*В.Г. Варданян <vaag@ispras.ru>*
[1]*В.А. Иванишин <vlad@ispras.ru>*
[2]*С.А. Асрян <seryozha.asryan@ysumail.am>*
[2]*А.А. Хачатрян <aramayis.khachatryan@ysumail.am>*
[2]*Дж.А. Акопян <dzhivan.hakobyan1@ysumail.am>*
[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2]*Ереванский государственный университет,*
0025, Армения, г. Ереван, ул. А. Манукяна, дом 1

Аннотация. В статье предлагаются методы, делающие возможной компиляцию программ на языке JavaScript в статически типизированное представление LLVM. В работе рассматривается многоуровневый динамический компилятор языка JavaScript V8, разработанный компанией Google. Основная цель работы – улучшение производительности программ на языке JavaScript. Для этого предлагается способ добавления в компилятор V8 нового уровня оптимизации, который использует инфраструктуру LLVM для генерации машинного кода. Это позволяет применять имеющиеся в LLVM оптимизации и технологии генерации машинного кода для разных архитектур к программам, написанным на JavaScript.

Ключевые слова: JavaScript, V8, LLVM, оптимизация программ, динамическая компиляция.

DOI: 10.15514/ISPRAS-2015-27(6)-3

Для цитирования: Варданян В.Г., Иванишин В.А., Асрян С.А.,. Хачатрян А.А, Акопян Дж.А. Динамическая компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 33-48. DOI: 10.15514/ISPRAS-2015-27(6)-3.

1. Введение

В настоящее время широкое распространение получили программы на нетипизированных сценарных языках. Одним из повсеместно используемых

языков является JavaScript. С использованием JavaScript написаны многие крупные многофункциональные приложения, такие как Gmail, Google docs и другие. JavaScript также используется в платформе Node.js [1] для разработки веб-приложений на стороне сервера. Более того, уже имеются разработки операционных систем для телефонов, планшетов и ноутбуков, которые подразумевают использование JavaScript как одного из основных языков для создания приложений. Примерами таких систем могут быть Tizen [2] и FirefoxOS [3]. Тем самым все больше возрастают требования к производительности программ на языке JavaScript, а также к паузам при интерактивном взаимодействии. Для обеспечения быстрого выполнения скриптов необходимо создание максимально качественного машинного кода, а также снижение затрат на все этапы оптимизации, выполняемые при компиляции программ. Для достижения этой цели многие современные JIT-компиляторы поддерживают разные уровни компиляции горячих участков кода. Целью данной работы является улучшение производительности программ на языке JavaScript методом добавления нового уровня оптимизации с использованием инфраструктуры LLVM [4] в компиляторе V8.

Дальнейшее изложение построено следующим образом. В главе 2 описана архитектура компилятора V8 и примененные технологии (замена на стеке, параллельное выполнение и т.д.) для построения многоуровневых JIT-компиляторов. В главе 3 дается обзор работ в предметной области. В главе 4 описывается схема функционирования предлагаемого решения, обеспечивающего использование биткода LLVM в качестве новой платформы для JIT-компиляции. В главе 5 приведены основные результаты.

2. Архитектура V8

Для генерации машинного кода V8 использует два разных компилятора[1] (Рис. 1). Единицей компиляции является функция (метод). Первыми этапами работы V8 являются лексический и синтаксический анализ. Исходный код разбивается на лексемы, методом рекурсивного спуска строится синтаксическое дерево. После этого начинает работать компилятор первого уровня Full-Codegen. На первом уровне функция переводится в машинный код с выполнением минимального набора оптимизаций, что позволяет быстрее приступить к выполнению кода. При генерации машинного кода для каждой инструкции учитываются все возможные случаи выполнения для данной операции. На этом уровне собирается профиль программы – информация о типах полей объектов. Кроме того, базовый компилятор V8 расставляет счетчики для определения горячих участков кода (функций и циклов). Когда такой участок кода обнаруживается, компиляция переходит на второй,

[1] На самом деле, разработчиками компилятора V8 активно разрабатывается и другой уровень оптимизации, который использует промежуточное представление “sea of nodes”, но в данной работе этот уровень не представляет интереса.

оптимизирующий уровень – Crankshaft. На этом уровне из абстрактного синтаксического дерева строится граф потока управления в SSA-представлении – Hydrogen. Это внутреннее представление позволяет выполнить ряд машинно-независимых оптимизаций, таких как встраивание функций, удаление мертвого кода, удаление общих подвыражений, оптимизации циклов и т.д.

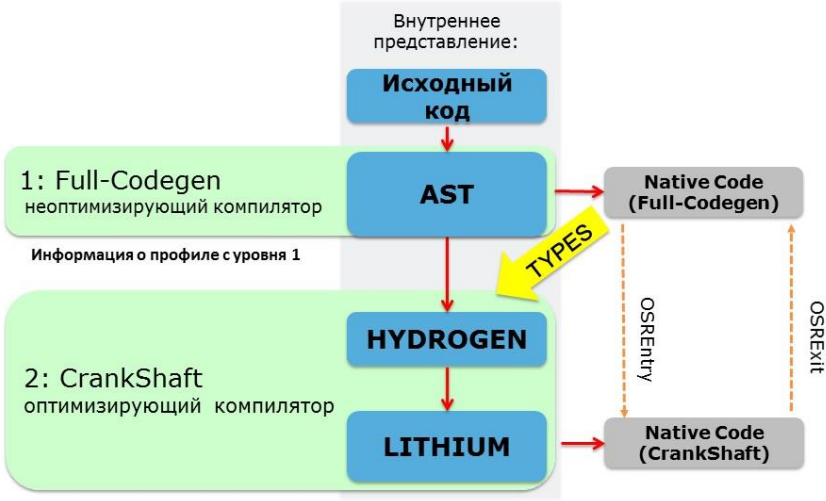


Рис. 1. Многоуровневая архитектура компилятора V8

Согласно спецификации EcmaScript [5], все числа в JavaScript являются вещественными числами двойной точности, удовлетворяющими стандарту IEEE 754. Однако Crankshaft может оптимизировать генерируемый машинный код (используя целочисленность переменных для более оптимального хранения и операций над ними) при помощи собранной на нижнем уровне информации о типах. На 64-битных архитектурах для целых чисел используется следующее кодирование: старшие 32 бита хранят число, младшие биты нулевые. Целые числа в таком представлении называются Smi (от английского small integer). В JavaScript отсутствует строгая типизация. Это значит, что в общем случае переменные ссылаются на объекты. Все указатели на объекты в V8 имеют младший бит (определяющий четность числа) установленным в единицу, что позволяет отличить Smi от других объектов. Кодирование использует тот факт, что все адреса выровнены, т.е. объектов с нечетными адресами не существует.

Crankshaft распространяет полученную из уровня Full-Codegen информацию о профиле по всему графу. Это позволяет генерировать машинный код спекулятивным образом, основываясь на предположении, что определенные свойства переменных (тип, значение и т.д.) останутся неизменными при следующих вызовах функции. Для этого в код вставляются необходимые проверки. Когда одна из таких проверок не выполняется, происходит переход на первый, неоптимизированный уровень. Для переключения между разными уровнями компиляции используется технология замены на стеке. При этом, как описано выше, переключение может произойти как с первого уровня на второй (OSR entry), так и наоборот (OSR exit, deoptimization).

Помимо технологии замены на стеке (OSR), для некоторых виртуальных машин [6], предусматривающих JIT-компиляцию, характерна еще одна особенность. В ряде случаев возникает необходимость как можно скорее остановить потоки выполнения скомпилированного кода[2]. Наиболее ярким и общим из таких случаев является остановка по причине начала работы сборщика мусора. Многие сборщики мусора не предусматривают параллельной работы вместе с выполнением программы, т.к. это приводило бы к конкуренции потоков за доступ к одним и тем же объектам и порождало бы недетерминированные паузы, кроме того, сборщик мусора может переносить объекты, что приводит к тому, что сам код, ссылающийся на них, должен быть изменен. Помимо приостановки выполнения, необходимо также гарантировать, что все объекты во время паузы будут находиться в целостном, согласованном состоянии.

Для достижения этой цели компилятор V8 использует технологию safepoints. Термин *safepoint* означает согласованное (и известное) состояние программы, а также точку в программе, в которой такое состояние достигается. Для того, чтобы исполняемый код мог быстро достичь safepoint, они, как правило, расставляются на обратных ребрах циклов и местах вызовов функций. В этих точках выставляются проверки, в случае срабатывания которых выполнение остановится в текущем состоянии.

После выполнения всех оптимизаций представление Hydrogen переводится в машинно-зависимое представление – Lithium. В отличие от представления Hydrogen, Lithium использует близкое к машинному коду трехадресное представление. Это представление используется для эффективной организации распределения регистров, а также для кодогенерации.

3. Обзор работ

С ростом популярности языка JavaScript, в течение последних лет несколько крупных корпораций, таких как Google, Mozilla и Apple, выпустили свои JIT-компиляторы для языка JavaScript. Разработанный компанией Apple компилятор под названием JavaScriptCore [7] также имеет многоуровневую

[2] Для JavaScript, согласно стандарту языка, может быть только один такой поток.

структуру (Рис. 2.). В отличие от V8, в JavaScriptCore реализованы четыре уровня компиляции. На первом уровне работает интерпретатор LLINT. Второй уровень представляет из себя простой JIT-компилятор Baseline JIT. На этом уровне для функций генерируется машинный код с минимальными оптимизациями аналогично уровню Full-Codegen в компиляторе V8. Baseline JIT создает для каждой операции байткода соответствующий машинный код. В этом коде реализуются все возможные случаи для данной операции. Baseline JIT, как и Full-Codegen, используется в качестве базовой версии кода для функций, которые скомпилированы с помощью оптимизирующего JIT-компилятора. Если оптимизированный код сталкивается со случаем, который в нем не поддерживается (например, тип или значение переменной не соответствует собранному профилю), то происходит обратная замена на стеке (OSR exit) и переход к коду Baseline JIT. Следующий уровень компилятора JSC представляет собой спекулятивный компилятор DFG.

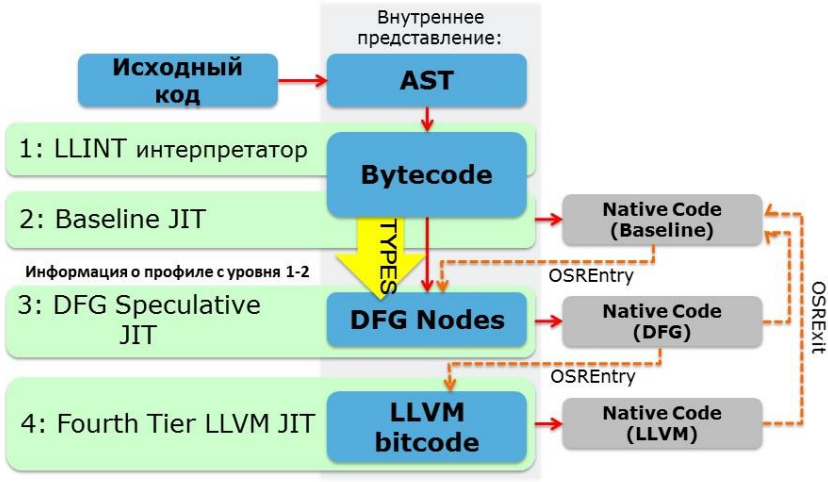


Рис. 2. Многоуровневая архитектура компилятора JavaScriptCore

На этом уровне из промежуточного представления байт-кода строится граф потока управления программы в виде SSA-формы. Это представление, аналогично уровню Crankshaft, делает возможным реализацию нескольких стандартных машинно-независимых оптимизаций, например, удаление общих подвыражений, удаление мертвого кода и т.д. DFG JIT код и Baseline JIT код могут сменять друг друга посредством замены на стеке (OSR). Когда код функции становится “горячим”, происходит переход на DFG JIT. Когда

выполняется деоптимизация, происходит обратный переход [8]. В мае 2014 года был добавлен новый, четвертый уровень компиляции – FTL JIT, который использует инфраструктуру LLVM для генерации машинного кода. На момент начала данной работы FTL JIT поддерживался только для платформ Mac OS X и iOS, в данной же работе реализация выполняется в первую очередь для операционной системы GNU/Linux и процессоров x86-64. Стоит также отметить, что, несмотря на внешние сходства общей архитектуры, эти два компилятора отличаются по внутренним структурам и реализациям. В данной работе рассматриваются основные подходы, примененные нами при реализации нового уровня оптимизации с использованием инфраструктуры LLVM в динамическом компиляторе V8 языка JavaScript. Освещаются особенности архитектуры V8, существенные с точки зрения реализации нового уровня. Отметим также, что V8 существенно более популярен: JavaScriptCore используется только в веб браузере Safari, тогда как V8 встроен в браузеры Chrome, Opera, Android-браузер, Node.js и в операционную систему ChromeOS.

4. Компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM

Промежуточное представление оптимизирующего уровня Hydrogen содержит всю необходимую информацию для построения статически типизированного представления LLVM. Данный метод позволяет применять оптимизации, имеющиеся в статическом компиляторе LLVM, к программам, написанным на языке JavaScript. Инфраструктура LLVM предоставляет средства для динамической компиляции в виде модуля MCJIT. В этом модуле уже задействованы все имеющиеся механизмы LLVM для кодогенерации и машинно-зависимых оптимизаций под различные платформы. Компиляция языка JavaScript в промежуточное представление LLVM была реализована путем добавления дополнительного уровня в компилятор V8 (Рис. 3.).

Несмотря на то, что и Hydrogen, и LLVM-биткод используют форму SSA, есть определенные различия в интерпретации определения SSA-формы. Отличие состоит в том, что LLVM требует, чтобы базовые блоки входов Ф-функций являлись непосредственными предшественниками данного базового блока. На Рис. 4.а изображен граф потока управления, корректный с точки зрения Hydrogen и некорректный с точки зрения LLVM. На Рис. 4.б приведен тот же граф, скорректированный под требования LLVM. Первым этапом при добавлении нового уровня в V8 была разработка и реализация алгоритма для преобразования промежуточного представления Hydrogen в корректную с точки зрения LLVM SSA-форму. При этом, чтобы использовать код, сгенерированный LLVM, в той же манере, что и код, сгенерированный самим компилятором V8, необходимо, чтобы они были совместимы на бинарном уровне. В первую очередь, это означает, что в LLVM должна быть реализована генерация кода в соответствии с соглашением о вызовах,

принятом в V8 (V8 использует собственное соглашение о вызовах). Кроме того, стоит отметить, что для генерации машинного кода для некоторых инструкций необходимо вызывать вспомогательные функции. Эти вызовы, в свою очередь, реализуют собственные соглашения о вызовах. Для достижения совместимости сгенерированных кодов в компиляторе LLVM были реализованы все необходимые соглашения о вызовах.

Одним из важных компонентов для обеспечения быстродействия современных динамических компиляторов является поддержка спекулятивной компиляции.

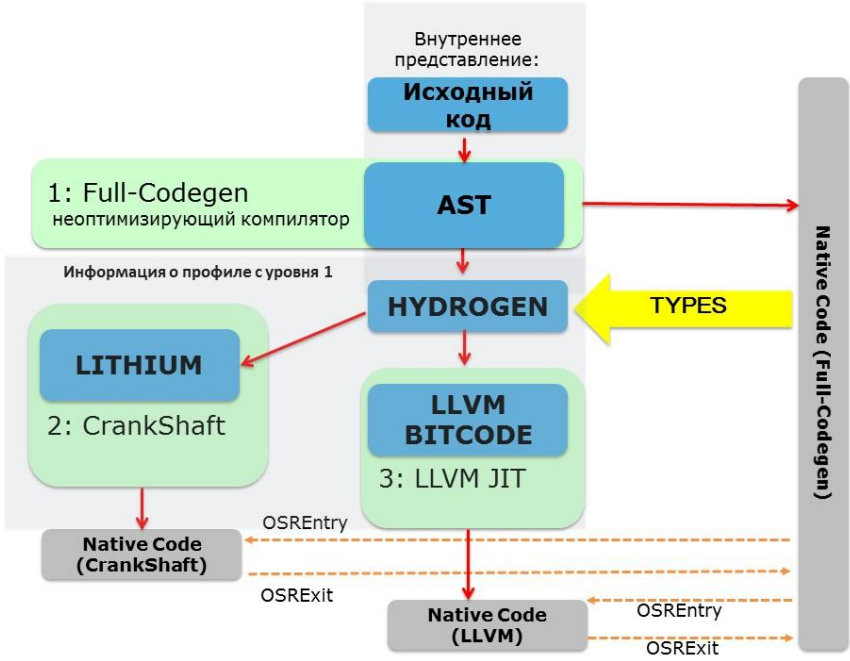


Рис. 3. Архитектура многоуровневого компилятора V8 с добавленным модулем LLVM

Эта технология используется, например, для частичной компиляции кода, что позволяет экономить используемую память, а также уменьшить общее время выполнения программы. Помимо этого, к спекулятивной компиляции можно отнести предсказание типов и технологию “полиморфный встроенный кэш вызовов” (англ. polymorphic inline cache) [9], применяемую для повышения производительности кода. Сложность реализации такой поддержки на уровне LLVM состоит в том, что, перекладывая генерацию кода на сторонний компилятор, мы теряем над ней контроль, что приводит к трудностям при реализации спекулятивной компиляции, а также при поддержке автоматической сборки мусора. Между тем, инфраструктура LLVM

предоставляет инструменты для контроля и модификации сгенерированного компонентом MCJIT кода. Эти инструменты были добавлены в LLVM в рамках реализации уровня FTL в компиляторе JSC. В итоге поддержка деоптимизаций (OSR exit) в новом уровне компилятора V8 была реализована с помощью встроенных в LLVM функций llvm.experimental.stackmap и llvm.experimental.patchpoint, которые позволяют модифицировать сгенерированный LLVM код на лету.

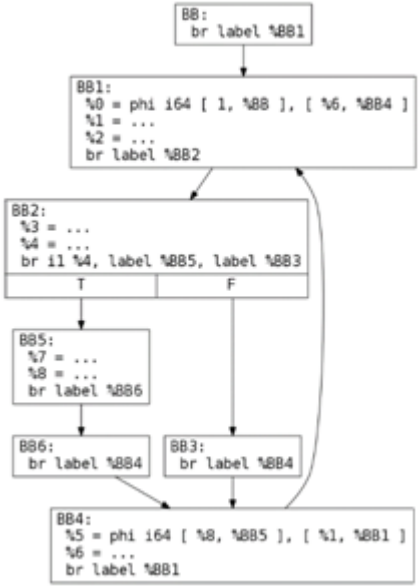


Рис 4.a Граф потока управления, некорректный с точки зрения LLVM

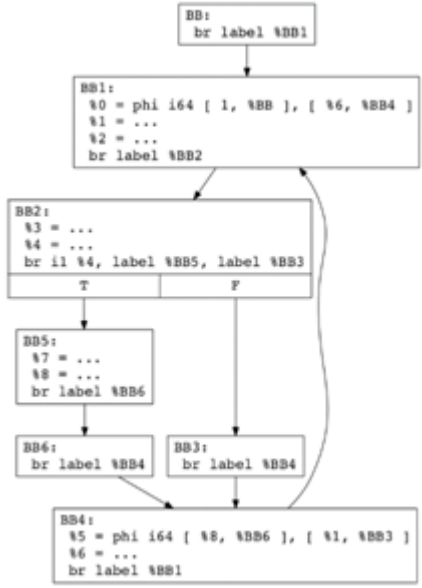


Рис 4.b Граф потока управления, корректный с точки зрения LLVM

Обе эти функции во время компиляции представления LLVM в машинный код инициируют создание специальной секции данных, содержащей структуру Stack Map [10]. В этой структуре сохраняется относительное смещение от начала функции в машинном коде, куда попадает вызов stackmap/patchpoint, а также местонахождение (слот стека, имя регистра, константа и т.п.) всех значений, переданных этим функциям в качестве параметров. Для поддержки деоптимизаций в месте, где происходит передача управления виртуальной машине, вставляется вызов stackmap. В качестве аргументов передаются значения, необходимые для продолжения выполнения на уровне Full-Codegen. Вычисление этих значений происходит с помощью уже имеющегося в Crankshaft механизма симуляции состояния стековой машины (которую реализует Full-Codegen).

Отдельное внимание следует уделить реализации механизма safepoint (см. главу 2). По сути, реализация safepoint во многом аналогична реализации

деоптимизации с помощью llvm.experimental.stackmap. Однако определение значений, информация о местоположении которых после генерации кода для нас важна, отличается. Для точки safepoint это все живые в этой точке значения, являющиеся указателями на объекты в динамической памяти. Когда сборщик мусора V8 совершает обход указателей, он учитывает объекты, ассоциированные с safepoint, на котором приостановлено выполнение, как живые. Без правильно сформированных сущностей safepoint, занимаемая этими объектами память могла бы быть освобождена, что привело бы к некорректному поведению. Для получения информации об интервалах жизни переменных (LLVM-значений), накрывающих определенную точку, были задействованы анализирующие проходы, уже реализованные в LLVM [11]. Кроме того, соответствующие преобразующие проходы LLVM были использованы для автоматической расстановки safepoints. А для генерации Stack Map для точек safepoint применена функция llvm.experimental.gc.statepoint [12].

Функции llvm.experimental.stackmap/patchpoint также используются для поддержки возможности перемещения объектов сборщиком мусора V8. Сгенерированный код может содержать абсолютные адреса объектов. При перемещении объектов необходимо заменить их старые адреса на новые. Кроме того, сгенерированный машинный код находится в куче и интерпретируется компилятором V8 как обыкновенный объект. Следовательно, сборщик мусора может передвигать сгенерированный код в процессе своей работы. Этот перемещаемый код вызывается по смещению из генерируемого LLVM кода. Для реализации такой поддержки используется функция llvm.experimental.patchpoint, которая помимо тех же параметров, что и llvm.experimental.stackmap, принимает также вызываемую функцию. Помимо создания Stack Map, при компиляции llvm.experimental.patchpoint в генерируемый код вставляется вызов этой функции в соответствии с заданным соглашением о вызовах. Благодаря Stack Map, позже сам вызываемый объект можно будет подменить.

Другим важным аспектом при проектировании многоуровневых JIT-компиляторов является поддержка замены исполняемого кода на оптимизированный код для горячих функций. В простом случае, когда функция имеет короткое время выполнения, переход осуществляется путем замены ее адреса на адрес оптимизированного кода во время следующих вызовов функции. (Это возможно, так как сгенерированные коды реализуют одинаковый бинарный интерфейс.) Однако в случае, если функция содержит цикл или несколько циклов с большим количеством итераций, появляется необходимость в организации перехода на оптимизирующий уровень во время выполнения функции. Такой переход между уровнями Full-Codegen и Crankshaft осуществляется путем простого безусловного перехода на начало соответствующего цикла в оптимизированном коде (Рис. 5). Так как управление передается не на начало функции, при переходе на новый код

также выполняются определенные действия для приспособления стека. Например, необходимо уменьшить значение указателя стека на количество локальных переменных, которые будут храниться в стеке в оптимизированном коде. При этом необходимо учитывать, что прежний уровень, в свою очередь, уже уменьшил это значение на количество своих локальных переменных. Основная проблема такого подхода при переходе на LLVM код заключается в том, что LLVM подразумевает, что каждая функция может иметь только одну точку входа.

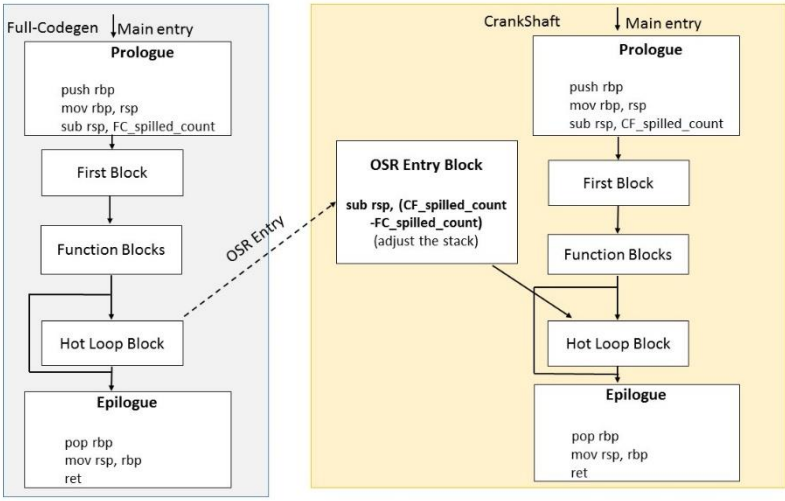


Рис. 5. Схема перехода между компиляторами Full-Codegen и CrankShaft

Нарушение этого инварианта может привести к сильному ограничению выполняемых в LLVM оптимизаций или даже к неправильной работе программы. Одно решение этой проблемы, использующееся в компиляторе JSC – это создание копий функции для каждого возможного входа. При этом все аргументы и локальные переменные сохраняются в глобальном буфере, а переход на уровень LLVM осуществляется путем вызова соответствующей функции. Явным недостатком такого подхода является более длительное время компиляции из-за создания копий функции. Нами был разработан и реализован другой метод, который позволяет осуществить переход во время выполнения на LLVM-код без создания дополнительных копий функции. При реализации такого подхода основная проблема состоит в приспособлении стека, так как при использовании стороннего компилятора LLVM нет возможности выяснить, какое количество локальных переменных будет храниться на стеке (эта информация становится доступна гораздо позже, при распределении регистров LLVM). Для решения проблемы при переходе на

оптимизирующий код во время выполнения цикла точка входа была перемещена в пролог кода функции, где значение указателя стека уменьшается на количество локальных переменных, хранящихся в стеке. После этого с помощью дополнительного аргумента решается, откуда именно будет выполняться функция.

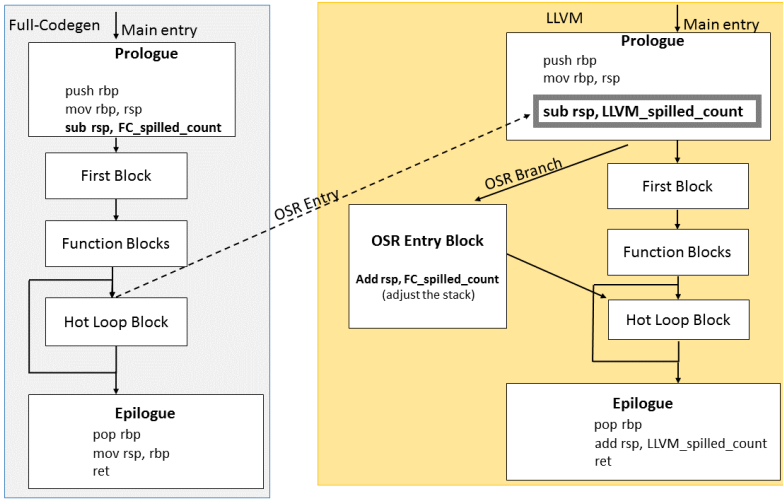


Рис. 6. Схема перехода между компиляторами Full-Codegen и LLVM

Если управление перешло на LLVM код из цикла, то необходимо выполнить дополнительное приспособление стека. В частности, надо учитывать тот факт, что прежний уровень мог, в свою очередь, уменьшить значение указателя стека (Рис. 6.). При таком подходе LLVM функция также может содержать две точки входа, но, в отличие от уровня Crankshaft, эти точки находятся в прологе функции и не являются препятствием для реализации оптимизаций в LLVM. Однако при реализации такого подхода появляются определенные проблемы при передаче локальных значений из уровня Full-Codegen уровню LLVM. Эти значения хранятся в начале стека, начиная с адреса, записанного в специальный регистр rbp (stack base pointer), и загружаются в регистры после соответствующей проверки для нахождения точки входа функции (Рис. 6.). Однако модуль распределения регистров LLVM не подразумевает наличия зарезервированных слотов стека. Таким образом, LLVM, в свою очередь, может использовать адреса начиная с регистра rbp для распределения своих локальных переменных. В итоге может получиться, что локальные переменные Full-Codegen могут быть переписаны компилятором LLVM. Одним решением этой проблемы может быть внесение изменений в модуль распределения регистров LLVM для резервирования адресов в стеке. Однако

подобные глобальные изменения в коде LLVM не универсальны и осложняют применение обновлений к кодовой базе из основного репозитория. По этой причине был избран другой подход. Желаемый эффект был достигнут посредством добавления фиктивных volatile значений. Так, при генерации биткода LLVM с помощью volatile значений были зарезервированы соответствующие слоты стека для локальных значений Full-Codegen.

Еще одним важным аспектом при компиляции программ на языке JavaScript является снижение пауз при интерактивном взаимодействии. Для достижения этой цели в многоуровневых компиляторах реализована поддержка параллельного выполнения функций. Так, например, код, сгенерированный уровнем Full-Codegen, будет выполняться параллельно, пока на уровне Crankshaft выполняются оптимизации и кодогенерация. Поддержка параллелизма на уровне LLVM полностью осуществлена и в нашем решении.

После осуществления поддержки всех технологий, обеспечивающих быстродействие современных многоуровневых JIT-компиляторов (спекулятивная компиляция, технология “замена на стеке”, параллельное выполнение и т. д.) следующим шагом была трансляция представления Hydrogen в промежуточное представление LLVM. При реализации преобразования Hydrogen в биткод LLVM был применен инкрементальный подход: количество доступных операций наращивалось постепенно, делая возможным компиляцию все более широких подмножеств языка JavaScript. В представлении Hydrogen насчитывается около 120 различных операций, на данный момент реализована трансляция более 70 вершин в промежуточное представление LLVM.

5. Результаты

На данный момент компиляция в LLVM биткод поддерживается не для всех структур языка JavaScript. Это накладывает ограничение на полное тестирование бенчмарков языка JavaScript. По этой причине в данном разделе приведены результаты тестирования на нескольких тестах из набора SunSpider. Оценка производительности для этих тестов была проведена для разного количества итераций, что позволяет оценивать реальную пользу компиляции “очень горячих” участков кода с помощью инфраструктуры LLVM.

Табл. 1. Сравнение производительности на тесте access-nsieve

access-nsieve			
Число итераций	**ориг. кол-во итераций**	***10**	***100**
Время выполнения с исп. Crankshaft, мс	**59**	**590**	**2980**
Время выполнения с исп. LLVM, мс	**63**	**580**	**2465**
Ускорение, число раз	**0.94**	**1.02**	**1.21**

Табл. 2. Сравнение производительности на тесте bitops-bits-in-byte

bitops-bits-in-byte			
Число итераций	ориг. кол-во итераций	*10	*100
Время выполнения с исп. Crankshaft, мс	37	216	2050
Время выполнения с исп. LLVM, мс	27	74	538
Ускорение, число раз	1.37	2.92	3.81

Табл. 3. Сравнение производительности на тесте 3d-cube

3d-cube			
Число итераций	ориг. кол-во итераций	*5	*10
Время выполнения с исп. Crankshaft, мс	100	391	842
Время выполнения с исп. LLVM, мс	85	177	325
Ускорение, число раз	1.18	2.21	2.59

Как видно из представленных выше таблиц, при увеличении числа итераций (времени выполнения машинного кода) увеличивается и выигрыш в производительности, обеспечиваемый более оптимальным машинным кодом уровня LLV8 в сравнении с Crankshaft. Для ответа на вопрос, чем обоснована более высокая производительность уровня LLV8, нами был проведен сравнительный анализ машинных кодов. Анализ теста bitops-bits-in-byte, например, показывает, что ускорение достигается в первую очередь благодаря наличию оптимизации “разворачивание цикла” (англ. loop unrolling) в списке оптимизационных проходов, выполняемых над внутренним представлением LLVM (в Crankshaft такая оптимизация отсутствует). Код, сгенерированный LLV8 для данного теста, не содержит цикла, тогда как код, сгенерированный Crankshaft, имеет цикл. Отметим также, что, время компиляции на уровне LLVM почти всегда больше чем на уровне Crankshaft. Например, для теста bitop-bits-in-byte, время, затраченное на компиляцию, составило примерно 0,2 мс в случае Crankshaft и 6,5 мс в случае LLVM. Однако даже такое увеличение времени компиляции приемлемо для функций, время выполнения которых существенно превосходит время компиляции. (Кроме того, при компиляции в параллельном потоке это имеет еще меньшее значение.) Именно для оптимизации таких функций изначально было принято решение реализации уровня LLVM в компиляторе V8.

6. Заключение и будущая работа

В рамках данной работы был разработан метод динамической компиляции программ на языке JavaScript в статически типизированное представление LLVM. Предложенный метод был реализован путем добавления дополнительного уровня оптимизации в компиляторе с открытым исходным кодом V8. При этом была реализована поддержка всех технологий,

обеспечивающих быстродействие современных многоуровневых JIT-компиляторов (спекулятивная компиляция, технология “замена на стеке”, параллельное выполнение и т. д.). В дальнейшем планируется добавление поддержки компиляции в LLVM биткод для всех конструкций языка JavaScript и тестирование системы на популярных тестовых наборах.

Список литературы

[1]. Страница платформы Node.js — https://nodejs.org
[2]. Страница платформы Tizen — http://www.tizen.org
[3]. Веб-сайт Mozilla — https://www.mozilla.org
[4]. Страница платформы LLVM – http://www.llvm.org/
[5]. Описание стандарта ECMA-262 http://www.ecma-international.org/publications/standards/Ecma-262.html
[6]. Страница документации динамического компилятора HotSpot для языка Java – http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html
[7]. Веб-сайт Webkit – http://www.webkit.org
[8]. Р. Жуйков, Д. Мельник, Р. Бучацкий, В. Варданян, В. Иванишин, Е. Шарыгин Методы динамической и предварительной оптимизации программ на языке JavaScript. Труды Института системного программирования РАН, Том 26. Выпуск 1. 2014 г. Стр. 297- 314. DOI: 10.15514/ISPRAS-2014-26(1)-10
[9]. U. Hölzle, C. Chambers, D. Ungar “Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches” ECOOP '91 Proceedings of the European Conference on Object-Oriented Programming, 21-38, 1991
[10]. Страница документации структуры StackMaps – http://llvm.org/docs/StackMaps.html
[11]. Страница обзора изменений LLVM в ревизии rL229945 – http://reviews.llvm.org/rL229945
[12]. Страница документации структуры StatePoints – http://llvm.org/docs/Statepoints.html

Dynamic Compilation of JavaScript Programs to the Statically Typed LLVM Intermediate Representation

[1]V. Vardanyan <vaag@ispras.ru>
[1]V. Ivanishin <vlad@ispras.ru>
[2]S. Asryan <seryozha.asryan@ysumail.am>
[2]A. Khachatryan <aramayis.khachatryan@ysumail.am>
[2]J. Hakobyan <dzhivan.hakobyan1@ysumail.am>
[1]I Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] Yerevan State University, 1 Alex Manoogian Str., Yerevan, 0025, Republic of Armenia

Abstract. Since its inception in the middle of the 90's, JavaScript has become one of the most popular web development languages. Although initially developed as a browser-agnostic scripting language, in recent years JavaScript continues its evolution beyond the desktop to areas such as mobile and server-side web applications. Many massive applications are written using JavaScript, such as gmail, google docs, etc. JavaScript is also used in the Node.js — server-side web application developing platform. Moreover, JavaScript is the main language for developing applications on some operating systems for mobile and media devices. Examples of such systems are Tizen and FirefoxOS. Modern JavaScript engines use just-in-time (JIT) compilation to produce binary code. JIT compilers are limited in complexity of optimizations they can perform at runtime without delaying the execution. To maintain a trade-off between quick startup and doing sophisticated optimizations, JavaScript engines usually use multiple tiers for compiling hot functions. Our work is dedicated to performance improvement of JavaScript programs by adding a new optimizing level to the JavaScript V8 compiler. This level uses the LLVM infrastructure to optimize JavaScript functions and generate machine code. The main challenge of adding a new optimizing level is to support all the technologies (speculative compilation, on-stack replacement, concurrent compilation etc.) that are used in the modern multi-tier JIT compilers for increasing the performance and minimizing pauses during the interactive communication. All these technologies are fully supported in our solution. This has resulted in significant performance gains on the JavaScript benchmark suites when compiling hot functions.

Keywords: JavaScript, V8, LLVM, program optimization, dynamic compilation.

DOI: 10.15514/ISPRAS-2015-27(6)-3

For citation: Vardanyan V., Ivanishin V., Asryan S., Khachatryan A., Hakobyan J. Dynamic Compilation of JavaScript Programs to the Statically Typed LLVM Intermediate Representation. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 33-48 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-3

References

[1]. Node.js website – https://nodejs.org
[2]. Tizen platform website – http://www.tizen.org
[3]. Mozilla website – https://www.mozilla.org
[4]. LLVM website – http://www.llvm.org/
[5]. ECMA-262 Standard http://www.ecma-international.org/publications/standards/Ecma-262.htm
[6]. HotSpot dynamic compiler documentation website – http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html
[7]. Webkit website – http://www.webkit.org
[8]. R. Zhuykov, D. Melnik, R. Buchatskiy, V. Vardanyan, V. Ivanishin, E. Sharygin. Metody dinamicheskoj i predvaritel'noj optimizacii programm na jazyke JavaScript. [Dynamic and ahead of time optimization for JavaScript programs] Trudy ISP RAN [The Proceedings of ISP RAS], Volume 26 (Issue 1). 2014, pp. 297-314. DOI: 10.15514/ISPRAS-2014-26(1)-10 (in Russian)
[9]. U. Hölzle, C. Chambers, D. Ungar "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches" ECOOP '91 Proceedings of the European Conference on Object-Oriented Programming, 21-38, 1991
[10]. StackMaps structure documentation webpage — http://llvm.org/docs/StackMaps.html

[11]. LLVM rL229945 revision review page — http://reviews.llvm.org/rL229945
[12]. StatePoints structure documentation webpage — http://llvm.org/docs/Statepoints.html

Методы коррекции профильной информации в процессе компиляции

О.А. Четверина <chetverina_o@mcst.ru>
АО "МЦСТ», 117105, Россия, г. Москва, ул. Нагатинская,дом 1.

Аннотация. Эффективность проводимых компилятором оптимизирующих преобразований может быть значительно повышена с помощью получения и использования профильной информации об исполнении программы, такую работу компилятора называют profile guided optimization (PGO). После проведения преобразований, изменяющих граф потока управления, необходимо скорректировать профильную информацию, с целью сохранения ее актуальности для качественной работы последующих оптимизаций. В работе рассматриваются два способа представления профильной информации и переход между ними, описаны возможные причины возникновения дополнительной информации о потоке управления в оптимизируемом коде, требующей изменения профиля. Рассмотрены наиболее часто возникающие задачи проведения коррекции профильной информации и предложены следующие решения: алгоритм коррекции значений счетчиков ациклического участка по значениям счетчиков узлов выхода; алгоритм коррекции значения среднего числа итераций цикла; алгоритм коррекции профиля при обнаружении «противоречивого узла». Доказано, что разработанные алгоритмы коррекции значений счетчиков ациклического участка и числа итераций цикла позволяют минимально изменить соотношение значений счетчиков исходного графа. На их основе описан механизм коррекции профильной и тонкой профильной информации после проведения преобразования откртки итераций цикла. Все предложенные методы реализованы и используются в процессе компиляции оптимизирующими компиляторами lcc, lccs, lfortran, lfortrans для архитектур Эльбрус и Спарк.

Ключевые слова: профиль исполнения задачи, PGO, коррекция профиля, коррекция числа итераций цикла, расширенная профильная информация, взвешенный граф

DOI: 10.15514/ISPRAS-2015-27(6)-4

Для цитирования: Четверина О.А. Методы коррекции профильной информации в процессе компиляции. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 49-68. DOI: 10.15514/ISPRAS-2015-27(6)-4.

1. Введение

Существенную роль в достижении максимальной производительности современных вычислительных систем играют оптимизирующие компиляторы, которые путем применения оптимизирующих преобразований кода позволяют

повысить эффективность использования вычислительных устройств. Для улучшения работы ряда преобразований разработаны механизмы, при которых компиляторам наряду с исходным кодом передается еще и дополнительная информация об исполнении программы. Один из наиболее используемых и значимых соответствующих механизмов это сбор и дальнейшее применение информации о частоте исполнения тех или иных событий [1,2,3]. Чаще всего в этом качестве используются значения счетчиков, фиксирующих число исполнения узлов и дуг (условных переходов) графа потока управления, в дальнейшем - *профильная информация*. По ним, для удобства расчета эвристик оптимизаций, вычисляются вероятности перехода от одного узла к другому, среднее число итераций цикла и другие характеристики взвешенного графа управления. Используя профильную информацию, компилятор может принимать более эффективные решения при применении таких преобразований, как предподкачка кода, программная векторизация [4], конвейеризация цикла [5], вынос инвариантных вычислений и других, что дает существенный прирост в производительности. Чаще всего компиляция с профилем производится для не очень больших пользовательских приложений, поскольку в этом случае достаточно легко произвести представительный тренировочный запуск. Но в последнее время все больше внимания уделяется возможности оптимизации с использованием профиля (PGO или FDO) и в более сложных ситуациях: на данный момент разрабатываются способы решения задачи профилирования операционных систем [6]; активно ведутся работы по использованию и улучшению качества профилирования в Just-in-Time компиляторах [7].

В то же время, несоответствие соотношений счетчиков линейных участков, используемых компилятором и возникающих при последующем исполнении программы, может приводить к ухудшению планирования кода при проведении тех же оптимизаций. Так, при неверном значении среднего числа итераций, компилятор может отказаться от программной конвейеризации цикла с большим реальным числом итераций, или, наоборот, конвейеризовать малоитерационный цикл с созданием большого числа стадий [8]. Если же распределение счетчиков ветвлений ациклических участков перестанет соответствовать реальному исполнению, то ряд преобразований может привести к перемешиванию маловероятных и высоковероятных операций с откладыванием времени исполнения последних, либо к отказу от применения некоторых оптимизаций к высоковероятным участкам [9]. Причина неточности используемой компилятором профильной информации может быть связана либо с исходной неполнотой собранной информации, либо с уменьшением ее достоверности в процессе компиляции. С целью уменьшения ошибок оптимизаций из-за неполноты собранной профильной информации может быть использовано смешанное профилирование, при котором полученные в процессе реальной работы программы данные дополняются статистически построенным, то есть предсказанным, профилем для ранее не

выполнявшихся участков кода [7]. Для повышения точности такой профиль может корректироваться с учетом профиля реального исполнения соседних процедур [10]. Со стороны же выполнения самой компиляции важной задачей является максимально точная коррекция профиля после применения изменяющих граф потока управления преобразований, то есть сохранение ее актуальности для последующих оптимизаций. Стоит отметить, что кроме возможности использовать более эффективные эвристики отдельными оптимизациями, сохранение достоверной информации о числе исполнения операций в процессе компиляции дает возможность предсказывать время работы и сравнивать качество получившегося кода в результате применения всей оптимизационной последовательности без реального запуска на машине [11,12].

В представленной работе описаны основные способы представления и коррекции профильной информации, а также дополнительные числовые характеристики, которые влияют на профиль и могут приводить к необходимости его коррекции. Поставлен ряд соответствующих задач на графах с весами и приведены разработанные способы их решения.

2. Способы представления профильной информации

Везде далее под профильной информацией подразумеваются счетчики числа исполнения узлов и дуг управляющего графа. Связано использование именно счетчиков с тем, что их проще всего собрать при исполнении с помощью инструментированного кода исходной программы. Зачастую более удачным вариантом представления этой информации внутри компилятора с точки зрения применения оптимизаций являются вероятности перехода по дугам. Удобство использования вероятностей заключается в возможности быстрой оценки и сравнения различных дуг без необходимости проводить нормировку.

Переход от счетчиков узлов и дуг к вероятностям и обратно является взаимно-однозначным с точностью до значения стартового счетчика рассматриваемой процедуры. При переходе от счетчиков к вероятностям, вероятности вычисляются как отношение счетчиков исходящих дуг к сумме значений счетчиков исходящих дуг в каждом узле. Обратный переход, то есть вычисление счетчиков по вероятностям, несколько сложнее — для ациклических (без контуров) связанных участков он производится пропагацией счетчиков по вероятностям при обходе сверху вниз с топологической сортировкой [13] для уменьшения количества вычислений:

1. Все стартовые узлы с известными значениями счетчиков обозначим *{Ni}*. Обозначим *{NOi}* - узлы с непроставленными значениями счетчиков.

2. Проставим каждому ребру, исходящему из каждого узла *{Ni}* счетчик, равный вероятности этого ребра на значение счетчика соответствующего узла *Ni.*

3. Если в *{NOi}* не осталось узлов, исполнение алгоритма закончено.

Иначе, проверяем $\{NOi\}$ на наличие таких узлов, где значения счетчиков для всех входящих дуг проставлены. Поскольку граф ациклический, то такие найдутся. Проставляем им в качестве значений счетчиков суммы счетчиков всех входящих дуг. Перекладываем их в $\{Ni\}$. Переходим в 2.

В случае же работы с циклами используется следующий алгоритм:

1. Строится дерево циклов, в котором у каждого цикла есть только одна голова [14].

2. В случае нескольких входов в цикл обходом графа цикла сверху вниз для каждого входа в цикл вычисляется счетчик, который пройдет от него до головы цикла через все обратные дуги. Полученные счетчики добавляются к счетчику головы цикла, в результате получается стартовый счетчик головы цикла C0.

3. Проходом сверху вниз от головы цикла вычисляется суммарная вероятность p пройти по обратным дугам цикла. Тогда среднее число итераций цикла:

$$I = 1 + p + p^2 + \cdots + p^n + \cdots = 1/(1-p)$$

4. Значение счетчика головы умножается на вычисленное число итераций и с помощью алгоритма для ациклического участка осуществляется пропагация с учетом значений счетчиков дополнительных входов.

При проведении значительной части преобразований, меняющих граф потока управления, для сохранения информации о профиле достаточно выставить требуемую вероятность на сформированные дуги и провести описанную пропагацию счетчиков по всему графу, либо просто провести коррекцию по уже имеющимся вероятностям. К примеру, при проведении подстановки процедур вместо вызова в точку вызова производится копирование управляющего графа вызываемой процедуры с коррекцией значений его счетчиков с учетом значения счетчика подставляемого вызова.

Но в ряде случаев в процессе проведения оптимизирующих преобразований выявляется необходимость согласовывать профильную информацию в виде счетчиков узлов и дуг или вероятностей переходов с дополнительными числовыми характеристиками, касающихся профиля, которые могут быть:

- заданы пользователем (pragma среднего числа итераций для цикла, likely/unlikely для ветвления),
- дополнительно собраны (расширенная профильная информация, такая, как вероятности выйти на определенных итерациях цикла),
- установлены компилятором эвристически при дублировании участков кода — ациклических или циклов, при построении новых ветвлений,
- вычислены точно по имеющимся операциям и константам.

Далее будут рассмотрены и предложены решения наиболее часто встречающихся задач согласования различных профильных характеристик и предложены способы коррекции профиля: коррекция значений счетчиков ациклического участка по значениям счетчиков узлов выхода (глава 3), коррекция числа итераций цикла (глава 4), коррекция профиля при обнаружении «противоречивого перехода» (глава 5), вычисление числа итераций и коррекция профильной и тонкой профильной информации после проведения откртки итераций (глава 6).

3. Алгоритм коррекции счетчиков ациклического участка по заданному входному счетчику и счетчикам выходов

Решение описанной в этой главе задачи наиболее востребовано для представленной далее коррекции числа итераций цикла, но используется и для ациклических участков при коррекции счетчиков откручиваемых итераций цикла или необходимости выставить определенную суммарную вероятность перехода на некоторый участок из стартового узла при создании нескольких дуг с неопределенными значениями вероятностей.

Задача: Рассмотрим ориентированный согласованный односвязный взвешенный ациклический граф с одним доминирующим и n концевыми узлами. Обозначим вес доминирующего узла *C0*. Рассмотрим некоторый набор новых вещественных значений *C1, C2,...Cn* таких, что *C1+C2+...+Cn=C0*. Требуется скорректировать веса узлов таким образом, чтобы в концевых узлах *N1, N2, ... Nn* стояли значения *C1, ...Cn,* соответственно. При этом нужно минимально изменить соотношение счетчиков внутри участка. Под изменением счетчиков будем подразумевать максимум отношений старых и новых счетчиков цикла всех узлов рассматриваемого графа.

Для решения задачи предлагается использовать пропагацию счетчиков снизу вверх в пропорции значении счетчиков входящих дуг в порядке обратной топологической сортировки.

Алгоритм:

1. Пометим все узлы графа как узлы с непроставленными весами. Поставим в узлы *N1, N2, ... Nn* счетчики *C1, C2,...Cn* соответственно. Набор узлов с уже проставленными в них счетчиками обозначим *{Ni}*, соответствующий набор счетчиков - *{Ci}*. Для каждого узла графа запоминаем число исходящих дуг с непроставленными счетчиками *{EOi}*.

2. Если в *{Ni}* находится только доминирующий узел участка, то работа алгоритма закончена.

3. Для всех узлов *Nk* из *{Ni}* производим следующее:

Пусть в узел *Nk* входит *m* ребер, его новый вес равен Ck, а старые веса ребер, входящих в *Nk,* обозначаются *{EC(i)} i=1,.. m.*

Присвоим каждому входящему в *Nk* ребру *j* вес *Ck*(EC(j)/ECsum),* где $ECsum=\sum EC(i)$- сумма всех старых весов ребер, входящих в узел *Nk.*

При этом из значения *EOi* для узлов, из которых выходят соответствующие ребра, вычитаем 1 для каждого ребра. Если в результате для каких-то узлов значение *EOi* стало равным 0, то добавляем их в *{Ni}.*

Исключаем узел *Nk* из *{Ni}.*

Идем в 2.

Сложность описанного алгоритма совпадает с минимальной сложностью алгоритма обратной топологической сортировки и равно *O(Edges*), где *Edges* — число ребер в цикле.

Обоснование:

1) Счетчик в доминирущем узле равен сумме счетчиков концевых узлов из-за сохранения суммы потоков [15].

2) Покажем, что приведенный алгоритм удовлетворяет условию минимального изменения счетчиков.

Во-первых, заметим, что при проведении изложенной коррекции на ациклическом участке с уже проставленными счетчиками и вероятностями и использовании исходных концевых значений будет получен граф, совпадающий с исходным, поскольку в этом случае каждый шаг будет приводить к идентичным счетчикам узлов и дуг.

Далее, рассмотрим представление исходного взвешенного графа со счетчиками в виде суммы графов, каждый из которых представляет собой проведенную описанную выше пропагацию из одного из концевых участков для узлов, из которых достижим соответствующий концевой. Счетчики указанной суммы будут совпадать со значениями исходного графа, поскольку разделение счетчиков производится с одинаковыми коэффициентами для всех схождений. То есть значение счетчика каждого узла рассматриваемого графа *Nl* можно представить как сумму:

$$C(Nl) = \sum_{i \le n} p(i,l)Ci,$$

где *p(i,l)* — вероятность дойти от концевого узла *Nk* до узла *Nl* при проходе снизу вверх.

Пусть новые счетчики концевых узлов *Ck'=a(k)*Ck, a_min=min(a(k)), a_max=max(a(k).*

Тогда, аналогично предыдущему, новые счетчики внутренних узлов после описанной пропагации равны:

$$C'(Nl) = \sum_{i \le n} a(i)p(i,l)Ci',$$

то есть *C'(Nl)/C(Nl)<a_max* и *C(Nl)/C'(Nl)<1/a_min*.

Поскольку 1/*a_min* и *a_max* — требуемые при коррекции коэффициенты изменения концевых узлов, то есть некоторых узлов графа, то максимум отношений не может быть меньше соответствующий значений, что и требовалось доказать.

4. Коррекция счетчиков узлов цикла при изменении среднего числа итераций

При проведении некоторых оптимизации (peeling, loop unswitching, loop splitting by index и т.д. [16,17]), учете дополнительной информации (pragma loop count) кода, а так же для одновременного использования статистики о локальных вероятностях и числе итераций при построении статического профиля, требуется изменить число итераций цикла. Для этого необходимо скорректировать счетчики внутренних узлов и вероятностей перехода по ветвлениям между ними. При этом, чтобы получить требуемое число итераций I, необходимо установить общую вероятность дойти до обратных дуг цикла равной $\frac{1}{I}$. В случае единственной обратной дуги в цикле это можно сделать путем изменения только вероятности перехода по обратной дуге. В общем же случае, то есть при наличии нескольких обратных дуг, изменение вероятности одного выхода приводит к изменению значений счетчиков нескольких узлов. В частности, меняются счетчики, и, следовательно, вероятности дойти до других выходов. Нетрудно убедиться в том, что по этой причине задача коррекции вероятностей соответственно заданному числу итераций является нелинейной с ограничениями. Приведем алгоритм ее решения, имеющий сложность *O(Edges)*, где *Edges* — число ребер в цикле.

Поскольку отсутствует дополнительная информация об относительном изменении счетчиков узлов внутри цикла, следует наложить дополнительное ограничение на решаемую задачу. Потребуем, как и ранее, чтобы минимально изменились значения счетчиков узлов. В таком случае полученная профильная информация будет больше соответствовать действительности, и дальнейшие преобразования станут более эффективными и обоснованными.

Чтобы избавиться от необходимости решать нелинейную задачу с прямым вычислением требуемых вероятностей переходов, предлагается представить цикл со счетчиками как взвешенный ориентированный односвязный ациклический граф, у которого один узел является доминирующим, и есть несколько концевых узлов, то есть таких, из которых не выходят дуги. Часть этих узлов соответствует переходам по обратным дугам, часть - выходам из

цикла. В таком случае, число итераций цикла задается соотношением сумм счетчиков переходов и выходов. Кроме того, счетчики во внутренних узлах цикла должны быть согласованы, то есть для всех узлов, кроме доминирующего и концевых, сумма счетчиков входящих ребер должна быть равна сумме счетчиков исходящих ребер. Теперь для коррекции числа итераций цикла можно воспользоваться алгоритмом коррекции счетчиков ациклического участка.

Алгоритм:

Рассмотрим некоторый сводимый цикл. Обозначим его дуги выхода через *{Ki}* и обратные дуги через *{Bi}* . Пусть *C0* - счетчик входа в цикл, I' - среднее число итераций, которое получилось после откртуки. Сначала мы опишем как провести необходимые вычисления для случая, когда цикл не содержит других циклов, а затем режим задачу для случая вложенных циклов.

1. Вариант внутреннего цикла:

Узлы и дуги цикла приводится к ациклическому виду (Рис.1, преобразование (1)):

Строятся два дополнительных временных узла.

Один узел соединяется со всеми обратными дугами *{Bi}*, в него ставится счетчик *C0*(I'-1)*.

Другой узел соединяется со всеми дугами выхода *{Ki}* , в него ставится счетчик *C0* .

В качестве старых счетчиков для получившихся ребер используются старые счетчики выхода из цикла и старые счетчики обратных дуг.

Применяется алгоритм коррекции счетчиков в ациклическом участке, начиная с этих двух узлов и до головы цикла. В результате, в голове цикла мы получим счетчик *C0*I'* (Рис.1, преобразование (2)).

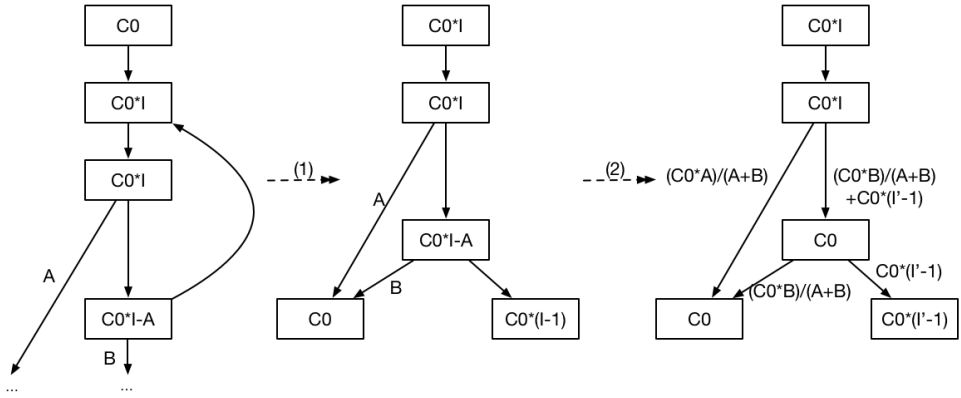


Рис. 1. Пример коррекции значений счетчиков узлов и дуг цикла

2. Вариант цикла, содержащего другие циклы:

Каждый внутренний цикл представляется как один узел, вход в который - вход в цикл, а выходы - обратные дуги цикла.

К внешнему циклу применяются операции, описанные в пункте 1.

Если после пересчета счетчиков внешнего цикла отношение счетчиков на дугах выхода внутреннего цикла не изменилось,

то все счетчики внутреннего цикла умножаются на коэффициент изменения, иначе внутренний цикл пересчитывается согласно пункту 1.

Заметим, что в случае, когда число итераций I' совпадает с начальным числом итераций I, счетчики и вероятности на соответствующих дугах останутся прежними.

5. Коррекция профиля при возникновении противоречивых переходов

При ряде оптимизирующих преобразований, таких как подстановка процедур, расщепление цикла по ветвлению, расщепление схождений производится дублирование участков графа управления или же *расщепление графа управления*. В этом случае часть узлов дублируется, и, поскольку отсутствует дополнительная информация о различии вероятностей аналогичных переходов между копиями узлов, на ветвления проставляются вероятности ветвлений исходного графа. В дальнейшем нередко оказывается, что за счет расщепления вероятность для некоторого перехода в одной из копий стала вычислимой. В этом случае может возникнуть противоречие между вероятностями на дугах и результатом вычисленного сравнения, или же *противоречивый переход,* то есть единственный выход из ненулевого по счетчику числа исполнения узла, имеющий вероятность 0. Пример возникновения такого противоречия приведен на Рисунках [2,3], после удаления невыполнимых дуг возникают узлы с только нулевыми исходящими дугами.

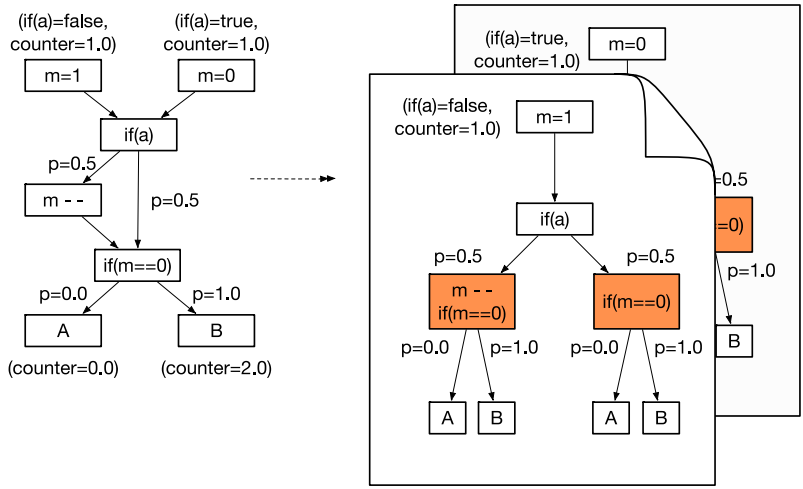


Рис.2. Применение преобразования расщепления с образованием вычислимых операций сравнения

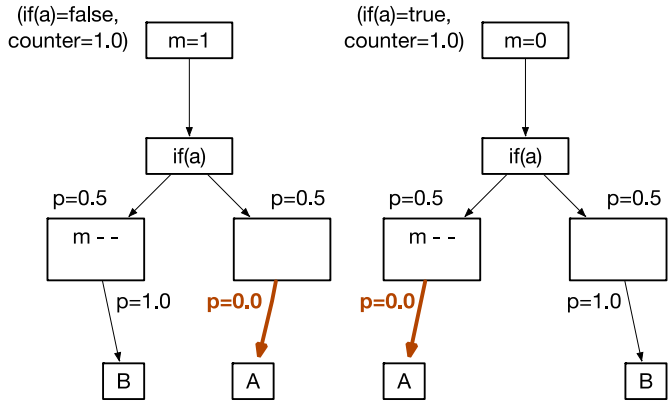


Рис.3. Противоречие после точного вычисления сравнения с удалением дуг

Причина возникающего противоречия заключается в том, что при реальном исполнении программы в узел с противоречивым ветвлением («противоречивый узел») никогда не был бы осуществлен переход, то есть все его счетчики и счетчики узлов, которые он постдоминирует, должны быть нулевыми. Правильная простановка вероятностей приведена на Рисунке 4.

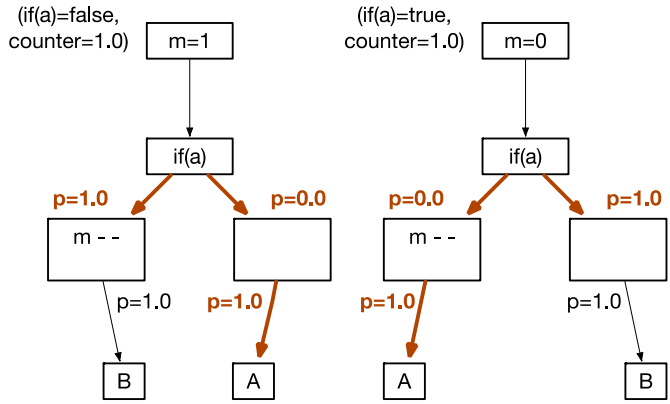


Рис. 4. Необходимая коррекция профиля

Следующий алгоритм позволяет исправить несоответствие и выявить ветвления, по которым должна быть выставлена нулевая вероятность перехода. При этом в порядке обратной топологической сортировки находятся узлы, которые постдоминирует противоречивый узел, их счетчики обнуляются и корректируются вероятности узлов, с ведущими в них ненулевыми переходами.

Алгоритм:

1. Добавим узел с возникшим противоречивым ветвлением к множеству {Node0}.
2. Выбираем произвольный узел N из {Node0}. Помечаем все входящие в него дуги.

   В узлы, из которых выходят эти дуги, записываем число выходящих из них не помеченных дуг.

   Если это число для узла = 0,

   то кладем соответствующий узел в {Node0} и удаляем его из NodeBorder, если он там есть (= если из него выходит больше одной дуги),

   иначе, если мы пометили первую дугу, то кладем соответствующий узел в {NodeBorder}.

   Удаляем узел N из {Node0}.
3. Если в {Node0} есть узел, то идем в 2.
4. Для всех узлов из {NodeBorder} делаем следующее: Между всеми не помеченными исходящими дугами узла распределяем вероятность 1.0 пропорционально начальным вероятностям дуг.

Если встретился узел, для которого все вероятности не отмеченных дуг равны 0, то кладем этот узел в {Node0}, идем в 2.

Если дошли сюда, то закончили работу.

6. Вычисление числа итераций цикла, коррекция профильной и тонкой профильной информации после преобразования открутки итераций цикла

Для эффективного проведения преобразования открутки итераций цикла используется расширенная профильная информация, или *тонкая профильная информация*, а именно, дополнительно к счетчикам числа исполнения узлов цикла собираются счетчики выхода на конкретной итерации для нескольких, далее – *K*, первых итераций цикла. Полученная информация позволяет достаточно точно определить оптимальное число итераций для открутки. При этом, кроме проведения самого преобразования, в ряде случаев существенным, с точки зрения производительности, является и использование имеющейся расширенной информации для коррекции профиля. Пример существенного различия для проведения последующих преобразований приведен на Рис.5 и Рис.6. В обоих случаях приведен пример идентичного с точки зрения профиля исходного цикла и существенно различающегося остатка после открутки одной итерации. Причина заключается в различии тонкой профильной информации — в обоих случаях среднее число итераций цикла равно 2, но в первом случае из цикла всегда осуществляется выход после второй итерации, во втором — в 90% случаев после первой и в 10% после 11-ой. В результате преобразования в первом случае остается цикл со средним числом итераций, равному 1, а во втором —10-ти.

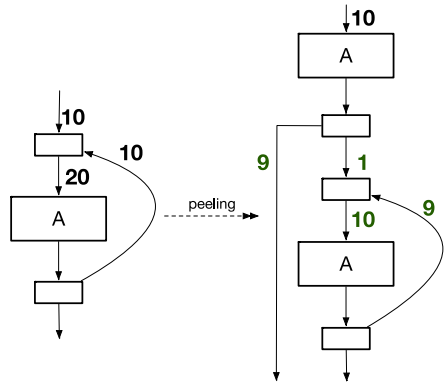


Рис.5. Результат применения открутки одной итерации в первом случае

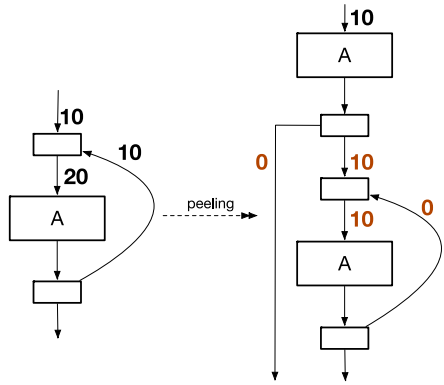


Рис.6. Результат применения открутки одной итерации во втором случае

Рассмотрим цикл со средним числом итераций I и вероятностями выхода на итерациях *p(k), k<=K,* и пусть при открутке было вынесено *m<=K* итераций. Тогда, с учетом тонкой профильной информации, меняется способ вычисления числа итераций нового цикла:

$$I' = \frac{(I - \sum_{k \le m} p(k) * k)}{(1 - \sum_{k \le m} p(k))} - m$$

С учетом наличия тонкой профильной информации меняется и механизм коррекции профиля вынесенных участков. Для каждой вынесенной итерации значения счетчиков и вероятностей пересчитываются поочередно. Как и в случае цикла строится два дополнительных узла - обобщенный выход и переход дальше. В обобщенный узел выхода ставится *C1= C0*p(k)/(1-p(1)-p(2)-.... - p(k-1))*, а в обобщенный узел перехода к следующей итерации ставится *C0-C1*. Далее, от этих узлов и до узла входа в итерацию применяется алгоритм коррекции счетчиков.

Кроме того, требуется пересчитать и тонкую профильную информацию для возможности ее дальнейшего использования. Рассмотрим цикл, у которого выносится *m* итераций.

а) При *k<=K-m* вероятности выхода из цикла точно вычисляются. Пусть $P1 = \sum_{k \le K} p(k)$ - вероятность не попасть в новый цикл, тогда вероятности выйти на итерациях нового цикла $p'(k) = p(k+m)/(1-P1)$.

б) При K-m<k<=K распределение вероятностей выходов по итерациям неизвестно. Поэтому распределим вероятности выхода таким образом, чтобы они были согласованы с числом итераций нового цикла, и чтобы при этом не менялись вероятности выхода при k<=K-m. Число итераций нового цикла:

$$I' = \sum_{k<\infty} p'(k) * k = \sum_{k \le K-m} p'(k) * k + \sum_{k>K-m} p'(k) * k \quad (1)$$

Обозначим $I1'$ и $I2'$ первое и второе слагаемое соответственно. Значения I' и $I1'$ известны, и задача состоит в том, чтобы получить вероятности $p'(k)$ из $I2'$.

Введем их сумму

$$P2' = \sum_{k>K-m} p'(k) = 1 - \sum_{k\le K-m} p'(k)$$

и произведем замену

$$l=k-(K-m),\ p''(l)=p'(l+K-m) \quad (2),$$

тогда

$$P2' = \sum_{l>1} p''(l) (3).$$

Из (1), (2) и (3) следует

$$I2' = (K-m) * P2' + \sum_{l\ge 1} p''(l) * l.$$

Таким образом, нужно подобрать такие $p''(l)$, чтобы

$$\sum_{l\ge 1} p"(l) * l = I2' - (K-m) * P2' = S \quad (4),$$

и при этом выполнялось (3).

Удачное решение дает распределение Пуассона, характеризуемое функцией вероятности $q(k) = a^k e^{-a}/k!$. Причина выбора именно этого распределения заключается в том, что оно соответствует одинаковой вероятности выйти из цикла для каждой последующей итерации, при условии прохода через голову цикла. Его первый момент, $\sum_{l\ge 1} q(l) * l$, равен a . Чтобы им воспользоваться, нам нужно скорректировать вероятности в (4), поскольку $P2'$ не равно единице. Для этого произведем замены $q(k)=p''(l)/P2'$ и $T=S/P2'$.

Получаем при $K-m<k<=K$ вероятности выхода:

$$p'(k) = P2' * q(l) = P2' * T^l * e^{-T}/l!,$$

где $P2' = 1 - \sum_{k\le K-m} p'(k)$, $T = \frac{(I-\sum p'(k)*k)}{P2'} - K - m$, $l=k-(K-m)$.

7. Заключение

В статье рассматривается проблема сохранения актуальности профильной информации в процессе компиляции. Описаны основные способы хранения информации о числе исполнения участков кода, взаимосвязь между ними и дополнительные характеристики, требующие коррекции профиля после проведения различных преобразований и анализов кода. Представлен алгоритм коррекции счетчиков ациклического участка по выходам, алгоритм

коррекции числа итераций цикла, механизм коррекции профиля при возникновении противоречивых переходов и решены задачи, возникающие при проведении преобразования открутки с использованием тонкой профильной информации.

Все предложенные методы реализованы и используются в процессе компиляции оптимизирующими компиляторами lcc, lccs, lfortran, lfortrans для архитектур Эльбрус и Спарк.

Список литературы

[1]. Chang P. P., Mahlke S. A., Hwu W. W. Using profile information to assist classic compiler code optimizations. Software Practice and Experience, V. 21, No12. -1991.-P. 1301-1321

[2]. W. Chen, R. Bringmann, S. Mahlke, S. Anik, T. Kiyohara, N. Warter, D. Lavery, W. - M. Hwu, R. Hank and J. Gyllenhaal., Using profile information to assist advanced compiler optimization and scheduling, 1993

[3]. Jan Hunicka, Profile driven optimisations in GCC, Proceedings of the GCC Developers' Summit June 22nd-24th, 2005, Ottawa, Ontario Canada

[4]. Волконский В. Ю., Ермолицкий А. В., Ровинский Е. В. Развитие метода векторизации циклов при помощи оптимизирующего компилятора. Высокопроизводительные вычислительные системы и микропроцессоры: сборник трудов ИМВС РАН, Выпуск N8, 2005.

[5]. Dmitry M Maslennikov, Vladimir Y Volkonsky: Compiler method and apparatus for elimination of redundant speculative computations from innermost loops. Elbrus International October 9, 2001: US06301706

[6]. Pengfei Yuan, Yao Guo , Xiangqun Chen, Experiences in profile-guided operating system kernel optimization, Proceedings of 5th Asia-Pacific Workshop on Systems, June 25-26, 2014, Beijing, China

[7]. Bo Wu, Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, Raul Silvera, Graham Yiu, Simple profile rectifications go a long way, Proceedings of the 27th European conference on Object-Oriented Programming, July 01-05, 2013, Montpellier, France

[8]. Дроздов А.Ю., Степаненков А.М. Технология оптимизации цикловых участков процедур в компиляторах для архитектур с аппаратной поддержкой конвейризации циклов. Информационные технологии и вычислительные системы №3, М. 2004

[9]. Иванов Д.С. Распределение регистров при планировании инструкций для VLIW-архитектур. Программирование, № 6, 2010, С.74-80

[10]. Bo Wu, Zhijia Zhao, Xipeng Shen, Yunlian Jiang, Yaoqing Gao, Raul Silvera, Exploiting inter-sequence correlations for program behavior prediction, Proceedings of the ACM international conference on Object oriented programming systems languages and applications, October 19-26, 2012, Tucson, Arizona, USA [doi>10.1145/2384616.2384678]

[11]. Calin Cascaval , Luiz De Rose , David A. Padua , Daniel A. Reed, Compile-Time Based Performance Prediction, Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, p.365-379, August 04-06, 1999

[12]. Jeremy Lau , Matthew Arnold , Michael Hind , Brad Calder, Online performance auditing: using hot optimizations without getting burned, Proceedings of the 2006 ACM

SIGPLAN conference on Programming language design and implementation, June 11-14, 2006, Ottawa, Ontario, Canada
[13]. Ананий В. Левитин. Алгоритмы: введение в разработку и анализ — М.: «Вильямс», 2006.— С. 220-224. — ISBN 5-8459-0987-2
[14]. Steven S. Muchnick, Advanced Compiler Design & Implementation, 2003
[15]. Introduction to algorithms, Third Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, 2009
[16]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman "Compilers: Principles, Techniques, and Tools" Book, Pearson Education, ISBN 0-321-48681-1, 2006

Methods of Profile Information Correction during Compilation

O.A. Chetverina <chetverina_o@mcst.ru>
AO "MCST», 117105, Russian Federation, Moscow, 1 Nagatinskaya Str.

Abstract. Performed by compiler code optimizing transformations efficiency can be greatly increased by obtaining and using program profile information, such compiler work is called profile-guided optimization (PGO). After applying transformations, which change control flow graph, it is necessary to adjust the profile information in order to maintain its adequacy for the work of succeeding optimizations. This paper considers two ways of representing profile information and the transition between them, describes the possible causes of uprising additional information on optimized code control flow requiring profile updating. Most frequently encountered problems of profile information update are considered and following solutions are offered: algorithm of acyclic graph profile correction according to its end nodes values; algorithm of loop average iteration number correction; control flow with "controversial node" profile correction algorithm. It is proved that the suggested algorithms of acyclic region counters and the loop iterations counter correction allow to change the counters ratio with original graph counters minimally. With the use of proposed algorithms, the mechanism of correction profile and thin profile information after loop peeling is described. All described methods are implemented in optimizing compilers lcc, lccs, lfortran, lfortrans for Elbrus and Sparc architectures.

Keywords: program profile, PGO, profile correction, loop iterations counter correction, extended profile information, weighted graph

DOI: 10.15514/ISPRAS-2015-27(6)-4

For citation: Chetverina O.A. Methods of Profile Information Correction during Compilation. *Trudy ISP RAN/Proc. ISP RAS,* vol. 27, issue 6, 2015, pp. 49-66 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-4

References

[1]. Chang P. P., Mahlke S. A., Hwu W. W. Using profile information to assist classic compiler code optimizations. Software Practice and Experience, V. 21, No12. -1991.-P. 1301-1321

[2]. W. Chen, R. Bringmann, S. Mahlke, S. Anik, T. Kiyohara, N. Warter, D. Lavery, W. - M. Hwu, R. Hank and J. Gyllenhaal., Using profile information to assist advanced compiler optimization and scheduling, 1993

[3]. Jan Hunicka, Profile driven optimisations in GCC, Proceedings of the GCC Developers' Summit June 22nd-24th, 2005, Ottawa, Ontario Canada

[4]. Volkonskiy V.Y., Ermolitskii A.V., Rovinskii E.V. Razvitie metoda vektorizacii ciklov pri pomoshchi optimiziruyushchego kompilyatora. [Development of loops vectorization method using an optimizing compiler]. Vysokoproizvoditel'nye vychislitel'nye sistemy i mikroprocessory: sbornik trudov IMVS RAN [High-performance computing systems and microprocessors: a collection of IMVS RAS works], Vypusk N8, 2005 (in Russian)

[5]. Dmitry M Maslennikov, Vladimir Y Volkonsky: Compiler method and apparatus for elimination of redundant speculative computations from innermost loops. Elbrus International October 9, 2001: US06301706

[6]. Pengfei Yuan, Yao Guo , Xiangqun Chen, Experiences in profile-guided operating system kernel optimization, Proceedings of 5th Asia-Pacific Workshop on Systems, June 25-26, 2014, Beijing, China

[7]. Bo Wu, Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, Raul Silvera, Graham Yiu, Simple profile rectifications go a long way, Proceedings of the 27th European conference on Object-Oriented Programming, July 01-05, 2013, Montpellier, France

[8]. Drozdov A.YU., Stepanenkov A.M. Tekhnologiya optimizacii ciklovyh uchastkov procedur v kompilyatorah dlya arhitektur s apparatnoj podderzhkoj konvejrizacii ciklov. [Procedures loop areas optimization technology for architectures with hardware loops pipelining] Informacionnye tekhnologii i vychislitenye sistemy.[Information Technology and computer systems] №3, M. 2004 (in Russian)

[9]. D.S. Ivanov, Register allocation with instruction scheduling for VLIW-architectures. Programming and Computer Software. November 2010, Volume 36, Issue 6, pp 363-367

[10]. Bo Wu, Zhijia Zhao, Xipeng Shen, Yunlian Jiang, Yaoqing Gao, Raul Silvera, Exploiting inter-sequence correlations for program behavior prediction, Proceedings of the ACM international conference on Object oriented programming systems languages and applications, October 19-26, 2012, Tucson, Arizona, USA [doi>10.1145/2384616.2384678]

[11]. Calin Cascaval , Luiz De Rose , David A. Padua , Daniel A. Reed, Compile-Time Based Performance Prediction, Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, p.365-379, August 04-06, 1999

[12]. Jeremy Lau , Matthew Arnold , Michael Hind , Brad Calder, Online performance auditing: using hot optimizations without getting burned, Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, June 11-14, 2006, Ottawa, Ontario, Canada

[13]. Ananias V. Levitin . Algorithms : Introduction to the design and analysis [Algoritmy: vvedenie v razrabotku i analiz] - M .: "Williams", 2006.— C. 220-224. — ISBN 5-8459-0987-2 (in Russian)

[14]. Steven S. Muchnick, Advanced Compiler Design & Implementation, 2003

[15]. Introduction to algorithms, Third Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, 2009

[16]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman "Compilers: Principles, Techniques, and Tools" Book, Pearson Education, ISBN 0-321-48681-1, 2006

Методы предварительной оптимизации программ на языке JavaScript

[1]Роман Жуйков <zhroma@ispras.ru>
[2]Евгений Шарыгин <eugene.sharygin@gmail.com>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1., стр. 52, факультет ВМК

Аннотация. Современные виртуальные машины для языка JavaScript используют многоуровневую компиляцию во время выполнения для создания машинного кода. При компиляции во время выполнения нецелесообразно выполнение сложных оптимизаций. Статическая компиляция, наоборот, имеет неограниченные возможности для выполнения сложных оптимизационных преобразований, но не может эффективно применяться к динамическим языкам, таким как JavaScript. В данной работе предлагается общий подход к предварительной компиляции программ на динамических языках, а также применение этого подхода для улучшения двух виртуальных машин — JavaScriptCore и V8. При реализации улучшенной виртуальной машины JavaScriptCore с использованием предварительной компиляции была учтена специфика использования JavaScript-программ в составе локально хранящихся приложений для платформы ARM. Для виртуальной машины V8 для платформы x86-64 в рамках исследования предварительная компиляция была реализована с помощью кэширования в отдельный файл одного из оптимизированных внутренних представлений.

Ключевые слова: оптимизация программ; JavaScript; компиляция во время выполнения; предварительная компиляция; Webkit JavaScriptCore; виртуальная машина V8

DOI: 10.15514/ISPRAS-2015-27(6)-5

Для цитирования: Жуйков Р.А., Шарыгин Е.Ю. Методы предварительной оптимизации программ на языке JavaScript. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 67-86. DOI: 10.15514/ISPRAS-2015-27(6)-5.

1. Введение

В данной работе рассматриваются две виртуальных машины для языка JavaScript. Первая виртуальная машина называется JavaScriptCore (JSC) [1] и

входит в состав браузерного движка WebKit [2] для отображения веб-страниц. Вторая виртуальная машина называется V8 [3] и используется в браузерах Chromium и Chrome. Обе виртуальных машины являются свободным программным обеспечением с открытым исходным кодом и являются многоуровневыми Just-In-Time (JIT) компиляторами, то есть содержат несколько реализаций компиляции программного кода в машинный код во время выполнения программы.

Целью данной работы является разработка подхода к предварительной компиляции программ на динамических языках [4, 5], а также применение этого подхода для разработки схемы улучшения виртуальной машины JavaScriptCore с использованием предварительной компиляции. При реализации улучшенной версии виртуальной машины JavaScriptCore необходимо учесть специфику использования JavaScript-программ в составе локально хранящихся приложений для платформы ARM. Кроме того, необходимо с использованием подхода разработать и реализовать предварительную компиляцию в рамках виртуальной машины V8 для платформы x86-64.

Дальнейшее изложение построено следующим образом. Сначала будет описана общая схема работы многоуровневых JIT-компиляторов, а также особенности реализации этой схемы в виртуальных машинах JavaScriptCore и V8. Потом будут описаны возможные направления улучшения производительности данных виртуальных машин. Далее будет описана общая схема подхода к применению идей предварительной компиляции в рамках многоуровневого JIT, и ее реализация в рамках JavaScriptCore и V8.

2. Многоуровневая JIT-компиляция в JavaScriptCore и V8.

Основная идея многоуровневой JIT-компиляции состоит в том, что время, потраченное на генерацию машинного кода для какого-то участка исходного кода, зависит от горячести этого участка. Для участков кода, которые выполняются один раз, оптимальным вариантом обычно является использование интерпретации. Следующим шагом является базовая JIT-компиляция — генерация неоптимизированного машинного кода, соответствующего заданному участку исходного кода. Для еще более горячих участков кода возможно использование спекулятивных оптимизаций с использованием профиля, собранного на предыдущих уровнях выполнения.

В JavaScriptCore единицей трансляции является функция на языке JavaScript, первыми этапами работы являются лексический и синтаксический анализ. Общая схема работы JavaScriptCore изображена на рис. 1. Исходный код разбивается на токены, методом рекурсивного спуска строится синтаксическое дерево (abstract syntax tree, AST), из которого в свою очередь строится внутреннее представление, называемое байткод (bytecode). В байткоде инструкции хранятся в виде массива ячеек, разные инструкции могут занимать разное количество ячеек. В первой ячейке хранится тип

инструкции, в следующих ячейках хранятся адреса операндов и результата. Адреса операндов могут представлять собой ссылки на константы или номера локальных псевдорегистров. При чтении или записи полей объектов, загрузка адреса поля по имени выглядит как отдельная инструкция, один из операндов которой — константная строка, содержащая имя поля. Для многих инструкций последняя ячейка в байткоде выделена для хранения информации о профиле. Необходимо отметить, что байткод для функции создается непосредственно во время выполнения программы при первом вызове данной функции.

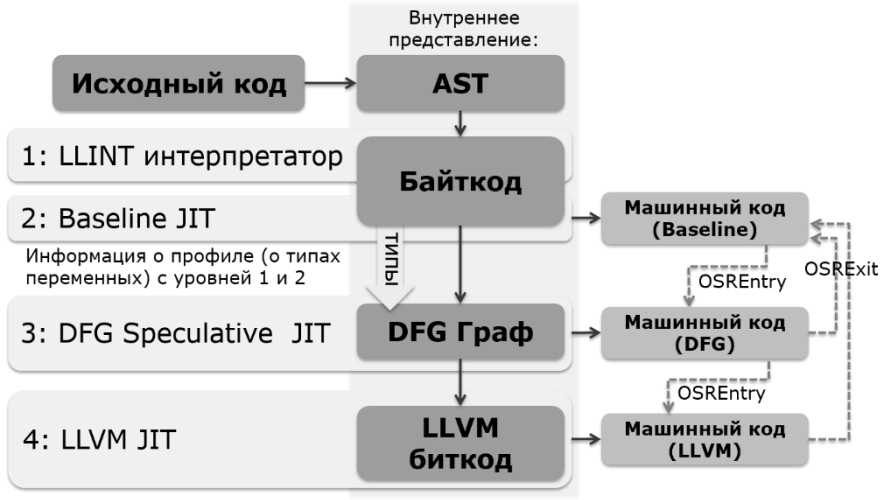


Рис. 1. Устройство виртуальной машины JavaScriptCore

В ранних версиях JavaScriptCore байткод сразу передавался на выполнение интерпретатору. Интерпретатор последовательно читал инструкции байткода и выполнял необходимые действия, переходы и циклы организовывались за счет условных и безусловных операций перехода. Переход указывает, что вместо чтения следующей инструкции в байткоде, интерпретатор должен перейти в другое место. В современных версиях JavaScriptCore вместо интерпретатора используется низкоуровневый интерпретатор (LLInt). Он фактически выполняет те же самые действия, однако запрограммирован на специальном мультиплатформенном ассемблере (offlineasm). Этот специальный ассемблер может быть скомпилирован на этапе сборки JavaScriptCore в машинный код для x86, ARM или нескольких других платформ, а также может быть преобразован в исходный код на языке C. LLInt, как и обычный интерпретатор, позволяет начать выполнение байткода, не выполняя никаких подготовительных этапов, тем самым обеспечивает быстрое начало выполнения. Все другие уровни оптимизации требуют предварительных затрат по созданию машинного кода, соответствующего

заданному участку байткода. LLInt поддерживает на уровне вызова функций взаимодействие со всеми уровнями оптимизации. Если функция уже была скомпилирована в машинный код, то вызов этой функции из низкоуровневого интерпретатора будет выглядеть так же, как и переход на точку входа в общий пролог интерпретатора для любой другой неоптимизированной функции. LLInt использует кэширование на уровне байткода для ускорения доступа к полям объектов по имени.

При работе низкоуровневого интерпретатора так же происходит сбор информации о профиле — сохраняются типы и последние значения полей объектов. Необходимость оптимизации функций определяется с помощью оценки того, сколько раз в ней выполняются те или иные участки кода. Для перехода на первый уровень оптимизации времени выполнения (JIT-оптимизация) необходимо, чтобы функция набрала не менее 100 “очков выполнения”, при этом за каждую пройденную итерацию цикла прибавляется одно “очко”, а за вызов функции — 15 “очков”. Отметим, что эти числа являются примерными, в реальности дополнительно применяется эвристика, результат работы которой зависит от размера рассматриваемой функции. Таким образом, небольшой функции без циклов достаточно быть вызванной около 7 раз, чтобы для нее была выполнена базовая компиляция времени выполнения (Baseline JIT).

Baseline JIT создает для каждой операции байткода соответствующий машинный код. В этом коде реализуются все возможные случаи для данной операции. Например, операция сложения для чисел будет выполнена как сложение, а для операндов-строк — как конкатенация. Генерируемый код будет содержать множество ветвлений для разбора всех таких случаев. После того как для функции будет создан машинный код, нет необходимости дожидаться окончания функции для запуска выполнения нового кода. Например, если функция выполняет цикл с большим числом итераций, то может быть выполнен немедленный переход на новый код (on-stack-replacement, OSR). Низкоуровневый интерпретатор закончит обработку очередной инструкции байткода и сразу перейдет в машинном коде в то место, которое соответствует началу следующей инструкции. Конечно, во всех местах вызова этой функции будет произведено перенаправление на новую версию функции — в машинном коде.

Baseline JIT код используется как базовая версия кода для функций, которые скомпилированы с помощью оптимизирующего JIT-компилятора. Если оптимизированный код сталкивается со случаем, который в нем не поддерживается (например, тип или значение переменной не соответствует собранному профилю), то происходит обратная замена на стеке (on stack replacement exit, OSR exit) к коду Baseline JIT. На уровне Baseline JIT, как и на LLInt сохраняется профиль — информация о типах полей объектов и аргументов функций, и выполняется кэширование для ускорения доступа к полям объектов.

Информация о профиле, собранная на уровнях Baseline JIT и LLInt используется для организации спекулятивного выполнения на следующем уровне оптимизации — оптимизации с использованием графа потока данных (Data flow graph, DFG JIT, Speculative JIT). Собранная информация содержит последние значения загруженных аргументов, полей объектов, а также результатов выполнения функций. Кэширование доступа к полям объектов на уровнях LLInt и Baseline JIT устроено так, что позволяет DFG быстро получать необходимую информацию. Например, по информации кэширования легко можно узнать, что некоторое обращение к полю объекта иногда, часто или всегда возвращает значение некоторого конкретного типа.

DFG JIT компиляция выполняется для функций, которые набрали не менее 1000 “очков выполнения”. На уровне DFG выполняются разнообразные оптимизации, опирающиеся на информацию о профиле. Из байткода с учетом профиля создается граф потока данных, в котором инструкции описаны в виде SSA-представления. На этом DFG графе выполняются оптимизации, и в конце итоговый набор инструкций преобразуется в машинный код.

DFG JIT распространяет полученную информацию о типах переменных по всему графу, и вставляет в код необходимые проверки типов. Иногда DFG даже выполняет спекулятивную оптимизацию по самому значению переменной. Например, если по результатам профилирования поле объекта является конкретной функцией, ее код может быть встроен в вызывающую функцию, с добавлением необходимой проверки. Как было описано выше, когда одна из проверок не выполняется, происходит деоптимизация, то есть обратная замена на стеке (on stack replacement exit, OSR Exit) на код Baseline JIT.

Таким образом, DFG JIT код и Baseline JIT код могут сменять друг друга посредством замены на стеке (OSR). Когда код функции становится “горячим” — происходит переход на DFG JIT. Когда выполняется деоптимизация — происходит обратный переход. В случае многократного OSR exit, сохраненная информация о том, почему произошла деоптимизация, так же становится своеобразным профилем, который позволяет организовать реоптимизацию DFG, то есть создание нового DFG графа и машинного кода с учетом новой информации о профиле. Эвристика, оценивающая необходимость реоптимизации использует экспоненциальную задержку в зависимости от количества уже выполненных реоптимизаций. Это позволяет исключить возникновение больших временных затрат на постоянную реоптимизацию кода и выполнение множества OSR переходов.

Четвертый уровень оптимизации — LLVM JIT, вызывается для функций, набравших не менее 10000 “очков выполнения”. В нем выполняется более широкий набор оптимизаций, а качестве внутреннего представления помимо DFG графа используется биткод компилятора LLVM. Перед генерацией машинного кода выполняются оптимизации, уже реализованные в компиляторе LLVM.

Итак, при выполнении скрипта, в любой момент времени функции, eval-блоки и глобальный код в JSC могут выполняться на любой комбинации LLInt, Baseline JIT и DFG JIT кода. В особом случае при выполнении рекурсивных функций, код одной и той же функции может существовать на стеке вызовов в разных вариантах: в одном уровне функция выполняется на LLInt, в другом на Baseline JIT, в третьем на DFG. Возможен еще более сложный случай — допускается выполнение старого варианта DFG кода на одном уровне стека, в то время как на более вложенном уровне рекурсии произошло много деоптимизаций, и была выполнена реоптимизация, после которой был запущен новый вариант DFG кода.

Табл. 1. Сравнение производительности уровней JSC.

Тест	v8-richards		Browsermark	
Ускорение	Быстрее интерпрета-тора, раз	Быстрее предыдущего уровня, раз	Быстрее LLInt, раз	Быстрее предыдущего уровня, раз
Интер-претатор	1.00	-	н/д	-
LLInt	2.22	2.22	1.00	-
Baseline JIT	15.36	6.90	2.50	2.5
DFG JIT	61.43	4.00	4.25	1.7
Код на C	107.50	1.75	н/д	

Все уровни выполнения обеспечивают одинаковую семантику выполнения, и единственный эффект переключения между ними — в производительности работы JavaScriptCore. В табл. 1 приведены примеры сравнения скорости выполнения теста v8-richards, а также набора тестов Browsermark. Для теста v8-richards дополнительно приведена скорость выполнения аналогичных вычислений, запрограммированных на языке C. Для набора тестов Browsermark не проводилось измерения производительности обычного интерпретатора в составе JSC, указанное в табл. 1 ускорение взято по среднему геометрическому, и необходимо отметить, что в одном из тестов наблюдается пятикратное преимущество Baseline JIT над LLINT интерпретатором, а также есть пример, где DFG JIT оказывается в 6 раз быстрее чем Baseline JIT.

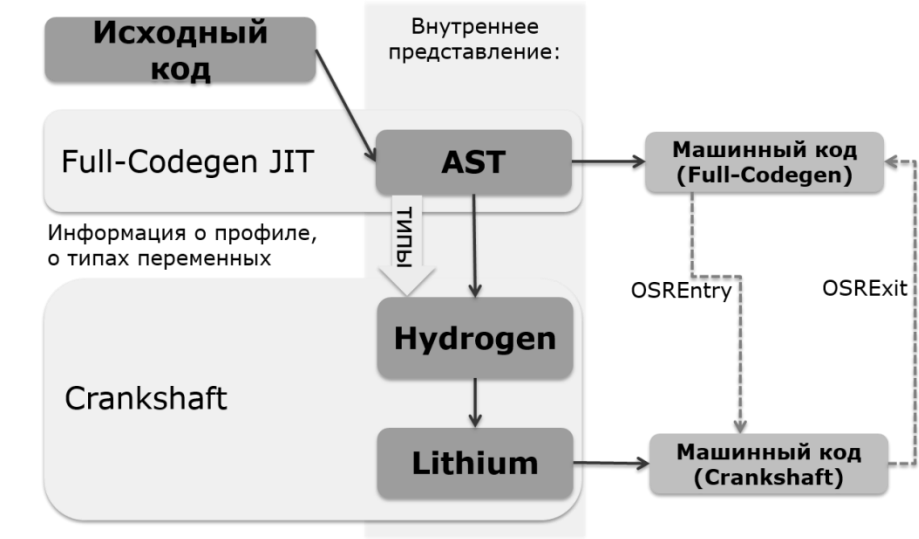


Рис. 2. Устройство виртуальной машины V8

Кратко опишем устройство виртуальной машины V8, изображенное на рис. 2. В V8 используется только два уровня выполнения, причем оба эти уровня осуществляют JIT-компиляцию, а уровень интерпретации отсутствует. Базовый не оптимизирующий JIT-компилятор называется Full-Codegen, а оптимизирующий JIT называется Crankshaft. В Crankshaft в качестве промежуточных представлений используется машинно-независимое представление высокого уровня Hydrogen, на котором выполняется большинство оптимизаций, и машинно-зависимое представление низкого уровня Lithium, на котором происходит распределение регистров и из которого генерируется машинный код. В виртуальной машине V8 так же имеется схема On-Stack Replacement, позволяющая переходить на оптимизированный код, не дожидаясь окончания выполнения функции, и позволяющая в случае ошибочных спекулятивных предположений перейти обратно на неоптимизированную версию машинного кода.

3. Оптимизация производительности многоуровневого JIT

Рассмотрим и проанализируем возможные методы улучшения производительности виртуальных машин. Первым направлением является улучшение имеющихся оптимизаций на спекулятивном уровне JIT-компиляции. Как и в случае статической компиляции программ на типизированных языках в данном направлении существует множество возможностей для адаптации оптимизационных преобразований для

особенностей той или иной платформы, либо определенного класса программ. Однако, необходимо учитывать, что, несмотря на поддержку выполнения JIT-компиляции в отдельном потоке, не прерывая выполнения кода на основном ядре процессора, требования ко времени JIT-компиляции предъявляются более жесткие, так как в отличие от статической компиляции возможны сценарии, когда время, затраченное на более сложную оптимизацию, не окупится в терминах времени выполнения кода. Также необходимо отметить, что дополнительный рост сложности оптимизационных преобразований приводит к большим требованиям к размеру оперативной памяти, что может так же оказаться существенным, например, в случае встраиваемых архитектур.

Следующим направлением оптимизации является создание дополнительных уровней JIT-компиляции с использованием спекулятивных оптимизаций, а также тонкая настройка взаимодействия различных уровней выполнения. Дополнительный уровень JIT-компиляции позволяет для самых горячих участков кода выполнить максимально сложный набор оптимизаций, в то время как менее горячие участки так же будут оптимизированы, но с меньшими затратами. Задача выбора эвристик и управления их параметрами для того, чтобы определять момент перехода на следующий уровень выполнения так же является достаточно обширной областью для исследований. Одним из аспектов тонкой настройки может являться не только аспект производительности, но и аспект энергопотребления процессора.

И наконец третьим направлением улучшения виртуальных машин JavaScript является применение идей предварительной компиляции. Данный подход особенно актуален для использования в сценарии выполнения приложений, локально сохраненных на встраиваемом устройстве. Использование предварительной компиляции позволит в такой ситуации совместить преимущества динамической JIT-компиляции и статической компиляции.

Основным вариантом реализации идей предварительной оптимизации является использование кэширования на уровне различных внутренних представлений виртуальной машины, возможно даже использование комбинации из нескольких внутренних представлений. Для некоторых представлений потенциально возможна разработка дополнительных статических оптимизаций, выполняемых в оффлайн фазе. На примере JavaScriptCore рассмотрим каким образом сохранение и последующая загрузка внутреннего представления позволяют увеличить производительность многоуровневого JIT.

Сначала для интересующих нас наборов тестов v8-v6 и SunSpider составим диаграмму профиля, показывающую какой объем времени затрачивается на выполнение тех или иных этапов работы виртуальной машины JavaScriptCore. Диаграмма изображена на рис. 3.

По такой диаграмме можно указать теоретическое максимально возможное увеличение производительности при наличии сохраненных внутренних

представлений. Например, сохранение такого внутреннего представления как байткод позволяет избавиться от двух начальных этапов — построения синтаксического дерева и непосредственно построения байткода. Однако, при этом на саму загрузку внутренних представлений так же необходимо затратить время выполнения, причем важным аспектом является линковка — восстановление ссылок на все внешние для данного представления объекты.

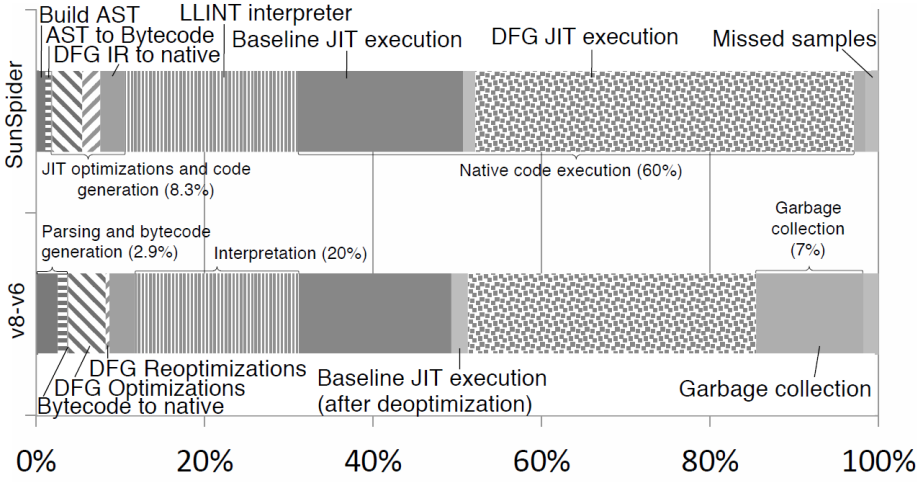


Рис. 3 Результаты профилирования JavaScriptCore

При сохранении оптимизированных внутренних представлений аналогичным образом можно оценить максимально возможное ускорение виртуальной машины для того или иного набора тестов. Для этого, помимо учета той особенности, что можно сэкономить время разбора исходного кода и построения некоторой части промежуточных представлений, необходимо учесть, что успешная загрузка кода позволит выполнять данную единицу трансляции сразу на оптимизированном уровне, не используя более медленные способы выполнения, как это делается в обычном режиме, до тех пор, пока функция не является горячей. Построив для интересующих нас тестов таблицу аналогичную табл. 1, можно получить максимальную оценку дополнительного ускорения, полученного таким образом. Если построить диаграмму как на рис. 3 для тестов из набора Browsermark, по такой диаграмме можно оценить, что потенциально время, обозначенное как “Baseline JIT Execution” может быть уменьшено в 1.7 раза, а время “LLINT interpreter” можно уменьшить в 4.25 раз.

Помимо максимального теоретически возможного ускорения отметим следующие аспекты, которые необходимо принимать во внимание при выборе комбинации внутренних представлений для сохранения. Во-первых, необходимо рассматривать возможность дополнительной статической

оптимизации в оффлайн фазе. Например, для байткода в JavaScriptCore может быть использовано удаление общих подвыражений. Во-вторых, необходимо оценивать сложность линковки во время выполнения. Например, при загрузке машинного кода большое количество времени будет тратиться на замену глобальных адресов, используемых в коде, на актуальные для текущего запуска программы. Третьим аспектом является вопрос поддержки перехода на менее оптимизированный код в случае невыполнения спекулятивных предположений на более высоких уровнях JIT. В обычном режиме работы виртуальной машины все необходимые данные для такого перехода имеются, но при загрузке готового оптимизированного внутреннего представления необходимо так же реализовать создание всей необходимой информации, например, обеспечить готовность создания неоптимизированного варианта JIT-кода. И наконец, в-четвертых, необходимо оценить трудоемкость реализации сохранения и загрузки выбранных внутренних представлений, с учетом реализации предыдущих трех аспектов, обозначенных в данном абзаце.

4. Реализация предварительной оптимизации в JavaScriptCore

В рамках JavaScriptCore была разработана идея инфраструктуры, в которой исходный код заранее преобразовывался в некоторый набор данных, содержащий байткод и другие внутренние представления. Впоследствии, при выполнении программы загружаются готовые оптимизированные внутренние представления, которые корректируются по мере необходимости. Общая схема работы разрабатываемой системы изображена на рис. 4.

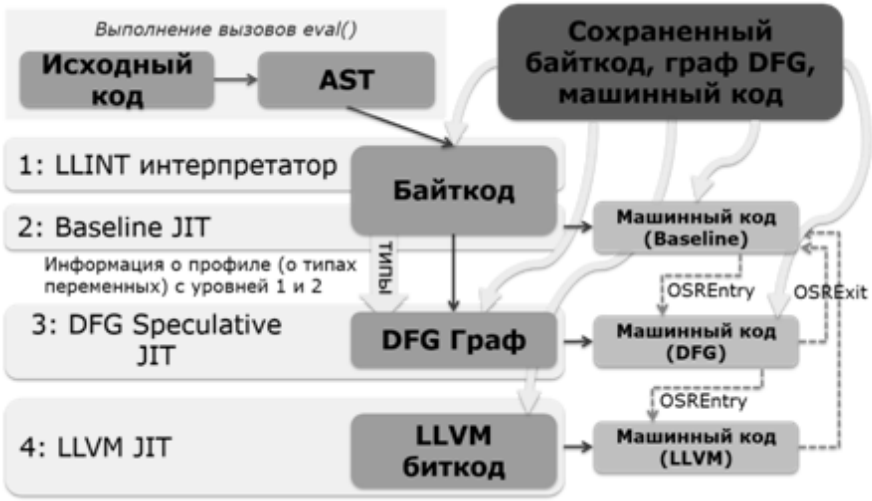


Рис. 4. Устройство системы предварительной компиляции (AOTC).

Такой порядок работы JavaScriptCore позволит получить помимо преимуществ, указанных в предыдущей главе, некоторое подобие шифрования исходного кода — ведь байткод и другие бинарные внутренние представления сложнее прочесть и изменить, чем обычный исходный JavaScript код.

4.1 Сохранение и загрузка байткода

Сначала в рамках работы над предварительной компиляцией (ahead of time compilation, AOTC) был реализован первый этап, который можно назвать AOTB (ahead of time bytecode) [6]. Он подразумевает сохранение исходного кода в виде байткода, для последующей загрузки при выполнении.

В обычном режиме работы JavaScriptCore при выполнении скриптов байткод для каждой функции генерировался только при первом вызове этой функции. Нами была разработана и реализована схема генерации и сохранения байткода без выполнения самого скрипта. К байткоду сохраняется так же вспомогательная информация, такая как таблицы констант, таблицы switch-переходов и исключений, необходимые данные для регулярных выражений. Для сохранения байткода без выполнения потребовалось эмулировать работу стека пространств имен.

Байткод JavaScriptCore не был задуман как промежуточное внутреннее представление для сохранения, основной его целью является эффективное выполнение и генерация машинного кода на уровне Baseline JIT. Байткод, в отличие от исходной программы на JavaScript, отражает семантику программы только в определенном контексте. Например, в зависимости от свойств объектов, созданных к моменту начала выполнения программы, для нее может быть сгенерирован различный байткод. В основном, эта разница в байткоде относится к дополнительным подсказкам, например, позволяющим быстрее организовать обращение к полям объектов. Однако, в некоторых случаях байткод, сохраненный вне того контекста, в котором программа будет исполняться, может приводить к некорректным результатам с точки зрения стандарта JavaScript. Эти особенности были учтены при сохранении байткода без выполнения. Кроме того, обращения к глобальным объектам содержат абсолютные адреса, и необходимо организовать сохранение так, чтобы можно было при загрузке байткода поменять адреса на новые, соответствующие адресам объектов во время выполнения.

Изначально планировалось хранение всей информации в виде базы данных SQLite, однако такого способа хранения пришлось отказаться из соображений эффективной загрузки. Теперь все данные, относящиеся к одной функции, хранятся в виде последовательного набора байтов внутри файла. В начале файла сохраняется карта адресов (смещений), по которым можно найти информацию для каждой из функций. Соответственно, при выполнении из файла читается эта карта смещений и байткод для глобального JavaScript кода, то есть всего кода, описанного вне функций. В дальнейшей работе, при

первом вызове функции вместо обычного разбора исходного кода байткод и все необходимые данные подгружается из файла по заданному смещению.

Необходимо отметить один из моментов, который позволил уменьшить размер сохраняемого файла — отказ от хранения двух вариантов байткода для каждой функции. При обычном выполнении JavaScript программ, для функций, вызываемых как конструктор с помощью вызова new (“var z = new f()”), создается отдельный байткод. В нашей реализации хранится только байткод для случая обычного вызова функции, который при необходимости преобразуется в вариант “для конструктора”.

При выполнении байткода вместо исходного кода стандарт ECMA-262 [7] поддерживается полностью, вызовы eval поддерживаются, для них исходный код компилируется обычным образом в процессе работы JavaScriptCore. Исключением является только работа операций, явным образом требующих наличия исходного кода. Примерами таких операций могут служить вызовы function.toString(), либо использование поля line у объекта исключения. В этом поле должен храниться номер строки в исходном коде, которая создала исключение.

4.2 Результаты AOTB

Текущая реализация сохранения и загрузки файла с байткодом успешно проходит регрессионное тестирование на наборах из Webkit JavaScriptCore и V8. За счет уменьшения времени обработки исходного кода, до 2 – 4% ускоряется работа тестов из SunSpider, v8 и kraken на платформе ARM. Крупные data-файлы для тестов из kraken обрабатываются значительно быстрее, время их обработки не учитывается в результатах теста. По результатам профилирования работы JavaScriptCore было выявлено, что на больших исходных текстах время, затрачиваемое на загрузку файла с байткодом, может быть до 3x раз меньше времени, необходимого на обычную обработку исходного кода.

Для тестов из наборов SunSpider, v8, kraken было измерено соотношение размера бинарного файла с сохраненным байткодом и размера исходного JavaScript-файла. Причем, был взят пример как использования оригинальных файлов, так и упакованных с помощью Google Closure Compiler. Во втором случае оба файла дополнительно архивировались с помощью утилиты gzip с использованием максимального сжатия. Результаты данных измерений изображены в табл. 2.

Тестовый набор	Соотношение размеров файла с байткодом и исходного файла	
	Оригинальный JavaScript	Google Closure Compiler + gzip
SunSpider	1.19	1.25
v8-v6	2.3	4.41
Kraken	1.97	1.31

Табл. 2. Результаты сравнения объема JS-файлов и файлов с байткодом.

Увеличение размера файла является значительным, но при выполнении файла с байткодом не используется больший объем оперативной памяти, чем при запуске JavaScriptCore в обычном режиме для выполнения аналогичной JavaScript программы. В современных встраиваемых системах объем внешней памяти существенно превышает объем оперативной памяти, и не является критическим параметром при создании программного обеспечения, поэтому инфраструктура AOTB может иметь достаточно широкое применение.

4.3 Сохранение машинного кода Baseline JIT

После реализации AOTB в данную инфраструктуру для платформы x86-64 были добавлены сохранение и загрузка неоптимизированного машинного кода, генерируемого на уровне Baseline JIT. Самой сложной задачей в этой работе была линковка адресов. При загрузке, для всех объектов, обращение к которым производится по абсолютным адресам, необходимо было внести актуальные адреса в загружаемый машинный код. На этапе сохранения генерация машинного кода была так же реализована без выполнения скрипта. При генерации машинного кода, все обращения по абсолютным адресам фиксировались в отдельные таблицы, чтобы в дальнейшем была возможность восстановить необходимые адреса при загрузке. Машинный код и все необходимые для линковки данные записывались рядом с байткодом соответствующей функции в тот же бинарный файл. При этом абсолютные адреса в машинном коде, которые будут заново сформированы при загрузке, при сохранении в файл заменялись нулями для более успешной работы алгоритмов сжатия.

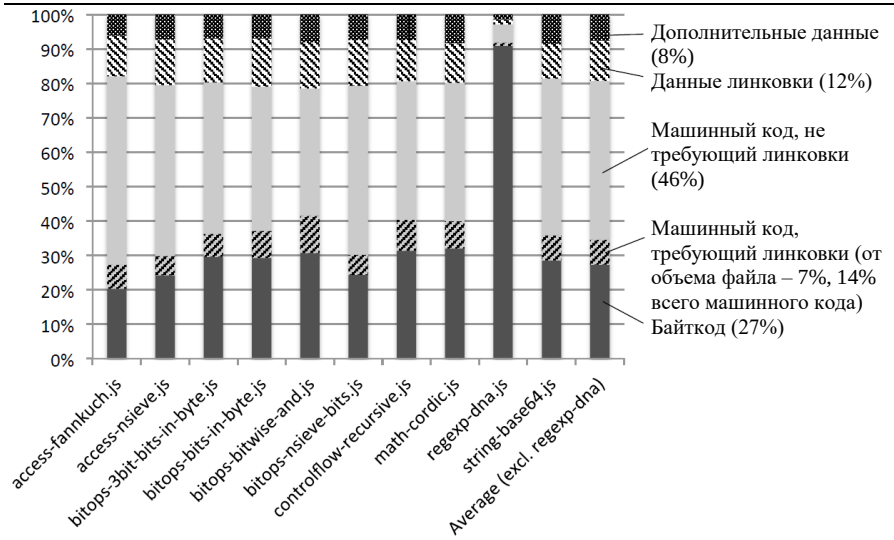


Рис. 5. Структура бинарного файла

В итоге для каждой функции в файле хранились следующие группы данных:

- Байткод и другая информация, необходимая для AOTB
- Машинный код, создаваемый Baseline JIT
- Данные для линковки
- Дополнительные технические данные генерируемые для функции после работы Baseline JIT, их необходимо сохранять из-за пропуска генерации Baseline JIT в варианте запуска с загружаемым готовым машинным кодом

Размер получаемых файлов оказался еще в 2.5 – 5 раз больше, чем размер файлов, генерируемых при работе AOTB, то есть содержащих только байткод. На рис. 5 изображена статистика по объему данных, содержащихся в полученных бинарных, для 10 тестов из набора SunSpider.

Поскольку на x86-64 размер указателя составляет 8 байт, получается, что значительная часть машинного кода состоит из абсолютных адресов. В данном случае, под линковку попадает от 10% до 23%, в среднем 14% машинного кода. Возможно, из-за этого оказывается, что сохранение машинного кода не позволяет ускорить производительность JavaScriptCore на тестах SunSpider и V8. Получается, процесс генерации машинного кода из готового байткода оказывается не таким уж медленным, и чтение значительно более объемного файла с последующей линковкой адресов не дает выигрыша в производительности.

5. Предварительная оптимизация в виртуальной машине V8

Поскольку в виртуальной машине V8 отсутствует самое общее внутреннее представление аналогичное байткоду, было выбрано принципиально иное инфраструктурное решение о реализации предварительной компиляции. Файл с сохраняемой информацией должен лишь дополнять исходный файл, а не заменять его.

От сохранения машинного кода было решено отказаться из-за сложности реализации и плохих практических результатов такого подхода в JavaScriptCore. Сохранение сложного высокоуровневого представления Hydrogen так же было решено не делать, в силу его сложной структуры, необходимой для проведения всех оптимизационных проходов. Машинно-зависимое внутреннее представление Lithium оказалось оптимальным для сохранения – в нем достаточно легко собирать необходимый объем информации для генератора машинного кода, и одновременно не стоит так остро вопрос актуализации ссылок на используемые объекты, как это происходит с адресами в машинном коде.

5.1 Сохранение Lithium

Итак, для виртуальной машины V8 была разработана система кэширования в файл оптимизированного внутреннего представления Lithium уже после оптимизационного прохода распределения регистров, то есть сохраняется ровно тот вариант внутреннего представления, который передается генератору машинного кода. Один из недостатков этого подхода – зависимость от архитектуры системы, так как представление является машинно-зависимым. В рамках данной работы была выполнена исследовательская реализация для процессоров x86-64.

Система встроена в V8 и доступна при запуске через интерфейс командной строки. Если обычный запуск производится командой v8 test.js, то модифицированная нами версия v8 позволяет добавить опцию -save-code=<filename> или -load-code=<filename> соответственно для сохранения и загрузки Lithium представления. Необходимо отметить, что в данной реализации поддерживается работа только с одним исходным файлом на JavaScript. Предположительно, при реализации аналогичного подхода для реальных крупных приложений на JavaScript следует использовать один общий файл кэширующий все Lithium-представления в рамках запускаемого приложения.

Некоторая часть инструкций Lithium не является самодостаточными, и содержит ссылку на «родительскую» инструкцию Hydrogen из которой она была сгенерирована. Генератор машинного кода в Crankshaft использует небольшую часть свойств инструкций Hydrogen, доступных по соответствующим ссылкам из Lithium.

Воссоздание Hydrogen в полном объеме на этапе загрузки предварительно загруженных инструкций Lithium не является целесообразным, а в некоторых случаях это даже невозможно без значительного вмешательства во внутреннюю структуру Hydrogen для отключения или обхода многочисленных проверок целостности.

В ходе данной работы было принято решение о создании дополнительного уровня между Hydrogen и Lithium. Данный дополнительный уровень содержит в точности объем информации необходимый для генерации машинного кода, имеет аналогичный Hydrogen интерфейс доступа к свойствам инструкций и содержит лишь нужную часть этих свойств.

Идентичность интерфейса позволяет минимизировать изменения, вносимые в генератор машинного кода компилятора Crankshaft, требуется лишь заменить обращения к тем же методам другого класса. Классы инструкций дополнительного уровня образуют иерархию, схожую и оригинальной иерархией классов инструкций Hydrogen. Инструкции дополнительного уровня сохраняются вместе с инструкциями Lithium.

5.2 Результаты

На искусственных тестовых примерах ускорение работы виртуальной машины составило до 20 раз. Для создания такого примера, необходимо выбирать такое количество итераций цикла внутри функции, которое минимально подходит для запуска Crankshaft-компиляции данной функции. Тогда в режиме запуска с загружаемым готовым представлением Lithium получается, что такая функция все итерации выполняет на оптимизированной версии кода, в то время как исходном режиме работы виртуальной машины V8 такая функция почти все время выполнялась на неоптимизированной версии машинного кода. А соотношение до 20 раз может быть достигнуто при добавлении в цикл нескольких присваиваний в неиспользуемые в дальнейшем переменные, поскольку не оптимизирующий компилятор Full-Codegen не реализует удаления мертвого кода.

Необходимо отметить, что реализация сохранения и загрузки Lithium была сделана лишь в рамках исследовательской работы, поэтому не была реализована совместимость механизма сохранения и загрузки внутреннего представления с обычно используемым на современных многоядерных процессорах режимом работы виртуальной машины V8, при котором компиляция оптимизированного кода происходит в фоновом режиме в отдельном потоке параллельно выполнению самого скрипта. В качестве базового времени для сравнения производительности использовались результаты работы оригинальной версии V8 с отключенной компиляцией в фоновом режиме.

Тестирование производительности производилось только при запуске тестов в режиме загрузки с теми же параметрами и входными данными, что и при сохранении. Данное ограничение связано с реализацией сохранения и загрузки

только для одного исходного файла в рамках одного запуска, а также с отсутствием промышленных тестов для языка JavaScript, позволяющих передавать входные данные посредством командной строки.

К сожалению, на тестах из наборов SunSpider, Octane и Kraken не было получено ускорения. Возможно, это связано с тем, что в данных тестах незначительное время тратится на работу оптимизирующего компилятора, а также нет функций, выполнение которых существенное время происходит на неоптимизированном машинном коде с последующей оптимизацией.

Поскольку в виртуальной машине V8 отсутствует поддержка статической генерации функций, реализованных на языке asm.js [8], являющемся подмножеством языка JavaScript, было решено протестировать сравнительно объемные тесты на asm.js.

Ускорение было получено на трех asm.js приложениях с сайта “Are we fast yet?” [9], занимающегося сравнением производительности различных JavaScript движков. Тесты Box2d и Bullet данного сайта ускоряются до 15%, а тест Zlib — до 33%. Данные тесты можно запускать с указанием выполняемого объема вычислений, передавая в командной строке число от 1 до 5. Важно отметить, что такое ускорение получено при выполнении с параметром 1. Однако, в абсолютном выражении данное ускорение сохраняется и при использовании большего объема вычислений, тесты ускоряются на 0.1 – 0.2 секунды. Таким образом, можно предположить, что при реализации данной исследовательской работы в виде реально используемого программного продукта в составе веб-движка для отображения веб-страниц, пользовательское JavaScript-приложение, основанное на box2d, будет загружаться на 0.1 секунду быстрее при использовании сохранения и загрузки Lithium.

Дополнительно был создан еще один тест на asm.js. Для статических компиляторов языка C++ существует тест tramp3d [10], предназначенный для оценок времени работы самих компиляторов. То есть обычно данный достаточно объемный исходный код на C++ используется для сравнения скорости компиляции. В рамках исследования, данный тест был скомпилирован с помощью emscripten в исходный файл на языке asm.js и полученный скрипт был протестирован. Поскольку в V8 компиляция происходит во время выполнения, и подход с загрузкой готового ранее сохраненного Lithium кода позволил уменьшить время компиляции, на таком тесте было получено ускорение в 30%.

6. Заключение

В рамках данной работы разработан общий метод применения предварительной компиляции в виртуальных машинах с использованием многоуровневой JIT-компиляции. С помощью этого метода разработан и реализован для платформы ARM оптимизирующий JavaScript компилятор на базе виртуальной машины JavaScriptCore, позволяющий добиться более

высокой производительности для локально хранимых программ. Данная реализация позволяет избавиться от хранения JavaScript кода в виде открытых исходных файлов, сохраняя код в файл в виде байткода JavaScriptCore, и позволяет сократить до 3х раз время, затрачиваемое во время выполнения для получения байткода, поскольку компилятору не нужно делать лексический и синтаксический анализ. Дополнительно к байткоду в файл может быть также сохранен неоптимизированный машинный код. На некоторых тестах данная реализация позволяет получить ускорение до 2 – 4%, при этом размер файла с байткодом может быть до 4.5 раз больше исходных файлов на JavaScript, а при сохранении машинного кода объем файла увеличивается еще в 2.5 – 5 раз без существенных улучшений производительности. В дальнейшем планируется продолжать разработку системы, добавив в нее предварительные оптимизации на уровне байткода.

Для виртуальной машины V8 с помощью применения метода разработан механизм кэширования внутренних представлений и реализована возможность сохранения и последующей загрузки промежуточного оптимизированного машинно-зависимого внутреннего представления, что позволяет ускорить повторный запуск программы. Данная реализация позволяет значительно ускорить специальный образом подобранные искусственные тесты, а также ряд тестов на языке asm.js. В данный момент — это лишь исследовательская разработка и в дальнейшем планируется расширить ее для возможности реального использования в составе систем, аналогичных движкам веб-браузеров.

Список литературы

[1]. Веб-страница описания реализации JavaScriptCore на веб-сайте разработчиков WebKit. http://trac.webkit.org/wiki/JavaScriptCore
[2]. Веб-сайт Webkit. http://www.webkit.org
[3]. Веб-сайт V8. https://code.google.com/p/v8/
[4]. S. Hong, J. Kim, J. W. Shin, S. Moon, H. Oh, J. Lee, H. Choi “Java client ahead-of-time compiler for embedded systems”, Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, 2007, pp. 63-72
[5]. S. Hong, S. Moon “Client-Ahead-Of-Time Compilation for Digital TV Software Platform” 3rd workshop on Dynamic Compilation Everywhere preprint, 2013. http://sites.google.com/site/dynamiccompilationeverywhere/home/dce-2014/DCE-2014-Sunghyun-Hong-article.pdf
[6]. Р. Жуйков, Д. Мельник, Р. Бучацкий, В. Варданян, В. Иванишин, Е. Шарыгин. Методы динамической и предварительной оптимизации программ на языке JavaScript. // Труды Института системного программирования РАН Том 26. Выпуск 1. 2014 г. Стр. 297-314. DOI: 10.15514/ISPRAS-2014-26(1)-10
[7]. Описание стандарта ECMA-262. http://www.ecma-international.org/publications/standards/Ecma-262.htm
[8]. Веб-сайт языка asm.js. http://asmjs.org/
[9]. Веб-сайт “Are we fast yet?”. https://arewefastyet.com/
[10]. Веб-страница тестирования tramp3d. http://gcc.opensuse.org/c++bench/tramp3d/

Ahead of time optimization for JavaScript programs

[1]*Roman Zhuykov <zhroma@ispras.ru>*
[2]*Eugene Sharygin <eugene.sharygin@gmail.com>*
[1]*Institute for System Programming of the RAS,*
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. Modern JavaScript engines use just-in-time (JIT) compilation to produce binary code. JIT compilers are limited in a complexity of optimizations they can perform at a runtime without delaying an execution. Static ahead-of-time (AOT) compilers do not have such limitations, but they are not well suited for compiling dynamic languages such as JavaScript. In the paper, we discuss methods for augmenting multi-tiered JIT compiler with a capability for AOT compilation, and implement some of ahead-of-time compilation ideas in two JavaScript engines — JavaScriptCore and V8. In JavaScriptCore (JSC), which is a part of open-source WebKit library, we have developed and implemented a framework, which allows saving of JavaScript programs as a binary package containing bytecode and a native code. The framework consists of two components: command-line compiler, which compiles source JavaScript program into compressed binary package, consisting of JSC internal representation called bytecode and native code produced by JSC's non-optimized JIT compiler. The second component is patched JSC engine with a capability for loading and executing binary packages produced by the compiler. In addition, we have implemented saving of optimized internal representation in another JavaScript engine, which is called V8 and is used in Chrome and many other browsers. When running the same JavaScript program, cached internal representation can be loaded to reduce JIT-compilation time and decrease percentage of running unoptimized code before it became hot.

Keywords: program optimizations; JavaScript; just-in-time optimization (JIT-optimization); ahead of time optimization; Webkit JavaScriptCore; V8.

DOI: 10.15514/ISPRAS-2015-27(6)-5

For citation: Zhuykov Roman, Sharygin Eugene. Ahead of Time Optimization for JavaScript Programs. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 67-86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-5

References

[1]. JavaScriptCore webpage on WebKit website. http://trac.webkit.org/wiki/JavaScriptCore
[2]. WebKit Browser Engine website. http://www.webkit.org
[3]. V8 Browser Engine webpage. https://code.google.com/p/v8/
[4]. S. Hong, J. Kim, J. W. Shin, S. Moon, H. Oh, J. Lee, H. Choi "Java client ahead-of-time compiler for embedded systems", *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, 2007, pp. 63-72

[5]. S. Hong, S. Moon "Client-Ahead-Of-Time Compilation for Digital TV Software Platform" *3rd workshop on Dynamic Compilation Everywhere preprint*, 2013. http://sites.google.com/site/dynamiccompilationeverywhere/home/dce-2014/DCE-2014-Sunghyun-Hong-article.pdf

[6]. R. Zhuykov, D. Melnik, R. Buchatskiy, V. Vardanyan, V. Ivanishin, E. Sharygin. Metody dinamicheskoj i predvaritel'noj optimizacii programm na jazyke JavaScript [Dynamic and ahead of time optimization for JavaScript programs]. *Trudy ISP RAN* [*The Proceedings of ISP RAS*], 2014, vol. 26, no. 1, pp. 297-314. DOI: 10.15514/ISPRAS-2014-26(1)-10 (in Russian)

[7]. ECMA-262 Standard description. http://www.ecma-international.org/publications/standards/Ecma-262.htm

[8]. Asm.js language website. http://asmjs.org/

[9]. "Are we fast yet?" website. https://arewefastyet.com/

[10]. Tramp3d testing webpage. http://gcc.opensuse.org/c++bench/tramp3d/

Техника инструментирования кода и оптимизация кодовых строк при моделировании фазовых переходов на языке программирования C++

Е.В. Пальчевский <teelxp@inbox.ru>
А.Р. Халиков <khalikov.albert.r@gmail.com>
Уфимский государственный авиационный технический университет,
450000, Россия, г. Уфа, ул. К. Маркса, 12

Аннотация. В данной статье рассматриваются техника написания кода, с помощью которой можно сэкономить время для написания определенной программы, техника инструментирования кода на языке высокого уровня C++, приведены примеры алгоритмов написания кода для системных программ и рассмотрены оптимизированные компиляторы, за счет которых можно сократить время компиляции и ускорить работу программы. Даны определения языка программирования «C» и «C++», кодовой структуры, кодовой оптимизации. Также рассмотрена статистика использования языков программирования по популяризации. Описаны основные преимущества языка программирования «C++», в сравнении с «C». Произведено обозначение структуры исходного кода для более удобного использования. Приведены примеры реализации кодового алгоритма для дальнейшей работы с кодовыми строками. Приведены примеры внятных и кратких комментариев к написанным исходным строкам. Реализованы классовые конструкторы и деструкторы, целью которых является детальная оптимизированность и уменьшение использования блоковой оперативной памяти при моделировании фазовых переходов. Рассмотрены виды оптимизаций: ручная и автоматизированная. В ручной оптимизации описаны несколько подпунктов: peephole, внутрипроцедурная, локальная, межпроцедурная и оптимизационные циклы. Разработан структурированный кода с последовательной заменой логических выражений с проработанными различными типами данных с последующей оптимизацией потребления оперативной памяти персонального компьютера. Показана работа с определенными функциями и представлены практические примеры использования. Произведены расчеты потребляемой памяти до и после оптимизации кодовых строк, на различных компиляторах и серверном оборудовании, конфигурация которого также приведена в данной статье.

Ключевые слова: код; оптимизация кода; инструментирование кода; техника написания кода; C++; программирование

DOI: 10.15514/ISPRAS-2015-27(6)-6

Для цитирования: Пальчевский Е.В., Халиков А.Р. Техника инструментирования кода и оптимизация кодовых строк при моделировании фазовых переходов на языке программирования C++. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 87-96. DOI: 10.15514/ISPRAS-2015-27(6)-6.

1. Введение

Не является секретом тот факт, что техника инструментирования кода – это одна из самых важных составляющих частей, которая позволяет выполнить поставленную задачу максимально качественно.

Актуальность выбранной темы подтверждается тем, что сфера программирования востребована, а язык *C++* занимает одну из лидирующих позиций, что подтверждает диаграмма, отображенная на рис.1.

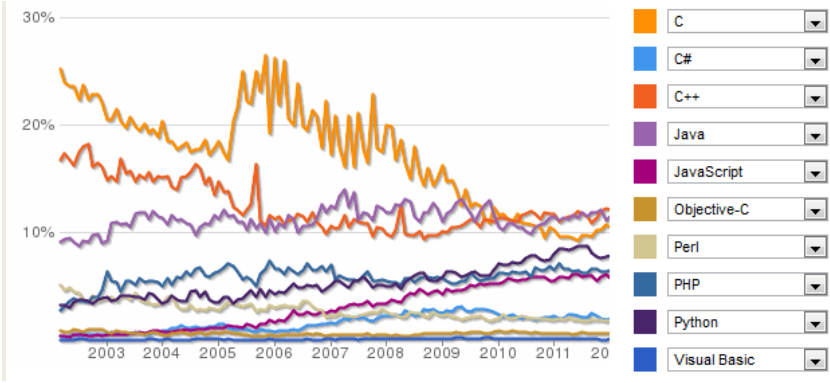


Рис. 1. Статистика языков программирования в 2003-2012 годах [1]

C++ - это язык программирования высокого уровня, который является компилируемым и предназначен для решения задач различного уровня. Этот язык очень широко используется в разработке программного обеспечения и является одним из самых популярных языков программирования. Область разработки включает в себя как написание консольных программ с выводом обычных сообщений, так и разработка крупных проектов, к примеру, операционная система *Windows*. Методики программирования на данном языке – это своеобразный структурированный алгоритм, за счет которого можно как ускорить работу кода, так и грамотно его разделить по частям, тем самым создавая удобства при программировании какого-либо проекта [2].

Если брать в сравнение с языком «*C*», то можно отметить следующие основные нововведения:

- в данном языке программирования, включена поддержка объектно-ориентированного программирования через классы, что позволяет строить абстрактные типа данных, к примеру, структуры и перечисления, интерфейсы, метаклассы, подпроцессы и характеризовать их своими функциями и особенностями;
- имеется поддержка программирования через шаблоны функций и классов, если называть иначе, то «обобщённое программирование», которое дает возможность описания алгоритмов и данных так, что можно не менять самого описания и есть возможность применить это описание к различным типам данных;
- появилась возможность обрабатывать исключения, что позволяет описывать реакцию программы на ошибки времени выполнения и другие проблемы, называемые «исключениями», появление которых может возникнуть при компиляции программы и могут привести к невозможной ее дальнейшей отладке;
- виртуальная функция или виртуальный метод. Используется для переопределения в классах-наследниках для того, чтобы вызов для конкретной реализации метода будет определен во время исполнения самой программы;
- встраиваемые функции «*inline*», которые используются для ускорения программы. Сам вызов функции занимает намного больше времени, чем код, который будет без написания данных функций. И если использовать встроенные функции для замены вызовов, то компиляция программы будет ускорена в 5-6 раз;
- перегрузка имен функций, которая позволяет определять множество функций с одинаковым именем и с точным типом возвращаемого значения.

2. Техника написания кода на языке программирования C++

Для того чтобы код функционировал правильно – нужно тщательно продумывать логику будущей программы.

Нужно разработать алгоритм программы, за счет которого будет написан будущий код. Грамотный подход составления алгоритма программы – это написание будущей программы псевдокодом, который использует ключевые слова языков программирования и опускает специфический синтаксис.

2.1 Обозначение структуры кода

Кодовая структура – это неотъемлемая составляющая всего кода, за счет которой достигается «понимание» программы [3].

Работать с определенными функциями очень удобно, так как есть разделяющие компоненты связанности, что упрощает структурированность всей программы. Но более удобно работать с такими вещами, как публичные классы с определенной точностью символов и отличающимися названиями. В качестве примера можно привести два публичных класса с конструкторами и деструкторами [4]:

- *class UfaClass*: *public ParentClass* // Название класса «*UfaClass*», в котором имеется публичный родительский класс «*ParentClass*».

```
  {
    public:
  UfaClass(); // конструктор
  ~UfaClass(); // деструктор
protected:
  // Доступ элементов секции из родительского класса.
private:
  // элементы доступны по умолчанию из класса.
};
```

и второй точно такой же публичный класс, с точно таким же названием:

- *class UfaClass*: *public ParentClass* // Название класса «*UfaClass*», в котором имеется публичный родительский класс «*ParentClass*».

```
  {
    public:
  UfaClass(); // конструктор
  ~UfaClass(); // деструктор
protected:
  // Доступ элементов секции из родительского класса.
private:
  // элементы доступны по умолчанию из класса.
};
```

Несмотря на комментарии в самом коде, можно запутаться даже в самой небольшой структуре кода программы. Именно поэтому рекомендуется использовать разные заголовки классов и публичных функций [5].

2.2 Внятные и краткие комментарии к написанному коду

Для более быстрой ориентации в коде, нужно оставлять комментарии после каждого написанного класса или функции, что показано в следующем примере, в котором рассмотрено выравнивание данных в оперативной памяти:

- *struct M*{ //
  char m; // целочисленный (символьный) тип данных
  long int p; // целочисленный тип данных «*p*»
  int n;
  };

3. Оптимизация кода на языке С++

Кодовая оптимизация – это преобразования кода за счет различных методов с целью улучшения различных атрибутов и характеристик. Основными целями кодовой оптимизации являются три составляющие:

- ускорение компиляции кода;
- ускорение работы самой программы;
- уменьшение объема кода;
- уменьшение потребления системных ресурсов.

Кодовая оптимизация может производить двумя способами:

- ручной способ. Программист сам отыскивает нужные участки кода и производит оптимизацию;
- автоматизированный способ, в котором оптимизацию выполняют «оптимизирующие компиляторы». В качестве примера можно привести компилятор «*VTune Amplifier*», который умеет анализировать код полностью и показывать время компиляции того или иного участка кода.

Пример скриншота программы приведен на рис.3.

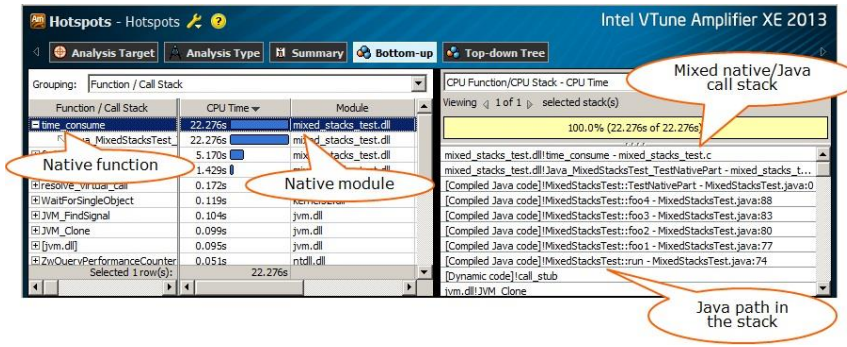


Рис. 3. Пример работы программы с компилятором «VTune Amplifier» [6].

Существуют несколько типов ручной оптимизации:

- *peephole*-оптимизация, суть которой состоит в том, что идет рассмотрение «инструкций», чтобы сделать вывод: можно ли сделать с ними какую-нибудь трансформацию для кодовой оптимизации [7];
- внутрипроцедурная оптимизация – это глобальная оптимизация, которая выполняется целиком и только в рамках одной единицы трансляции: функции. Данная оптимизация эффективнее, к примеру, чем локальная и позволяет достигать оптимизационных эффектов в разы больше, за счет больше затребованной информации и системных ресурсов, которые понадобятся для различных вычислений;
- локальная оптимизация охватывает рассмотрение информационный базовый блог, строго за один шаг. В самих базовых блоках нет переходов к потокам управления, что значит следующее: данная оптимизация экономит время и оперативную память, но теряет информацию к следующему шагу оптимизации;
- оптимизационные циклы (оптимизация циклов) – это одна из самых важных оптимизаций в самой программе, так как большие количество циклов замедляет как работу программы, так и компиляцию кода. Если объединять несколько циклов в один, то компилятору будет легче компилировать, за счет уменьшения вызовов функций и объема кода;
- межпроцедурная оптимизация. С помощью данного метода оптимизации можно анализировать сразу весь код, без определенного поиска нужных блоковых частей. Встраивание копии тела функций, с помощью данного метода, позволяет сэкономить системные ресурсы, которые связаны именно с вызовом функций.

Можно привести пример «выравнивания данных» с оптимизационными методами. Выравнивание данных – это способ выравнивания данных в оперативной памяти вычислительной техники (компьютера), который помогает разместить данные таким образом, что доступ к ним будет ускорен в несколько раз.

В качестве первого кода, возьмем структуру «*one*» и зададим в ней следующие параметры:

- *struct one*{
  char m; // целочисленный (символьный) тип данных
  long int p; // целочисленный тип данных «*p*»
  int n;
  };

данная структура занимает 24 байта. Узнать это можно с помощью вывода «*sizeof(A)*». А если логически поменять местами целочисленный тип данных «*p*» с «*n*», то получится следующая структура «*two*»:

- *struct two*{

char m; // целочисленный (символьный) тип данных
int n;
long int p; // целочисленный тип данных «*p*»
};

Таким образом, структура будет занимать уже 16 байт, а не 24. Связанно это с тем, что на 64-х битной операционной системе, считывание памяти происходит участками, которые равны 8 байтам. Пример считывания для данной программы:

Под первую структуру подсчет памяти будет вестись следующим образом:

1 байт выдается под структуру «*one*», остальные 7 байт остаются пустыми;

нужно добавить 8 байт под целочисленный тип данных «*p*»;
и под «*n*» выделяется 4 байта и 4 остаются пустыми.
(1+7) + 8 + (4+4) = 24 байта, что не совсем рационально.

Под вторую структуру подсчет будет вестись другим образом:

1 байт выдается под структуру «*two*», остальные 3 байта – пустые и нужно прибавить 4 байта под тип данных «*n*». Сокращение произошло за счет изменения структуры целочисленных типов данных;

Выделение 8 байт происходит для целочисленного типа данных «*p*».
(1+3+4) + 8 = 16 байт, что выглядит более рационально.

Данная оптимизация напрямую влияет на скорость компиляции кода. Результаты тестирования приведены в таблице 1.

Табл. 1 – Тестирование кодовых структур на скорость компиляции

Кодовая структура	Общая скорость компиляции, мс.	Параметр «*debug*», мс.	Параметр «*realese*», мс.
«*one*»	14	8	6
«*two*»	12	7	5

В качестве теста использовали программу для моделирования кинетики упорядочения бинарных сплавов по вакансионному механизму диффузии состава *AnBm* в модели твердых сфер [8].

Тестирование проводилось на машине (конфигурация):

- *CPU* 2 x *Intel Xeon* 5660 (в сумме 12 ядер / 24 потока по 2.8*GHz*, 12*Mb Cache*, 6.40 *GT/s*);
- *RAM* 32*Gb DDR*3-10600 *ECC REG*;
- *Intel S3710 SSDSC2BA012T401*.

Компилятор, на котором производилось тестирование: Microsoft Visual Studio 2010.

Таким образом, при смене логических выражений числовых типов данных, можно оптимизировать и ускорить структуру «two» и снизить объем потреблении оперативной памяти.

4. Заключение

В данной статье была рассмотрена статистика языков программирования по популярности, после обсуждения которой был выяснен тот факт, что язык программирования *C++* является одним из самых популярных языков программирования высокого уровня. После обсуждения статистики были рассмотрены базовые аспекты языка программирования *C++* с примерами, а также детально рассмотрены методики программирования на *C++*, которые позволяют сделать код более удобным и практичным. Приведены примеры оптимизирующих компиляторов, за счет которых можно оптимизировать код в автоматическом режиме, а также рассмотрены виды ручной оптимизации, за счет которых можно ускорить работу программы и компиляцию. Приведены примеры структурированного кода с заменой логических выражений различных типов данных для оптимизации потребления оперативной памяти персонального компьютера и ускорения компиляции программы с разными параметрами для двух структур.

Список литературы:

[1]. Материалы с ресурса HABRAHABR: http://habrahabr.ru/ [Электронный ресурс] // Хабрахабр

[2]. Стивен Прата. Язык программирования C++ - Вильямс, 2012.

[3]. Медведев В.И. Особенности объектно-ориентированного программирования на C++/CLI, C# и Java - РИЦ «Школа», 2010.

[4]. Литвиненко Н.А. Технология программирования на C++. Win32 API-приложения - РИЦ «Школа», 2010.

[5]. Страуструп Бьярн. Программирование. Принципы и практика использования C++ - «Вильямс», 2011.

[6]. Материалы с ресурсов INTEL: http://www.intel.com/ [Электронный ресурс] // INTEL

[7]. Каретин И.И., Макаров В.А. Энергосберегающая оптимизация кода за счет использования отключаемых компонентов процессора / Труды Института системного программирования РАН Том 19. 2010 г. Стр. 187-194.

[8]. Халиков А.Р., Искандаров А.М. Моделирование кинетики упорядочения бинарного сплава по вакансионному механизму диффузии в модели твердых сфер / Известия высших учебных заведений. Физика. 2012 г. №12. Стр. 87.

Technique the Instrumentation a Code and Optimization of Code Lines in Modeling Phase Transitions on the Programming Language C++

E.V. Palchevsky <teelxp@inbox.ru>
A.R. Khalikov <khalikov.albert.r@gmail.com>
Ufa State Aviation Technical University, 450000, Russia, Ufa, st. Marx, 12

Abstract. This article discusses the technique of writing code that you can use to save time to write a specific program, instrumenting code technique on high-level language C ++, examples Coding algorithms are provided for system software and reviewed optimized compilers, by which it is possible to reduce compile time and speed program work. The definitions of the programming language «C» and «C ++», code structure, code optimization. It is also considered the use of statistics to promote programming languages. The basic advantages «C ++» programming language, in comparison with the «C». Produced designation of source code structure for better usability. Examples of the implementation of the code of the algorithm for further work with the code strings. Examples of clear and concise written comments to the source lines. Implemented class constructors and destructors, which aim to detailed optimization and reduction of the use of block RAM in the modeling of phase transitions. The types of optimizations: manual and automatic. In manual optimization described a few paragraphs: peephole, vnutriprotsedurnaya, local, and interprocedural optimization cycles. A structured code with sequential replacement of logical expressions with well developed various types of data for further optimization of consumption of RAM PC. It is shown that work with specific functions and provides practical examples of usage. Manufactured memory consumption calculations before and after optimization of the code lines at various compilers and server hardware, the configuration of which is also given in this article.

Keywords: code; code optimization; code instrumentation; techniques of writing code; C++; programming.

DOI: 10.15514/ISPRAS-2015-27(6)-6

For citation: Palchevsky E.V., Khalikov A.R. Technique the Instrumentation a Code and Optimization of Code Lines in Modeling Phase Transitions on the Programming Language C++. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 87-96 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-6

References

[1]. Materials with resource HABRAHABR: http://habrahabr.ru/ [Electronic resource] // Habrahabr

[2]. Steven Pratt. The programming language C++ - Williams 2012.

[3]. Medvedev V.I. Features of object-oriented programming in C++ / CLI, C# and Java - RIP "School", 2010.

[4]. Litvinenko N.A. Programming Technology in C++. Win32 API-application - RIP "School", 2010.

[5]. Bjarne Stroustrup 5. Programming. Principles and practice of using C ++ - «Williams» 2011.

[6]. Materials with resources INTEL: http://www.intel.com/ [Electronic resource] // INTEL

[7]. Karetin I.I., Makarov V.A. Jenergosberegajushhaja optimizacija koda za schet ispol'zovanija otkljuchaemyh komponentov processora [Energy optimization of code by using disconnected components CPU] / Trudy ISP RAN [The Proceedings of ISP RAS], Volume 19, 2010, pp. 187-194. (in Russian)

[8]. Khalikov A.R., Iskandarov A.M. . Modelirovanie kinetiki uporjadochenija binarnogo splava po vakansionnomu mehanizmu diffuzii v modeli tverdyh sfer [Modeling the kinetics of ordering of the binary alloy of the vacancy mechanism of diffusion in the model of hard spheres] / Izvestija vysshih uchebnyh zavedenij. Fizika [Proceedings of the higher educational institutions. Physics]. 2012, №12. Pp. 87. (in Russian)

Разработка и реализация метода масштабирования по памяти для систем межмодульных оптимизаций и статического анализа на основе LLVM

Долгорукова К.Ю. <unerkannt@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. Проблема масштабируемости систем оптимизации времени связывания и систем статического анализа не потеряла своей актуальности в настоящее время: несмотря на рост производительности и увеличение объема памяти компьютеров, программы растут в размерах и сложности пропорэ иона льно, особенно когда дело касается таких сложных многомодульных программ, как, например, операционные системы, браузеры и другие. Эффективная оптимизация таких программ с использованием таких мощных инструментов, как межпроцедурные оптимизирующие преобразования, проводимые во время связывания, и преобразования с использованием профиля исполнения программы, требует существенных вычислительных ресурсов. В статье рассматривается подход к масштабированию по памяти системы оптимизаций времени связывания в целях ограничения потреб ляемЙ й памяти заданным пороговым значением. Предложенный метод включает в себя следующие этапы: аннотирование промежуточного представления на этапе генерации промежуточного кода; во время компоновки чтение объявлений и аннотаций из файлов с промежуточным представлением, предварительный анализ, в котором происходит построение и анализ графа вызовов, отложенную загрузку участков кода во время оптимизаций и выгрузку участков кода по требованию. Также предложен подход к применению масштабирования по памяти к системе статического анали за. ОЙ исанный метод масштабирования был реализован на основе инструмента GOLD-plugin системы LLVM[1]. Представленные предварительные результаты тестирования реализации данного подхода на тестах SPEC CPU2000[2] показывают увеличение размера кода на 6%, увеличение накладных расходов по времени на 0.2% и по памяти на 36%.Ключевые слова: межмодульные анализ и оптимизации; системы межмодульных оптимизаций; масштабирование.

DOI: 10.15514/ISPRAS-2015-27(6)-7

Для цитирования: Долгорукова К.Ю. Разработка и реализация метода масштабирования по памяти для систем межмодульных оптимизаций и статического анализа на основе LLVM. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 97-110. DOI: 10.15514/ISPRAS-2015-27(6)-7.

1. Введение

Традиционный метод сборки программ из исходного кода состоит из двух этапов: компиляции и связывания. Обычно все модули программы компилируются отдельно и независимо в объектные файлы, которые потом связываются компоновщиком. В случае раздельной независимой оптимизации у компилятора нет программного кода других модулей, поэтому эффективность межпроцедурных оптимизаций ограничена одним файлом с исходным кодом. Если же есть возможность оптимизировать все входящие в программу модули вместе, эффективность этих оптимизаций значительно возрастает. Оптимизации, проводимые на всей программе целиком, называются межмодульными оптимизациями.

На втором этапе сборки компоновщик получает все скомпилированные файлы программы и файлы библиотек, разрешает коллизии и зависимости между ними и строит исполняемый файл. Практика показала, что целесообразно проводить межмодульные оптимизации на этапе связывания, так как именно на этом этапе системе сборки доступна вся программа целиком: для этого в объектные файлы программ, полученные на этапе компиляции, добавляется некоторая информация о программе, достаточная, чтобы построить промежуточное представление программы, используемое компилятором. Такие системы называются системами оптимизаций времени связывания.

Межмодульные оптимизирующие преобразования, проводимые над всеми модулями программы, всегда связаны с построением промежуточного представления для всей программы. В случае больших приложений этот процесс может потребовать огромных ресурсов. Для сборки с оптимизациями времени связывания таких программ, как операционные системы или интернет-браузеры, состоящих из нескольких тысяч файлов исходного кода, может требоваться память, которой не обладают не только обычные настольные компьютеры, − но и далеко не все серверные архитектуры способны собрать такие программы без задействования отгрузки на диск.

Методы регулирования потребления ресурсов системой для различных архитектур называются масштабированием системы.

Масштабирование может проводиться как по времени, так и по памяти, ускоряя процесс сборки – или уменьшая количество потребляемой памяти. Ускорение сборки может проводиться как посредством ручной оптимизации существующих решений, так и посредством распараллеливания для запуска сборки на нескольких ядрах. Уменьшить количество потребляемой памяти можно также с помощью ручной оптимизации, то есть посредством поиска ошибок в использовании памяти или усовершенствования алгоритмов, либо с помощью механизмов управления потреблением ресурсов.

Существует несколько примеров компиляторных инструментов, которые дают возможность так или иначе управлять потреблением ресурсов во время проведения оптимизаций времени связывания, в том числе GCC[3], компиляторы HP[4,5], Google’s LIPO[6] и другие; подробно принципы их

работы описаны в статье [7]. Для большинства из них применяется одна схема работы, включающая в себя 3 этапа: генерацию промежуточного кода (ГПК), стадию межпроцедурного анализа и стадию генерацию машинного кода (ГМК).

Во время генерации промежуточного кода компилятор получает файлы с кодом на исходном языке программирования, проводит небольшое количество локальных оптимизаций в каждом файле отдельно, а также генерирует дополнительную информацию о зависимостях между функциями в так называемые аннотации, представляющие собой некую информацию о программе, необходимую для проведения межпроцедурного анализа. В результате работы первого этапа генерируется расширенный файл в промежуточном представлении (либо в двоичном формате), содержащий также дополнительную информацию. Обычно этап ГПК легко параллелизуем.

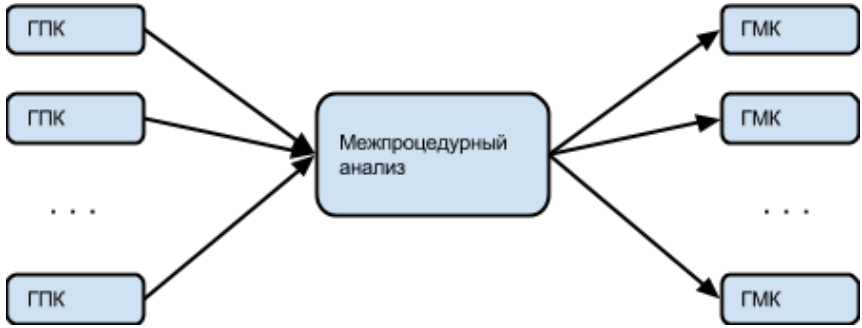


Рис. 1. Схема работы систем оптимизаций времени связывания

Второй этап, как правило, проходит во время связывания и читает файлы, полученные на первом этапе, анализирует межпроцедурные зависимости, производит некоторые оптимизирующие преобразования, и снова генерирует файлы в промежуточном представлении и некоторое количество дополнительной информации. Этот процесс трудно параллелизуем, и все компиляторные системы с открытым исходным кодом выполняют межмодульный анализ в один поток, поэтому именно анализ – узкое место этих систем.

Последняя фаза принимает файлы, сгенерированные на втором этапе и генерирует из них объектный код, который затем подается ассемблеру. Иногда на этой стадии проводятся дополнительные оптимизации. Эта фаза у разных компиляторов может быть параллелизуемой или непараллелизуемой, и в результате может получаться либо один объектный файл, либо несколько, которые впоследствии передаются стандартному системному компоновщику.

2. Особенности сборки программ в системе LLVM

Система LLVM – это набор библиотек и компиляторных утилит для анализа и оптимизации кода. В основе LLVM лежит его характерное промежуточное представление – похожий на ассемблер язык с трёхадресными инструкциями, находящимися в SSA-форме. Файлы, содержащие код на языке LLVM, называются биткодом.

Когда пользователь собирает программу на языке Си или Си++ утилитами LLVM с оптимизациями времени связывания, он должен вызвать компилятор CLANG с опцией -flto, например, так:

clang -flto file1.c -file2.c … -fileN.c -o program.exe

В этом вызове прозрачно для пользователя происходит следующее: сначала каждый файл программы транслируются компиляторами CLANG в промежуточное представление LLVM и сохраняется в файлы с биткодом, затем вызывается компоновщик LD-GOLD[8], который считывает сгенерированные посредством CLANG файлы с биткодом и передает их на обработку плагину llvm-gold-plugin.

Каждый считанный плагином файл преобразуется в единицу абстракции промежуточного представления, называемую модулем. Модуль (Module) состоит из набора объявлений и определений глобальных переменных, тел функций, констант, набора синонимов и дополнительной информации, необходимой для некоторых оптимизаций, или мета-данных (MetaData). Функции, в свою очередь, состоят из базовых блоков, а последние, в свою очередь, – из инструкций.

Библиотека оптимизаций времени связывания (LTO – Link-Time Optimization) позволяет создавать так называемый LTO-модуль – структуру, в которую происходит чтение символов файла. Для запуска оптимизаций времени связывания также необходим объект генератора кода (LTO Code Generator), который производит анализ и оптимизации скомпонованного модуля, а затем генерирует объектный код.

Во время сборки программы компоновщик передает их по одному на обработку плагину llvm-gold. Плагин создает для текущего файла объект LTO-модуль, читает и переводит во внутреннее представление содержимое файлов, затем передает LTO-модуль компоновщику модулей, а тот, в свою очередь, компонует его в хранящийся в памяти композитный модуль, разрешая коллизии типов и составляя связи с новым участком кода. После того как все файлы оказываются обработаны, компоновщик посылает плагину запрос на генерацию кода. Тогда генератор кода проверяет композитный модуль на правильность, запускает на нем последовательно анализ и оптимизации, и, наконец, генерирует объектный код.

Анализ и оптимизации в LLVM реализованы в виде проходов компилятора. Проходы разделены соответственно абстракциям промежуточного представления, на которых они могут производиться независимо. Существуют

проходы, исполняемые на модуле, на функции, на базовых блоках. Помимо этого, есть проходы, выполняемые на сгенерированных структурах: на графе вызовов функций, на циклах. Также есть постоянные проходы, которые исполняются в самом начале работы компилятора на всём модуле. Для управления порядком запуска компиляторных проходов в LLVM применяется стек менеджеров проходов, каждый из которых выстраивает порядок пассов согласно зависимостям между ними. В стеке менеджеры разделяются по типу проходов, которыми они управляют, и для каждого уровня абстракции есть свой менеджер. Сами менеджеры также являются проходами. Когда компилятор запускает оптимизации, он начинает оптимизировать с самой верхней абстракции – модуля. Таким образом, сначала запускаются модульные проходы, среди которых есть менеджер следующих абстракций – функций и сильносвязных компонент графа. Менеджеры компонент графа, в свою очередь, запускают проходы, работающие на сильносвязных компонентах, а менеджеры функций – проходы, работающие на телах функций, а также менеджеры циклов, – и так далее вплоть до инструкций.

При работе оптимизаций всегда неявно подразумевается, что модуль полностью сформирован в памяти, в нем присутствуют все необходимые объявления и тела функций, разрешены все коллизии, сформированы все зависимости. Таким образом, пиком потребления памяти при сборке программы с помощью утилит LLVM будет именно стадия оптимизаций.

В задачу масштабирования, таким образом, входят:

- разделение фазы анализа и фазы оптимизаций;
- разработка метода распараллеливания межпроцедурных оптимизаций;
- разработка метода ограничения потребляемой памяти с учетом желания пользователя или согласно ограничениям оборудования, на котором производится сборка программы.

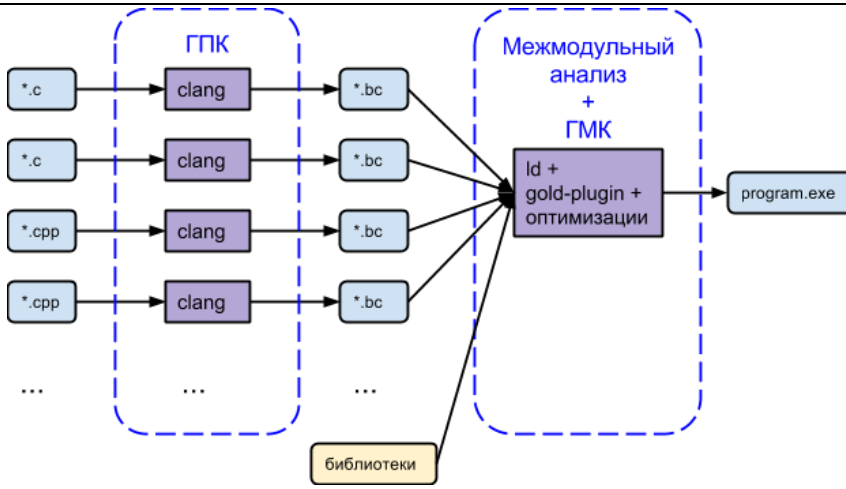


Рис. 2. Схема сборки программы с помощью утилит LLVM

3. Управление потреблением памяти

Существует несколько основных подходов к регулированию потребления памяти, реализованных в разных компиляторах [3]. Один из них, предложенный разработчиками HP в системе HLO (High-Level Optimization) для компилятора HP-UX, использует диспетчеризацию ресурсов, в которой отслеживается доступ к участкам кода и потребление памяти. При достижении указанного пользователем порога занимаемой памяти компилятор начинает отгружать давно неиспользуемые участки промежуточного представления кода на диск. Когда компилятор пытается получить доступ к отгруженным участкам кода, диспетчер подгружает их в память. Данный подход позволяет выполнять оптимизации с произвольным ограничением ресурсов, но платой за такую гибкость будет время: выгрузка кода на жесткий диск – очень тяжеловесный процесс, требующий доступа к более медленной памяти, а также операций архивирования и разархивирования кода.

Другой подход предлагают разработчики GCC в рамках проекта WHOPR (WHOle-PRogram optimization – оптимизация программы целиком). Межмодульные анализ и оптимизации строго разделены на разные стадии, причем, стадия оптимизаций может быть выполнена в несколько потоков. Для того, чтобы иметь возможность выполнять оптимизации параллельно, на стадии анализа обеспечивается независимость файлов друг от друга при возможности использовать информацию из других модулей.

3.1 Предлагаемый подход к управлению памятью для систем межмодульных оптимизаций

Авторами статьи был учтен опыт существующих решений, и предложен комбинированный подход к управлению использованием ресурсов. Наш метод условно разделяет процесс сборки на две стадии:

1. Загрузка информации о глобальных структурах и данных для всех компонуемых модулей: данные о типах, описание функций, описание типов, глобальные переменные. На основе этих данных можно проводить некоторый легковесный анализ, например, строить граф вызовов. При этом доступ к коду функций на данной стадии ограничен только глобальной метаинформацией, поставляемой с кодом. К самим телам функций доступа нет.
2. Загрузка кода тел функций и оптимизирующие преобразования над ним. Данный этап допускает любые оптимизирующие преобразования. Загрузка функций осуществляется последовательно, по мере запроса алгоритмов оптимизации. Диспетчер памяти следит за потреблением ресурсов, и, когда размер занятой памяти переходит заданный пользователем порог, отгружает неиспользуемый код на диск.

Таким образом, с одной стороны, достигается гибкость в управлении памятью, а с другой, представляется возможность производить часть анализа программы вовсе без загрузки кода в память.

3.2 Предлагаемый подход к управлению памятью для систем статического анализа

Для статического анализа кода был предложен несколько иной подход, основанный на обходе графа. Для этого предполагается использовать граф вызовов в качестве структуры для задания порядка обхода кода программ. Для того, чтобы иметь граф вызовов еще до момента загрузки кода, необходимо на этапе генерации промежуточного кода встраивать метаинформацию о вызовах функций. Получить доступ к графу вызовов можно с помощью интерфейса прикладного программирования (API – application programming interface). По мере завершения работы над анализом функции, код функции можно выгрузить, вызвав соответствующую функцию. Таким образом, интерфейс прикладного программирования для статического анализа представлен методами:

- получения доступа к графу вызовов;
- загрузки тела функции;
- выгрузки тела функции.

4. Реализация метода ленивой загрузки в системе LLVM

Чтобы иметь возможность ограничить потребление ресурсов компоновщиком, необходимы ключевые изменения в стратегии оптимизации, чтения и компоновки файлов с биткодом в модуль промежуточного представления. В общем случае необходимо отделение стадии анализа от стадии оптимизаций, а также реализация анализа, способного работать лишь на описаниях кода, не имея доступа к самому коду компилируемой программы. Также необходим механизм разделения файлов с биткодом на группы, которые можно оптимизировать независимо, без доступа к телам функций.

Для реализации ограничения потребления ресурсов был внедрен механизм загрузки тел функций в память по запросу. Для этого в механизме чтения биткода из файла были реализованы пропуски тел функций, а компоновщик модулей допускал подобные пропуски, помечая пропущенные функции, как «встреченные». Тел у встреченных функций нет, но они отличаются от объявлений тем, что могут быть загружены в любой момент по требованию какой-либо из оптимизаций. Также смещения тел встреченных функций сохраняются в специальном журнале для более быстрого доступа к ним при повторном чтении файла.

Таким образом, загрузка файла с биткодом состоит из двух этапов: загрузка объявлений, глобальных переменных и констант с предварительной их компоновкой в композитный модуль, и загрузка тел функций с окончательной компоновкой.

Специально для возможности загрузки тела функции из очереди оптимизирующих проходов был реализован дополнительный проход ленивой загрузки. Каждый оптимизирующий или анализирующий компиляторный проход, требующий доступа к телам функций, должен запросить при инициализации проход-загрузчик, чтобы менеджер проходов инициировал ленивую загрузку раньше, чем запуск прохода, требующего тело функции.

Когда проход-загрузчик инициирует загрузку тела функции, в работу вступает менеджер ленивой загрузки, который достает из журнала информацию о функции, такую как: файл, где лежит тело функции, смещение, размер кода и таблицы глобальных значений и типов исходного модуля, которые также были сохранены во время предварительной загрузки. Затем менеджер запускает загрузчик кода, который считывает участок файла с кодом в буфер и преобразует его в промежуточное представление. После этого менеджер ленивой загрузки вновь запускает компоновщик, который разрешает и уточняет типы загруженных переменных и копирует их в композитный модуль.

После всех этих действий код функции готов к оптимизациям.

5. Реализация легковесного построения графа вызовов

В библиотеке LLVM есть реализация графа вызовов, широко используемая в оптимизациях, но использовать её вместе с ленивой загрузкой оказалось невозможно: эта реализация тесно привязана к инструкциям вызова, что, во-первых, не нужно при анализе зависимостей между файлами, а во-вторых, невозможно обеспечить в отсутствие кода тел функций. Чтобы всё-таки получить граф вызовов без доступа к телам функций, в механизме записи биткода в файлы, работающего на этапе генерации промежуточного представления, были реализованы запись информации о вызовах перед телами функций. Для чтения информации были произведены соответствующие изменения в механизме чтения биткода. Построенный таким образом граф вызовов содержит информацию о факте вызова функции, но не предоставляет информации, например, о параметрах вызова. Тем не менее, имея такой граф, возможно проанализировать связи между модулями, а также строить ленивый диспетчер оптимизаций, итерирующийся не последовательно по функциям, а методом обхода вглубь дерева вызовов. Этот диспетчер можно использовать как при оптимизациях в условиях нехватки ресурсов, так и для ресурсоемкого анализа, например, статического анализа с поиском ошибок или анализа сущностей и связей программ для понимания кода [8].

Рисунки 3 и 4 показывает сравнение двух графов одной программы: первый получен в результате обычного построения графа через обход инструкций вызова, а второй построен посредством аннотаций без доступа к инструкциям. Как можно видеть из рисунка, графы различаются наличием источника и стока, рёбра от и к которым добавляются автоматически всем узлам при построении графа в LLVM.

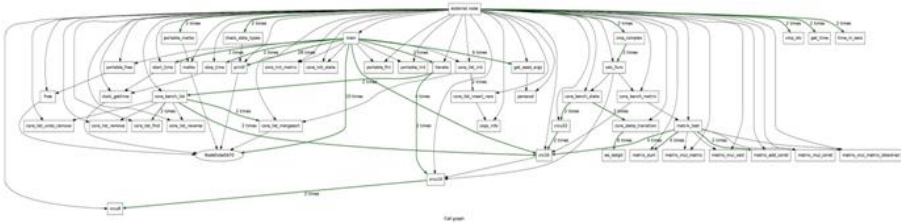


Рис. 3. Граф вызовов, построенный стандартным проходом построения графа вызовов в LLVM

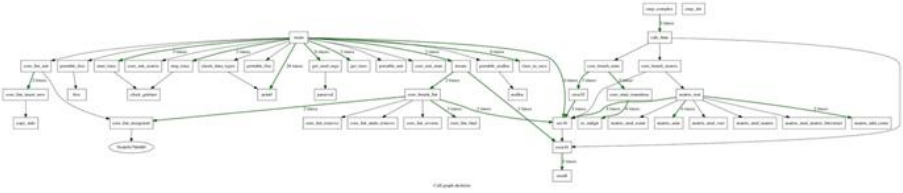


Рис. 4. Граф вызовов, построенный только с помощью аннотаций

6. Результаты работы

В ходе работы были разработаны и реализованы:

- метод ленивой загрузки кода в два этапа: предварительное чтение и чтение тел функций;
- метод выгрузки функций;
- метод легковесного получения графа вызовов без доступа к инструкциям;
- интерфейс прикладного программирования для систем статического анализа.

Существующее решение было протестировано на некоторых тестах SPEC CPU2000 на архитектуре x86_64 Intel Core i5-2500 с 32ГБ оперативной памяти. Таблица 1 показывает, насколько увеличивается объем, занимаемый файлами с промежуточным представлением, когда в него записываются аннотации.

Табл. 1. Увеличение объема биткода от аннотаций

Название теста	**Размер файлов с биткодом в байтах при обычной сборке**	**Размер файлов с биткодом в байтах при сборке с аннотациями**	**Увеличение размера кода в процентах**
gcc	2797940	2970212	6.16
gzip	89156	96808	8.58
vpr	303208	318008	4.88
mcf	43863	45656	4.09

crafty	571744	583700	2.09
parser	312748	377888	20.83
bzip2	86516	92292	6.68
Суммарное значение	**4205175**	**4484564**	**6.64**

Исходя из результатов таблицы 1, размер биткода увеличился в среднем на 6,6%. При этом размер аннотаций зависит от количества функций и вызовов в программе. Для программ с большим количеством маленьких функций относительный объем аннотаций может достигать 20%, как это видно на тесте parser, тогда как для программы с несколькими крупными функциями этот показатель составляет всего 2%.

Табл. 2. Временные издержки на ленивую загрузку функций без выгрузки.

Название теста	**Время сборки в секундах**	**Время ленивой сборки в секундах**	**Увеличение времени в процентах**
gzip	0.2993	0.3001	0.27
vpr	0.9942	0.996	0.18
gcc	13.0642	13.0932	0.22
mcf	0.1164	0.1174	0.86
crafty	1.4256	1.4185	-0.50
parser	1.0047	1.0176	1.28
bzip2	0.3827	0.385	0.60
Суммарное значение	**17.29**	**17.33**	**0.24**

Время сборки программ, как видно из таблицы 2, от ленивой загрузки, отложенной до момента оптимизаций, практически не страдает, и увеличивается всего на 0,2%. Тестирование сборки проводилось по 10 раз для каждого теста и было усреднено.

Табл. 3. Накладные расходы на ленивую загрузку

Название теста	**Пик потребления при обычной сборке, Кб**	**Пик потребления при ленивой сборке, Кб**	**Накладные расходы, %**	**Среднее потребление при обычной сборке, Кб**	**Среднее потребление при ленивой сборке, Кб**	**Накладные расходы, %**
gzip	324	404	24.69	177	269	51.98
vpr	18720	21652	15.66	4484	4765	6.27
gcc	168992	212468	25.73	94811	132811	40.08
mcf	n/a	n/a	n/a	n/a	n/a	n/a
crafty	23328	28700	23.03	6287	7319	16.41
parser	20296	25264	24.48	4924	5742	16.61
bzip2	n/a	n/a	n/a	n/a	n/a	n/a
Сум-марное значение	**231660**	**288488**	**24.53**	**110683**	**150906**	**36.34**

Таблица 3 показывает накладные расходы по памяти на поддержание журнала для ленивой загрузки. Тест проводился при выключенной отгрузке кода функций, таким образом, чтобы замерить объемы журнала. Тестирование проводилось с помощью утилиты vmstat с частотой 1 с. При учете, что большинство программ компоновались всего несколько секунд, тестирование не исключает погрешность. Мало того, для двух тестов не удалось получить данных ввиду очень малого времени сборки.

7. Заключение

В результате работы была получен работоспособный прототип системы, на основе которого возможно разрабатывать межмодульные анализирующие и оптимизирующие проходы, работающие только на аннотациях, а также строить системы статического анализа. Тем не менее, экспериментальные данные показали, что необходим менеджер ресурсов, который смог бы отслеживать необходимости выгрузки функций по достижению

определенного пользователем порога потребления памяти. Также планируется оптимизировать способ хранения структур в журнале с целью уменьшения накладных расходов.

Список литературы

[1]. The LLVM Compiler Infrastructure. http://LLVM.org/
[2]. SPEC CPU benchmark. https://www.spec.org/cpu2000/
[3]. Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, Ollie Wild. WHOPR - Fast and Scalable Whole Program Optimizations in GCC. Initial Draft, 12 Dec 2007.
[4]. Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable crossmodule optimization. SIGPLAN Not., 33(5):301–312, 1998. ISSN 0362-1340. doi:http://doi.acm.org/10.1145/277652.277745.
[5]. Sungdo Moon, Xinliang D. Li, Robert Hundt, Dhruva R. Chakrabarti, Luis A. Lozano, Uma Srinivasan, and Shin-Ming Liu. SYZYGY - a framework for scalable cross-module IPO. In CGO '04: Proceedings of the international symposium on Code generation and optimization, page 65, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9
[6]. Xinliang David Li, Raksit Ashok, Robert Hundt. Lightweight Feedback-Directed CrossModule Optimization. CGO'10, April 24–28, 2010, Toronto, Ontario, Canada. ACM 978-1-60558-635-9/10/04.
[7]. К.Ю. Долгорукова. Обзор масштабируемых систем межмодульных оптимизаций. Труды Института системного программирования РАН Том 26, выпуск 3, 2014, стр. 69-90. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2014-26(3)-3
[8]. А.А. Белеванцев, Е.А. Велесевич. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ. Труды Института системного программирования РАН. Том 27, выпуск 2, 2015, стр. 53-64. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2015-27(2)-4
[9]. Пакет GNU Binutils. http://www.gnu.org/software/binutils/

Implementation of Memory Scalability Approach for LLVM-Based Link-Time Optimization and Static Analyzing Systems

Ksenia Dolgorukova <unerkannt@ispras.ru>
Institute for System Programming of the RAS,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. Link-time optimization and static analyzing systems scalability problem is of current importance: in spite of growth of performance and memory volume of modern computers programs grow in size and complexity as much. In particular, this is actual for such complex and large programs as browsers, operating systems, etc. To improve performance of these programs as much as possible, there are several aggressive optimising

techniques like interprocedural optim ization a nd profile-guided optimization. These techniques applied to large programs claim for large memory and need a lot of time to be performed. This paper introduces memory scalability approach for link-time optimization system and proposes technique for applying this approach to static analyzing systems. The approach involve several steps: adding a summary information to intermediate representation at compile time, reading declarations and summaries from IR files, analysing summary and computing call graph at special pre-analysis phase, lazy code loading during optimization phase and code unloading on demand. Proposed approach was implemented as the linking tool based on the LLVM GOLD-plugin. The tool was tested on SPEC CPU2000 benchmark suite. Preliminary results show increasing of average intermediate code on 6%, increasing of average time on 0.2% and increasing of total memory usage to 36%.

Keywords: link-time optimization; cross-module optimization systems; scalability.

DOI: 10.15514/ISPRAS-2015-27(6)-7

For citation: Dolgorukova Ksenia. Implementation of Memory Scalability Approach for LLVM-Based Link-Time Optimization and Static Analyzing Systems. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 6, 2015, pp. 173-192 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-7.

References

[1]. The LLVM Compiler Infrastructure. http://LLVM.org/

[2]. SPEC CPU benchmark. https://www.spec.org/cpu2000/

[3]. Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, Ollie Wild. WHOPR - Fast and Scalable Whole Program Optimizations in GCC. Initial Draft, 12 Dec 2007.

[4]. Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable crossmodule optimization. SIGPLAN Not., 33(5):301–312, 1998. ISSN 0362-1340. doi:http://doi.acm.org/10.1145/277652.277745.

[5]. Sungdo Moon, Xinliang D. Li, Robert Hundt, Dhruva R. Chakrabarti, Luis A. Lozano, Uma Srinivasan, and Shin-Ming Liu. SYZYGY - a framework for scalable cross-module IPO. In CGO '04: Proceedings of the international symposium on Code generation and optimization, page 65, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9

[6]. Xinliang David Li, Raksit Ashok, Robert Hundt. Lightweight Feedback-Directed CrossModule Optimization. CGO'10, April 24–28, 2010, Toronto, Ontario, Canada. ACM 978-1-60558-635-9/10/04.

[7]. Ksenia Dolgorukova. Obzor masshtabiruemyh sistem mezhmodul'nyh optimizacij [Overview of Scalable Frameworks of Cross-Module Optimization]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, issue 3, 2014, pp. 69-90. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2014-26(3)-3. (in Russian)

[8]. A. Belevantsev, E. Velesevich. Analiz sushhnostej programm na jazykah C/C++ i svjazej mezhdu nimi dlja ponimanija programm [Analyzing C/C++ code entities and relations for program understanding]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 27, issue 2, 2015, pp. 53-64. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2015-27(2)-4. (in Russian)

[9]. GNU Binutils. http://www.gnu.org/software/binutils/

Статический анализатор Svace как коллекция анализаторов разных уровней сложности

[1]А.Е. Бородин <alexey.borodin@ispras.ru >
[1,2]А.А. Белеванцев <abel@ispras.ru>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1., стр. 52, факультет ВМК

Аннотация. В статье описывается статический анализатор Svace, разрабатываемый в ИСП РАН.

Текущая версия анализатора осуществляет поиск ошибок в программах, написанных на языках Си, Си++, Java и Си#. Svace осуществляет поиск дефектов различных типов, включая ошибки разыменования нулевого указателя, переполнение буфера, использование неинициализированных переменных, утечки памяти, двойные блокировки, наличие недостижимого кода, несогласованность конструкторов и деструкторов, ошибки деления на ноль, возвращение адреса локальных переменных, использование объектов после удаления. Целью анализа является поиск как можно большего количества дефектов при приемлемом количестве ложных срабатываний и времени анализа.

Дефекты в программе имеют разную природу и для их поиска необходимо правильно выбрать алгоритм анализа. Хороший инструмент будет включать в себя как простые детекторы, использующие синтаксический анализ, так и сложные детекторы, позволяющие найти нетривиальные межпроцедурные ошибки. Построение инструмента на основе нескольких анализаторов позволяет использовать преимущества этих видов анализаторов и находить больший диапазон ошибочных ситуаций.

Инструмент Svace состоит из набора анализаторов реализующих анализы разных типов: анализ на основе абстрактного синтаксического дерева, консервативный анализ потока данных для одной функции, потоко- чувствительный и межпроцедурный неконсервативный анализ с возможностью использовать чувствительность к путям.

Межпроцедурный анализ осуществляется на основе аннотаций. При таком подходе после анализа функции создаётся её аннотация, описывающая интересующие эффекты вызова функции. Аннотация используется при обработке вызова функции для эмуляции вызова, что позволяет избежать повторного анализа функции. На основе анализа отдельных функций реализован анализ пар конструкторов и деструкторов Си++, позволяющий находить несогласованность при их написании. Все описываемые

анализы иллюстрируются примерами ошибок, найденными анализатором на проектах с открытым исходным кодом.

Ключевые слова: статический анализ; язык Си; дефекты в исходном коде; абстрактное синтаксическое дерево; потоковая чувствительность; межпроцедурный анализ; чувствительность к путям; неконсервативный анализ; разыменование нулевого указателя; утечки памяти.

DOI: 10.15514/ISPRAS-2015-27(6)-8

Для цитирования: Бородин А.Е., Белеванцев А.А.. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8.

1. Введение

В современном обществе программное обеспечение используется повсеместно: в телефонах и компьютерах, в медицинском оборудовании, в банках, в самолётах и автомобилях, в детских игрушках и электростанциях. Зависимость от программного обеспечения повышается с каждым днём, а сами программы становятся сложнее и больше. Было показано, что плотность ошибок также растёт с размером программного обеспечения [1]. Многие ошибки могут приводить к катастрофическим последствиям. Поэтому при разработке программного обеспечения поиску и исправлению ошибок уделяют особое внимание.

Для поиска ошибок используют различные методики и инструменты: ручное и автоматическое тестирование, инструменты статического анализа, динамический анализ и др. Жизненный цикл разработки программ в крупных компаниях обязательно включает в себя применение инструментов статического анализа [2, 3]. Дополнительные требования к использованию статического анализа регламентировано регуляторами как в России (Роскомнадзор, ФСБ России [4]), так и в других странах.

Статические анализаторы осуществляют поиск ошибок в программе без её фактического запуска. При этом, как правило, сразу анализируется множество путей исполнения. Благодаря тому, что статические анализаторы просматривают сразу все пути исполнения и анализируют пути независимо от вероятности их выполнения, анализаторы находят многие ошибки на редко исполняемых путях, которые часто остаются незамеченными во время тестирования. Особенно это касается кода, обрабатывающего ошибочные ситуации. Другим преимуществом статических анализаторов является диагностика места ошибки. При выдаче предупреждения об ошибке статические анализаторы сразу показывают место ошибки и описывают, почему именно это является ошибкой.

В данной статье будут рассмотрены различные подходы для поиска ошибок с помощью статического анализа. Описание статических анализаторов будет

приведено на примере инструмента Svace, разрабатываемого в ИСП РАН. В настоящее время инструмент содержит несколько видов анализаторов различной сложности и способен осуществлять поиск ошибок в программах, написанных на языках C, C++, Java и C#. Svace осуществляет поиск дефектов различных типов, включая ошибки разыменования нулевого указателя, переполнение буфера, использование неинициализированных переменных, утечки памяти и других ресурсов, двойные блокировки и тупики, наличие недостижимого кода, несогласованность конструкторов и деструкторов, ошибки деления на ноль, возвращение адреса локальных переменных, использование объектов после удаления и другие.

Примеры разных видов анализаторов будут иллюстрироваться ошибками в этих проектах, найденными с помощью Svace. С помощью Svace был проанализирован исходный код операционной систем tizen и android, а также 30 проектов, написанных на языках Си и Си++ с открытым исходным кодом. Большинство проанализированных проектов существуют уже много лет, их регулярно тестируют, проверяют с помощью статических и динамических анализаторов и исправляют найденные ошибки. Тем не менее с помощью Svace удалось обнаружить дефекты в исходном коде этих проектов.

2. Использование статического анализатора в жизненном цикле разработки программ

Статический анализатор осуществляет поиск ошибок в программе без её фактического запуска. После анализа выдаётся список предупреждений о дефектах в исходном коде программы. Предупреждение называют истинным, если оно соответствует дефекту в исходном коде, т.е. описывает ситуацию, которую желательно исправить. Если предупреждение не описывает некоторый дефект, то его называют ложным.

Идеальный анализатор после анализа программы за приемлемое время найдёт все дефекты в программе и не выдаст ни одного ложного срабатывания. К сожалению, создание такого анализатора невозможно, что следует из теоремы Райса, согласно которой для любого нетривиального свойства, определение того, вычисляет ли произвольный алгоритм функцию с таким свойством, является алгоритмически неразрешимой проблемой. Свойство считается нетривиальным, если существуют функции обладающие этим свойством и существуют функции не обладающие этим свойством. Поэтому проблема поиска дефектов в произвольной программе является алгоритмически неразрешимой задачей, т.к. содержит ли программа некоторую ошибку является нетривиальным свойством. Из-за проблемы неразрешимости для создания алгоритма, осуществляющего поиск ошибок, необходимо пожертвовать возможностью поиска всех ошибок, либо выдачей только истинных срабатываний, либо ограничить класс анализируемых программ. На практике также важной характеристикой любого алгоритма является время его работы. Реализация конкретного анализатора является компромиссом

между количеством выдаваемых истинных предупреждений, количеством ложных предупреждений и временем анализа.

Часть инструментов статического анализа используются для проверки программного обеспечения с высокими требованиями к надежности и безопасности. Для таких анализаторов важным требованием является поиск всех потенциальных ошибок. Такие анализаторы могут выдавать множество ложных срабатываний, а также принимать в качестве входа не все возможные программы.

Если анализатор выдаёт предупреждения со слишком большим количеством ложных предупреждений, то его никто не будет использовать. В результате даже истинные предупреждения не будут просмотрены, т.к. необходимо затратить много времени на фильтрацию ложных предупреждений. Более того, если первые 3 просмотренные предупреждения оказываются ложными, то пользователи часто вообще отказываются просматривать остальные предупреждения [5]. На практике более важным является не поиск всех дефектов, а исправление как можно большего количества дефектов, а также стоимость исправления каждого дефекта. Стоимость будет пропорциональна количеству ложных срабатываний, которые необходимо просмотреть на каждое истинное срабатывание. Поэтому важным требованием к анализаторам является небольшая доля выданных ложных срабатываний.

Использовании статического анализатора в жизненном цикле разработки программ имеет свои особенности. Как правило, нет задачи найти все возможные дефекты в программе. Анализ проводится регулярно во время ночной сборки программы, поэтому время анализа ограничено примерно 12 часами. Количество ложных срабатываний должно быть достаточно низким, т.к. каждое ложное срабатывание отнимает время программиста и фактически выражается в убытках для компании. Для удовлетворения этих требований анализатор может в случае, когда нет достаточных оснований, что предупреждение будет истинным, просто не выдавать предупреждение. Это позволяет существенно уменьшить количество выдаваемых ложных срабатываний и в ряде случаев сократить время анализа. Такой анализатор может пропустить реальную ошибку, но, т.к. большинство выдаваемых предупреждений будут истинными, то увеличивается шанс, что каждый реально найденный дефект будет исправлен, и уменьшается стоимость каждого исправленного дефекта.

Долю истинных срабатываний в 60-70% можно считать приемлемой. При таком соотношении истинных и ложных срабатываний на одно истинное срабатывание приходится не более одного ложного и пользователь не тратит много времени на просмотр предупреждений. Естественно, более высокие показатели в 80-90% для истинных срабатываний предпочтительнее, но только в том случае, если при этом не происходит потери количества найденных дефектов.

Описанный выше подход используется в инструменте Svace. Целью анализа является поиск как можно большего количества дефектов при приемлемом количестве ложных срабатываний и времени анализа. Как упоминалось выше, инструмент Svace выдал истинные предупреждения для проектов, для которых уже проводилось тестирование и осуществлялся поиск ошибок с помощью других статических анализаторов. Тестирование не покрывает все возможные пути в программе, поэтому после проведения тестирования, запуск статических анализаторов может выявить необнаруженные ошибки. Среди причин, почему найденные ошибки не были исправлены после прогона других статических анализаторов, можно выделить 2:

- Другие анализаторы также не осуществляют поиск всех возможных дефектов и используют другие методы поиска дефектов и отсеивания ложных срабатываний.
- Анализаторы нашли все возможные дефекты, но соответствующие предупреждения либо не были просмотрены, либо были по ошибке помечены как ложные. Т.е. поиск всех дефектов не гарантирует, что они все будут исправлены.

Важно отличать дефекты в исходном коде и дефекты во время выполнения программы. Не все дефекты в исходном коде могут привести к проблемам во время выполнения программы, но они могут свидетельствовать о наличии ошибок в логике реализации алгоритмов, либо затруднять чтение кода и поддержку программы. Одним из таких примеров является наличие переменных, которые были инициализированы, но нигде не используются. Такие переменные не могут привести к падению программы, либо к другим дефектам, но являются признаком плохого кода, и возможно являются опечатками и свидетельствуют о наличии других проблем.

Даже если некоторая функция написана, но нигде не вызывается, желательным является поиск дефектов в её коде. Возможно, программист написал функцию, но не успел написать код её использующий. Если в функции есть дефект, то лучше, если на следующий день программист получит сообщение о дефекте в новой функции.

3. Классификация методов статического анализа

Многие подходы статического поиска ошибок исторически развились из области компиляции программ и оперируют абстракциями, взятыми оттуда: поток токенов, абстрактное синтаксического дерево, граф вызовов, граф потока управления, поток данных.

По типу используемых абстракций методы поиска ошибок можно разделить на следующие группы:

- Лексические анализаторы, рассматривающие программу как поток токенов. С их помощью можно найти только самые простейшие виды дефектов и в работе они рассматриваться не будут.

- Легковесные анализаторы (анализаторы 1го уровня), осуществляющие анализ преимущественно с помощью просмотра абстрактного синтаксического дерева (АСД), а также с использованием других абстракций уровня синтаксического анализа.
- Более сложные анализаторы (анализаторы 2го уровня), которые используют абстракции, связанные с фазами после синтаксического анализа.

В результате работы синтаксического анализатора компилятором строится абстрактное синтаксического дерево, которое позже передаётся на следующие фазы компиляции программы. Анализаторы, построенные на базе АСД, осуществляют проход по узлам АСД и делают относительно простые проверки анализируемых правил. Время работы таких анализаторов линейно зависит от размера программы.

В группе легковесных анализаторов можно выделить более сложные анализы, использующие некоторую модель памяти [6]. Такие анализаторы позволяют реализовать довольно сложные детекторы и найти ошибки, которые невозможно найти простым проходом по АСД. Но не имеет смысла делать их сложнее, чем анализаторы 2го уровня, т.к. более сложная реализация потребует абстракций, которые не доступны на фазе синтаксического анализа, но будут доступны позже. Такая реализация не будет иметь преимуществ перед детекторами, использующими следующие фазы компиляции.

На втором уровне может быть выполнено значительное количество анализов. Здесь доступна информация об алиасах в программе, о константах, возможных значениях переменных, графе вызовов и др. Методы анализа 2го уровня также можно разделить по тому какие свойства они учитывают при анализе: чувствительность к потоку, межпроцедурность, чувствительность к контексту и др.

Потоково-нечувствительные анализы рассматривают программу как неотсортированный набор инструкций. Часть дефектов в программах не зависят от порядка выполнения инструкций и их реализация с помощью потоково-нечувствительного анализа будет более эффективной. Потоково-чувствительный анализ учитывает порядок инструкций. Такие анализы по-разному отвечают на вопросы о свойствах программы в зависимости от точки программы. Такой анализ в среднем требует больше памяти на хранение информации для каждой точки программы. При эффективной реализации требования к памяти не растут линейно с ростом размера программы, т.к. большая часть информации о свойствах программы разделяется между различными точками программы.

Потоково-чувствительные анализы в свою очередь можно разделить на анализы с чувствительностью к путям и без чувствительности к путям. Анализы с чувствительностью к путям учитывают по какому пути прошло выполнение программы. Не все пути в программе могут быть выполнены, т.к. будут зависеть от противоречивых условий. Для решения задачи

выполнимости путей такие анализы часто используют SMT-решители, отсеивающие заведомо невыполнимые пути.

Многие дефекты являются результатом неправильного использования нескольких функций. Такие дефекты можно обнаружить только если проводить межпроцедурный анализ. Будем считать, что межпроцедурный анализ имеет контекстную чувствительность, если анализ отличает эффекты вызова функции в зависимости от контекста её вызова.

Учёт какого-либо свойства требует большего количества памяти и времени работы анализа, но зато даёт лучшую точность. Недостаточная точность анализа приводит либо к пропуску ошибок, либо к появлению ложных срабатываний. Будем считать более сложным анализ, который учитывает больше свойств. При этом анализы могут иметь разную степень чувствительности к какому-либо свойству. Например, при чувствительности к контексту вызова можно учитывать значения не всех переменных или анализировать разную высоту стека вызовов.

Критичность найденных ошибок, как правило, не зависит от сложности анализатора. Многие критичные ошибки являются результатом опечатки и часто могут быть найдены с помощью анализаторов 1го уровня. Более простые правила желательно реализовывать с помощью легковесных анализаторов на основе АСД. В этом случае и скорость написания правила и скорость анализа будет лучше, чем если бы оно реализовывалось в более тяжеловесном анализаторе. Кроме того, часть информации, которая доступна в АСД, может быть недоступной на более поздних фазах анализа. Например, наличие или отсутствие фигурных скобок можно проверить только на этапе синтаксического анализа, т.к. далее эта информация не сохраняется.

Для каждого конкретного правила желательно выбрать наиболее подходящую сложность используемых анализаторов. Если использовать более сложные анализаторы для поиска всех дефектов, то общее время работы инструмента увеличится. Разгрузив более сложные анализы от поиска дефектов, которые могут быть найдены более лёгкими анализами, можно увеличить скорость работы, а во многих случаях и скорость разработки детекторов.

Из-за проблемы неразрешимости задачи поиска дефектов в исходном коде программ, любой статический анализатор находит приближённое решение. Анализ будем считать консервативным, если приближённое решение гарантированно включает все возможные варианты при выполнении программы. Консервативность гарантирует, что все полученные выводы являются корректными. Это очень важно для оптимизирующих компиляторов, которые не должны изменять семантику программы. Если нет достаточно уверенности, что оптимизация безопасна, то лучше вообще её не применять, чем изменить поведение программы. При поиске же ошибок, иногда желательно выдать предупреждение об ошибке, которое может оказаться ложным, либо основано на некоторых эвристиках. Использование неконсервативного анализа позволяет создать более простой и быстрый

анализатор, и в тоже время находить довольно сложные типы ошибок. Расплатой за это является недостаточно точная модель программы, что в некоторых случаях может приводить к непонятным ложным срабатываниям.

На рис. 1 показан фрагмент кода, где осуществляется запись в переменную x через указатель p. В конце функции происходит считывание значения переменной в переменную i, которая используется для доступа к массиву размера 10. Таким образом может произойти переполнение массива, если значение i равно 20. Оптимизирующий компилятор в данном случае не может сделать такой вывод, т.к. возможно на эту область памяти указывает другой указатель, который обновляет память при вызове функции func. В данном случае ничего не известно про функцию func, она может менять значение переменной x, причём переменная может меняться при любых условиях, либо только при некоторых условиях. Будет ли в примере ошибка, зависит от реализации функции func. Человек, использующий статический анализатор, скорее всего, хотел бы, чтобы данная ошибка была выдана. Исключением можно назвать ситуацию, когда функция func тривиальна и перезаписывает значение x на безопасное.

```
char buf[10];
int* p = &x;
*p = 20;
func();
int i = *p;
buf[i] = 0;
```

Рис. 1. Потенциальное переполнение буфера.

Подчеркнём, что проблема не в том, что неизвестна реализация функции func, а в том, что в общем случае невозможно определить, что именно она будет делать. Даже если доступен исходный код функции, не всегда можно определить при каких условиях функция выполняет некоторые действия. К примеру, в функции есть условие, описываемое некоторой недоказанной математической теоремой. Если теорема верна, то условие будет тривиально истинным.

4. Краткое описание Svace

Инструмент Svace, разработанный в Институте системного программирования РАН, осуществляет статический поиск дефектов, используя несколько видов анализаторов. Текущая версия Svace[1] для анализа Си и Си++ программ

[1] В статье описывается только часть инструмента Svace для поиска ошибок в программах на языках Си и Си++.

содержит статические анализаторы двух уровней: легковесный анализатор и основной анализатор. Легковесный анализатор, интегрированный в компилятор Clang, осуществляющий поиск дефектов просматривая абстрактное синтаксическое дерево. Более сложный анализатор, осуществляет межпроцедурный потоково- и контекстно-чувствительный анализ.

Основной анализатор осуществляет межпроцедурный анализ путём обхода графа вызовов «снизу-вверх», начиная с листьев графа таким образом, чтобы вызываемые функции анализировались до вызывающих. Каждая функция обходится только один раз, вся необходимая в дальнейшем информация помещается в специальную структуру данных, называемую аннотацией. Такой подход позволяет анализировать каждую функцию только один раз, что существенно улучшает скорость анализа и масштабируемость. При таком подходе естественным образом получается чувствительность к контексту, т.к. каждая аннотация применяется независимо для каждого контекста вызова. Недостатком такого подхода является необходимость хранить все данные, которые могут потребоваться: если какая-то информация не будет сохранена, то в дальнейшем получить её будет уже невозможно.

При анализе каждой функции создаётся граф потока управления и производится его топологическая сортировка. Затем осуществляется анализ 2х видов: консервативный анализ потока данных и неконсервативный анализ, использующий сложную модель памяти. Оба вида анализов осуществляют как прямой, так и обратный анализы. При этом прямые анализы осуществляют прямой топологический обход графа потока управления и обычно отвечают на вопрос, что произошло или могло произойти с переменной. Обратные анализаторы выполняют обратный обход и отвечают на вопрос, что произойдёт или может произойти с переменной. Например, классический анализ живых переменных является обратным анализом и отвечает на вопрос может ли значение переменной быть использовано.

Неконсервативный анализ имеет поддержку чувствительности к путям, контексту и поддерживает межпроцедурный анализ. Ядро анализатора поддерживает возможность проводить анализ разной степени сложности, а использование этих возможностей зависит от конкретного детектора. Благодаря этому для каждого детектора можно выбрать необходимую степень учитываемых свойств. Основное неконсервативное предположение, которое используется в этом анализе - отсутствие алиасов среди входных параметров функции и глобальных переменных. Как правило, это условие выполняется, т.к. при наличии большого количества алиасов программисту самому сложно понимать семантику функции.

В конце анализа запускается парный анализ конструкторов и деструкторов, осуществляющий поиск несогласованностей при их реализации. Описание этой фазы приведено в главе 10.

Описание Svace также можно найти в [7-9].

Инструмент Svace имеет приемлемые характеристики истинных срабатываний и время анализа. По данным наших заказчиков стабильные детекторы Svace, реализованные в основном анализаторе для Си и Си++, имеют долю истинных срабатываний от 69 до 95%. Детекторы из легковесного анализатора имеют лучший процент истинных срабатываний.

Для оценки времени анализа необходимо в общее время также включать время сборки проекта. Общее время для сборки и анализа должно позволять проводить анализ во время ночной сборки. Время анализа для небольших проектов (busybox, cairo, dnprogs) составляет около 5 минут на обычном компьютере. Время сборки сравнимо. Анализ операционной системы android-5.0.2 на сервере с 256Гб оперативной памяти занимает около 5 часов. Сборка андроида занимает 2.5 часа. Таким образом общее время, необходимое на сборку и анализ андроида составляет 7.5 часов, что является приемлемым для ежедневного анализа во время ночной сборки проекта.

5. Анализатор на основе АСД

Многие виды дефектов могут быть обнаружены только на стадии синтаксического анализа, т.к. необходимая информация не передаётся на следующие стадии. Такие детекторы осуществляют просмотр АСД и проверяют его соответствие некоторым правилам. На этой фазе не доступна информация об алиасах программы, графе потока управления, графе вызовов, и не производится межпроцедурный анализ. Поэтому сложные дефекты не могут быть обнаружены на этой фазе. Но, т.к. критичность ошибок не зависит от сложности используемого анализа, а также то, что выданные предупреждения имеют высокий процент истинных срабатываний из-за относительно простой реализации, анализ на основе АСД позволяет находить множество дефектов в программах.

Детектор NO_EFFECT.SELF_ASSIGN проверяет аргументы для узла присваивания в АСД. Если аргументы соответствуют одному и тому же символу, то выдаётся предупреждение. Детектор способен найти ошибки вида: «x = x» или «p->q = p->q». Но так как сравнение происходит только для символов и отсутствует информация об алиасах, то ошибка не будет выдана для следующего кода:

```
x = y;
y = x;//здесь переменная x уже имеет значение y.
```

Пример предупреждения NO_EFFECT.SELF_ASSIGN, выданное для проекта nss-3.12.9+ckbi-1.82 приведён на рис. 2. Строка 921 содержит присваивание для одной и той же переменной. Скорее всего ошибка является результатом опечатки.

```
913| if (rawptr >= end) {
 pubk->u.fortezza.DSSKey.len = pubk->u.fortezza.KEAKey.len;
```

```
 pubk->u.fortezza.DSSKey.data=
          pubk->u.fortezza.KEAKey.data;
921|  pubk->u.fortezza.DSSprivilege.data =
          pubk->u.fortezza.DSSprivilege.data;
goto done; }
```

Рис. 2. Пример срабатывания NO_EFFECT.SELF_ASSIGN.

6. Консервативный анализ потока данных

Консервативный анализ используется главным образом для сбора вспомогательных данных о функции и переменных: анализ живых переменных, недостижимого кода, функций, завершающих выполнение программы. Результаты этого анализа используется неконсервативным анализом. Этот вид анализа имеет чувствительность к потоку и ограниченную межпроцедурность (распространяются только данные о функциях, завершающих программу). Чувствительность к путям и контексту не реализована, чтобы не слишком замедлять эту фазу.

В консервативном анализаторе реализован детектор поиска недостижимого кода. Анализ недостижимого кода используется далее в основном анализаторе svace, чтобы исключить влияние на анализ инструкций, которые недостижимы. Анализатор реализован в консервативной фазе, т.к. этот анализ может существенно влиять на работу других анализаторов.

Обычно наличие недостижимого кода не влияет на производительность программы, оптимизирующий компилятор всё равно удалит этот код из программы. Но наличие такого кода может свидетельствовать об опечатках, либо неправильном понимании программы. Программисты редко пишут код, который никогда не может быть выполнен.

Предупреждение UNREACHABLE_CODE выдаётся для участков кода, которые не могут быть достигнуты при выполнении программы. Написано 4 вида анализов потока данных для поиска недостижимого кода: на основе интервалов, ненулевых значений, простых предикатов и функций, завершающих программу. Анализ на основе интервалов для переменных программы сопоставляет интервал возможных целочисленных значений [a; b], что означает, что во время выполнения все возможные значения переменной принадлежат этому интервалу. После завершения анализа осуществляется проход по графу потока управления и для всех условных инструкций проверяются возможные интервалы значений. Если одна из веток условной инструкции при этом не может быть выполнена, то выдаётся предупреждение.

С помощью интервала нельзя описать интервал с выколотой точкой, и в частности ситуацию, что некоторая переменная имеет любое ненулевое значение. Так как сравнение с нулём частая операция, используемая в Си, то был реализован анализ ненулевых значений, который для каждой переменной проверяет, что переменная точно имеет ненулевое значение. После чего также

выполняется проход по графу потока управления. На рис. 3 показан пример предупреждения, найденный описанным анализом для проекта tizen/external-swig/allegrocl.cxx:2699. В примере содержится лишнее сравнение указателя tm с нулём, поэтому код на else-ветке будет недостижимым. После первого условия анализ пометит переменную tm, как имеющую ненулевое значение, далее при анализе второго условия, анализатор выдаст предупреждение, т.к. tm не может быть нулём и сравнивается с нулём. Эта ошибка не обязательно является безобидной лишней проверкой на ноль, которую легко выкинет компилятор. В данном случае, если tm равно нулю, функция не завершит программу, а продолжит выполнение. При этом в отладочном логе не будет содержаться запись о дефекте.

```
if (!is_void_return && tm) {
   if (tm) {
     Replaceall(tm, "$result", "lresult");
     Printf(f->code, "%s\n", tm);
     Printf(f->code, "        return lresult;\n");
     Delete(tm);
   } else {
   Swig_warning(WARN_TYPEMAP_OUT_UNDEF, input_file, line_number,
         "Unable to use return type %s in function %s.\n",
         SwigType_str(t, 0), name);
}}
```

Рис. 3. Недостижимый код

Анализ предикатов в качестве свойств потока данных использует конъюнкции простых предикатов вида “a op b”, где a и b - переменные, либо константы, а op - следующие операции: ==, !=, >, <, >=, <=. Если для некоторой точки оказывается, что предикат тривиально равен лжи, значит данная точка недостижима. Предыдущий пример также может быть найден этим анализом, т.к. в точке вызова функции Swig_warning предикат имеет вид “is void_return==0 & tm!=0 & tm==0”.

Последний вид анализов осуществляет анализ функций, которые завершают выполнение программы, и распространяет вдоль графа потока управления свойство “завершающая функция точно была вызвана в данной точке”. Если это свойство истинно, то код не может быть выполнен.

7. Потоково-чувствительный анализ

Основной анализатор Svace является неконсервативным потоково-чувствительным. При необходимости отдельные детекторы могут использовать общую инфраструктуру для межпроцедурного анализа и анализа с чувствительностью к путям.

Многие ошибки, встречающиеся в коде, являются локальными и часто вообще находятся на одной строке, поэтому имеет смысл реализовать простые детекторы для поиска таких ошибок. В Svace реализовано несколько относительно простых детекторов, осуществляющих поиск ошибок разыменования нулевых указателей. Каждый детектор осуществляет поиск некоторого конкретного вида ошибки. Преимуществом использования таких детекторов помимо скорости работы и небольшого использования памяти, является простота их реализации и, следовательно, минимизация ошибки при реализации детектора.

Каждая функция программы анализируется сама по себе без конкретного контекста, где она может быть вызвана. Серьёзной проблемой при этом является то, что некоторые функции накладывают определённые требования к контексту использования, и не могут быть вызваны в произвольном контексте. Примером такой функции является функция стандартной библиотеки Си memcpy, осуществляющая копирование из одной области памяти в другую. Функция имеет 3 параметра: 2 указателя на области памяти и количество копируемых байт. Если количество копируемых байт не нулевое, то 2 указателя не могут иметь нулевые значения. Если вызвать функцию memcpy с параметрами (0, 0, 10), то произойдёт ошибка разыменования нулевого указателя. Но это будет ошибка вызывающего кода, а не кода memcpy. Поэтому нельзя выдавать предупреждение каждый раз, когда происходит разыменование указателя без проверки на нулевое значение без информации о том, в каком контексте функция может быть вызвана.

Сформулируем следующий принцип написания функции: «каждая инструкция функции должна быть достижима хотя бы для одного потенциального контекста вызова». Потенциальный контекст вызова не обязательно присутствует в анализируемой программе. Если инструкция недостижима ни из одного потенциального контекста вызова, то возникает вопрос, зачем этот код вообще писали. Данный принцип позволяет находить большое количество дефектов при анализе функций самих о себе. Предупреждение выдаётся, если все пути, проходящие через некоторую инструкцию, содержат ошибку. Если инструкция достижима во время выполнения программы, то предупреждение соответствует дефекту во время выполнения программы. Если инструкция не достижима, то это является дефектом исходного кода программы. Таким образом, если из гипотезы о достижимости инструкции следует наличие в коде ошибки, то выдаётся предупреждение, если гипотеза не верна, то в теле функции находится инструкция, которая ни при каких условиях не будет исполнена, что само по себе является дефектом в исходном коде.

Предупреждение DEREF_OF_NULL описывает ситуации, когда указатель сравнивается с нулём и затем разыменовывается, в точке разыменования указатель может иметь только нулевое значение. Ошибка выглядит довольно глупой, но тем не менее, встречается в реальных проектах. Детектор ассоциирует с переменными свойство «переменная точно была положительно

сравнена с нулём» и распространяет это свойство по графу потока управления. На рис. 4 показаны примеры ошибок, которые могут быть найдены этим детектором:

```
if(!p) {
  *p = 0;//p может иметь только нулевое значение
}
if(q) {
  exit(0);
}
*q = 0;//q может иметь только нулевое значение
```

Рис. 4. Паттерны детектора DEREF_OF_NULL

Рис. 5 содержит пример подобной ошибки из того же проекта tizen/framework-uifw. Ошибка довольно тривиальна. Место разыменования находится всего через 2 строки от проверки на ноль. Не стоит недооценивать такие предупреждения. Если по какой-то причине функция NextIndicatorName вернёт нулевой указатель, то пользователь вместо сообщения об ошибке получит падение программы. Для поиска таких ошибок нет смысла реализовывать сложный детектор, имеющий чувствительность к путям или контексту.

```
260| old = new
261| new = NextIndicatorName(info);
262| if (!new)
263| {
264|    WSGO1("Couldn't allocate name for %d\n", new->ndx);
265|        ACTION("Ignored\n");
266|        return False;
267| }
```

Рис. 5. Разыменование нулевого указателя

Детектор DEREF_AFTER_NULL находит ситуации, в которых указатель сравнивается с нулём, а позже разыменовывается. В отличии от DEREF_OF_NULL в точке разыменования указатель не обязательно имеет нулевое значение. Детектор использует вышеописанный принцип, что каждая инструкция должна быть достижима хотя бы для некоторого контекста вызова. Детектор находит ошибки для следующего паттерна:

`if(p) { /*..*/}`

`*p = 0;`//здесь p, может иметь нулевое значение. Точка разыменования постдоминирует над сравнением.

Т.к. в точке разыменования указатель не обязательно имеет нулевое значение, то для поиска таких ошибок нужна чувствительность к путям, чтобы

проверить, что такой путь может существовать. Детектор DEREF_AFTER_NULL не использует чувствительность к путям, и чтобы не выдавать ложные срабатывания, связанные с тем, что путь не существует, детектор выдаёт предупреждение только, если точка разыменования постдоминирует над точкой сравнения, т.е. если после положительного сравнения с нулём указатель обязательно будет разыменован. Для реализации детектора используется прямой анализ, распространяющий свойство, что указатель был положительно сравнен с нулём, и обратный анализ для свойства, что указатель будет разыменован.

Большое количество найденных ошибок связано с неправильным использованием конъюнкций и дизъюнкций, их часто путают друг с другом. Наиболее типичной является ситуация, где с помощью конъюнкции проверяют более узкое условие P && E, вместо проверки E.

Пример ошибки, найденный детектором для tizen/external-eglibc показан на рис. 6. Если указатель namehashent имеет нулевое значение, а переменная replace имеет значение true, то не произойдёт выход из функции, и указатель будет разыменован. По всей видимости проверка значения replace здесь лишняя, возможно “&&” надо заменить на “||”.

```
   struct namehashent *namehashent = insert_name (ah, alias, strlen (alias), replace);
720:  if (namehashent == NULL && ! replace)
   return;
722:  if (namehashent->name_offset == 0) {
```

Рис. 6. Разыменование указателя после сравнения с нулём.

8. Чувствительность к путям

Анализ с чувствительностью к путям учитывает предикаты в инструкциях ветвления и способен отсеять несуществующие пути, зависящие от предикатов, которые не могут одновременно выполняться. Как правило такие анализы используют SMT-решатели для определения выполнимости предикатов.

Проблему зависимых предикатов можно проиллюстрировать с помощью паттерна «двойного ромба». Паттерн получил такое название, т.к. в графе потока управления выглядит как два последовательных ромба. На рис. 7 показан пример кода, имеющего 2 условия и 4 возможных пути выполнения. Если не рассматривать условия от которых зависит выполнения программы, то нельзя определить содержит ли данный код ошибку разыменования нулевого указателя Поэтому анализатору без чувствительности к путям остаётся либо выдавать предупреждение об ошибке, что приведёт к ложным срабатываниям во многих случаях, либо не выдавать, т.к. недостаточно данных. При использовании более сложных алгоритмов можно определить, что разыменование нулевого указателя произойдёт только если (!(a > b) &&

(a>b+1)), и с помощью SMT-решателя определить, что данная формула не имеет решений.

```
int g;
void func(int a, int b) {
        int*p = 0;
        if(a > b) { p = &g; }
        if(a > b + 1) {*p  = 6;}
}
```

Рис. 7. «Двойной ромб»

Даже при запоминании всех условий и использовании SMT-решателей не всегда возможно сказать может ли существовать некоторый путь или нет, что является следствием проблемы неразрешимости. Например, многие анализируемые условия зависят от решения задачи алиасов и без её решения консервативно можно только сказать, что условия могут быть зависимыми. Существующие лучшие алгоритмы анализа указателей, как правило, потоково-нечувствительные и часто имеют не лучшую скорость работы. Анализ Андерсона является одним из наиболее точных потоково-нечувствительных алгоритмов, но при этом алгоритм имеет кубическую сложность анализа [10] и поэтому плохо масштабируется для больших программ. Поэтому даже при использовании SMT-решателей может быть оправдан неконсервативный анализ и использование эвристик. К примеру можно считать, что значения глобальных переменных не являются алиасами входных параметров и не могут быть косвенно изменены другими функциями. Подобные допущения позволяют существенно ускорить анализ, и обычно не сильно ухудшают результаты анализа.

В Svace реализовано несколько детекторов, имеющих чувствительность к путям. DEREF_AFTER_NULL.EX - версия DEREF_AFTER_NULL с чувствительностью к путям. Детектор DEREF_AFTER_NULL из-за отсутствия анализа выполнимости условий на путях выдаёт предупреждение только, если на всех путях будет разыменование, благодаря чему удаётся избежать ложных срабатываний, связанных с отсутствием чувствительности к путям. Предупреждение DEREF_AFTER_NULL.EX выдаётся, если переменная сравнивается с нулём, а затем на некотором пути разыменовывается. Для поиска ошибки Svace осуществляет проход по графу потока управления и для каждой точки разыменования собирает необходимые условия достижимости точки разыменования. Затем начиная с каждой точки сравнения указателя с нулём Svace осуществляет проход по графу потока управления и собирает условия, при которых можно попасть из точки сравнения во все остальные точки. Затем для инструкции разыменования проверяется все условия с помощью SMT-решателя Z3[11]. Если условия могут быть выполнимы, значит может существовать путь, где указатель сначала сравнивается с нулём, а затем разыменовывается, в этом случае выдаётся предупреждение.

Пример предупреждения для проекта tizen-2.3 (gdhcp/client.c:2124). На строке 2096 переменная “message_type” сравнивается с нулём, но выход из функции произойдёт только если “client_id” тоже имеет нулевое значение. В противном случае если dhcp_client->state имеет одно из значений REQUESTING, RENEWING, REBINDING, то произойдёт разыменование на строке 2124. Детектор DEREF_AFTER_NULL не может обнаружить подобную ошибку, т.к. разыменование происходит не на всех путях, а для путей, на которых происходит разыменование детектор не может определить их выполнимость. Разыменование произойдёт, если “client_id != 0 && dhcp_client->state==REQUESTING”, но для детектора без чувствительности к путям это условие не отличимо от условия “client_id != 0 && message_type != 0” и от условия “client_id != 0 && client_id == 0”, при которых путь будет невыполним.

```
2096|   if (message_type == NULL && client_id == NULL)
2098|       return TRUE;
2100|   debug(dhcp_client, "received DHCP packet xid 0x%04x "
2101|           "(current state %d)", xid, dhcp_client->state);
2102|
2103|   switch (dhcp_client->state) {
2121|   case REQUESTING:
2122|   case RENEWING:
2123|   case REBINDING:
2124|
      if (*message_type == DHCPACK) {
2125|           dhcp_client->retry_times = 0;
```

Рис. 8. Разыменование нулевого указателя на одном из путей.

9. Межпроцедурный анализ

Часть ошибок является результатом неправильного использования функций, либо неправильной реализации функций. Поиск таких ошибок представляет сложность как для человека, так и для статического анализатора. Вручную найти такие ошибки сложно из-за того, что код, который вызывает ошибку разнесён в разные функции или даже в разные файлы. Статическому анализатору необходимо учитывать межпроцедурные эффекты функций. Инструмент Svace осуществляет межпроцедурный анализ на основе аннотаций. При таком подходе после анализа функции создаётся её аннотация, описывающая интересующие эффекты вызова функции, а затем аннотация транслируется в вызывающий контекст, эмулируя вызов функции. Создание аннотаций требует процессорное время, а их хранение занимает место в оперативной памяти. Аннотации можно сохранять на диск, чтобы снизить требования к оперативной памяти, но сохранение и чтение с диска

замедляет анализ. Поэтому межпроцедурные детекторы потребляют больше памяти и процессорного времени по сравнению с внутрипроцедурными.

Аннотации, используемые в Svace, основаны на атрибутах, описывающих интересующие свойства. При этом атрибут может использоваться разными детекторами, если их интересует одно и тоже свойство. Выше описывались детекторы DEREF_OF_NULL, DEREF_AFTER_NULL, NULL_AFTER_DEREF, осуществляющие поиск разыменования нулевого указателя. Всем этим детекторам интересно свойство, что указатель разыменовывается в некоторой функции, для этого свойства создаётся атрибут, который записывается в аннотацию один раз и используется всеми этими детекторами, что экономит память по сравнению с версией, если бы каждый детектор самостоятельно писал в аннотацию.

При создании аннотации необходимо решить, что именно сохранять в аннотации. Чем больше информации будет сохранено, тем более медленным будет анализ. Чем меньше информации будет сохранено, тем меньше дефектов будет найдено. В Svace используется 2 подхода: запомнить свойство, которое верно для всех вызовов функций; запомнить при каких именно условиях происходит некоторое событие.

Если некоторое событие происходит при любом вызове некоторой функции, то для его сохранения не требуется много памяти. В минимальном случае достаточно использовать один бит - произошло событие или нет. На практике при сообщении об ошибке пользователю необходимо показать почему именно здесь ошибка, поэтому требуется также сохранить трассу, содержащую точки в программе для описываемого события. Для детекторов DEREF_OF_NULL, DEREF_AFTER_NULL, NULL_AFTER_DEREF это событие - разыменование. Если некоторый указатель разыменовывается на всех возможных путях, проходящих через некоторую функцию, то в аннотацию добавляется атрибут содержащий трассу разыменования - последовательность точек вызовов функций, завершающуюся точкой разыменования. Если есть путь, на котором нет разыменования указателя, то в аннотацию ничего не добавляется и предупреждение не будет выдаваться. Межпроцедурные версии этих детекторов находят те же самые дефекты, что и внутрипроцедурные, но способны выдать предупреждение, о разыменовании, происходящем в вызываемой функции.

Пример межпроцедурного дефекта проекта xorg-server-1.7.6 показан на рис. 9. Сравнение с нулём находится на 232 строке, разыменование же происходит внутри функции __glXDrawableRelease, которая вызвана на строке 242. Рис. 9 содержит код этой функции, где на строке 1147 происходит разыменование входного аргумента. В данном примере уже сложно разобраться что происходит, по этой причине помимо сообщения об ошибке, Svace дополнительно выдаёт трассы, показывающие, почему произошло разыменование. Предупреждение содержит 2 трассы: одну для точки сравнения с нулём, и одну для разыменования, включающую в себя 2 точки

(glxdri.c:242, где происходит вызов функции, и glxcmds.c:1147, где происходит разыменование).

```
232|    if (drawable->pDraw != NULL) {
233|      screen = (__GLXDRIscreen *) glxGetScreen(drawable->pDraw-
>pScreen);
234|   (*screen->core->destroyDrawable)(private->driDrawable);
236|      __glXenterServer(GL_FALSE);
237|      DRIDestroyDrawable(drawable->pDraw->pScreen,
238|                              serverClient, drawable->pDraw);
239|      __glXleaveServer(GL_FALSE);
240|    }
242| __glXDrawableRelease(drawable);
```

Код функции _glXDrawableRelease:

```
1145|void __glXDrawableRelease(__GLXdrawable *drawable)
1146|{
1147|    ScreenPtr pScreen = drawable->pDraw->pScreen;
1149|    switch (drawable->type) {
1150|    case GLX_DRAWABLE_PIXMAP:
1151|    case GLX_DRAWABLE_PBUFFER:
```

Рис. 9. Межпроцедурный дефект разыменования нулевого указателя

Наиболее сложный анализ осуществляется для поиска межпроцедурный ошибок в ситуациях, когда некоторое событие происходит в функции условно. Рассмотрим поиск разыменований нулевых указателей, где разыменование может произойти в функции при некоторых условиях. Необходимо запомнить условия, при которых вызываемая функция разыменовывает входные параметры, а также взаимосвязь всех переменных. Во многих случаях условия становятся слишком громоздкими и требующими много памяти для сохранения в аннотации. Если формула для условия становится слишком сложной, то Svace не сохраняет её в аннотации, и считается, что функция не разыменовывает свой аргумент. Для определения сложности формулы используются несколько параметров, включая высоту дерева разбора формулы, количество используемых атомарных формул, общее количество узлов в дереве разбора. Для увеличения количества формул, которые могут быть сохранены, используется несколько методов их упрощения. Во-первых, используются некоторые факты из логики высказываний, которые позволяют упростить формулы вида «a & (b & a)» в «a & b». Т.к. запись в аннотацию некоторой формулы означает, что если формула имеет решение, то произойдёт разыменование переменной, то корректно разрывать дизъюнкции и вместо формулы «a | b» сохранять «a». Формула «a» может быть проще формулы «a | b» и пройдёт требования на сложность формулы. При этом

произойдёт потеря точности, т.к. не будет сохранена информация, что если «b» истинно, то произойдёт разыменование, но это всё равно лучше, чем отсутствие любой информации из-за слишком сложной формулы.

При анализе вызова функции все формулы в аннотации необходимо оттранслировать в контекст вызова и переименовать все используемые переменные в соответствующие аналоги на стороне вызова. После этого для каждой переменной анализатор имеет условие в виде формулы, что переменная имеет нулевое значение перед вызовом функции, условие, что переменная будет разыменована в вызываемой функции, и также условие что инструкция вызова будет достигнута. Решателю Z3 передаётся конъюнкция всех 3х условий, и если он находит модель результирующей формулы, то вызывающая функция может разыменовать нулевой указатель.

10. Анализ конструкторов и деструкторов

В языке Си++ при создании объекта вызывается конструктор, осуществляющий его инициализацию. При удалении объекта будет вызван деструктор для очистки используемых ресурсов. Деструктор должен гарантировать, что все созданные в конструкторе ресурсы будут освобождены при уничтожении объекта. Несогласованность между конструкторами и деструкторами может приводить к утечке ресурсов, их двойному освобождению, либо освобождению не выделенных ресурсов.

Для поиска таких дефектов в конце анализа запускается фаза парного анализа конструкторов и деструкторов. Эта фаза была написана таким образом, чтобы по максимуму переиспользовать возможности основного анализатора Svace. Для этого в код деструктора в качестве первой инструкции вставляется вызов конструктора. Затем производится обычный анализ модифицированного кода, при обработке вызова конструктора используется его аннотация, также как и при межпроцедурном анализе. Различные детекторы при этом могут проверить несогласованность в конце анализа деструктора. Если у класса есть несколько конструкторов, то такой анализ проводится для каждого конструктора в отдельности. Также парный анализ осуществляется для операторов присваивания.

Детектор MEMORY_LEAK.CTOR выдаёт предупреждение, если в конструкторе выделяется память для некоторого члена класса, а в деструкторе отсутствует освобождение выделенной памяти. Для поиска ошибок в конце анализа модифицированного кода деструктора детектор запускает стандартный анализ утечек памяти, осуществляющий поиск достижимых ячеек памяти. Если в результате поиска находится утечка памяти, то это означает, что после вызова деструктора не все ресурсы будут освобождены. Пример ошибки для проекта android 5.0.2 (AST.cpp:612) приведён на рис. 10.

```
611| FinallyStatement::FinallyStatement()
612|         :statements(new StatementBlock) {  }
  Код деструктора :
616| FinallyStatement::~FinallyStatement()
617|{  }
   Определение FinallyStatement (FrameworkListener.h) :
280| struct FinallyStatement : public Statement
281| {
282|     StatementBlock* statements;
```

Рис. 10. Несогласованность конструктора и деструктора.

11. Заключение

Как было показано, инструменты статического анализа позволяют находить ошибки в хорошо протестированных приложениях. Многие найденные ошибки связаны с обработкой ошибочных ситуаций и с редко исполняемыми путями. Для реализации подобных ошибок программа должна попасть в состояние, которое сложно смоделировать с помощью тестирования.

Каждое ложное срабатывание, выданное инструментом, выражается в деньгах, затрачиваемых на зарплату программистам, просматривающих сообщения об ошибках. Поэтому для практического поиска дефектов низкий уровень ложных срабатываний может быть даже важнее, чем общее количество найденных дефектов. Разные инструмента статического анализа используют различные алгоритмы поиска дефектов и способы борьбы с ложными срабатываниями. По этой причине даже после проверки программы статическим анализатором имеет смысл использовать другой анализатор для поиска дефектов.

В отличии от оптимизирующих компиляторов статическим анализаторам не обязательно выдавать консервативный результат. Во многих ситуациях лучше сообщить о проблеме даже, если нет достаточно уверенности в её наличии, что мотивирует использование неконсервативных анализаторов, использующих предположения, которые не обязательно верны для всех программ и функций.

Дефекты в программе имеют разную природу и для их поиска необходимо правильно выбрать алгоритм анализа. Хороший инструмент будет включать в себя как простые детекторы, использующие анализ на основе АСД, так и сложные детекторы, позволяющие найти нетривиальные межпроцедурные ошибки. Инструмент Svace состоит из набора анализаторов реализующих анализы разных типов: анализ на основе абстрактного синтаксического дерева, консервативный анализ потока данных для одной функции, потоко-чувствительный и межпроцедурный неконсервативный анализ с возможностью использовать чувствительность к путям. Построение инструмента на основе нескольких анализаторов позволяет использовать

преимущества этих видов анализаторов и находить больший диапазон ошибочных ситуаций.

Список литературы:

[1]. S. C. Misra, V. C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality //Computational Science and Its Applications—ICCSA 2003. – Springer Berlin Heidelberg, 2003. – С. 724-732.

[2]. https://msdn.microsoft.com/library/cc307416

[3]. M. Tim Jones. Static and dynamic testing in the software development life cycle. 26 August 2013 (http://www.ibm.com/developerworks/library/se-static/)

[4]. А. С. Марков, В. Л. Цирлов, А. В. Барабанов. Методы оценки несоответствия средств защиты информации //М.: Радио и связь. – 2012.

[5]. T. Kremenek, D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations //Static Analysis. – Springer Berlin Heidelberg, 2003. – С. 295-315.

[6]. В.Н. Игнатьев. Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования. Труды ИСП РАН, том 22, 2012, с. 169–188. DOI: 10.15514/ISPRAS-2012-22-11.

[7]. В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ // Труды Института системного программирования РАН. 2014. Т. 26. С. 231–250. DOI: DOI: 10.15514/ISPRAS-2014-26(1)-7.

[8]. А.И. Аветисян, А.Е. Бородин. Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок. Труды ИСП РАН, том 21, 2011, с. 39–54.

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

[10]. M. Shapiro, S. Horwitz. Fast and accurate flow-insensitive points-to analysis //Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – ACM, 1997. – С. 1-14.

[11]. L. De Moura, N. Bjørner. Z3: An efficient SMT solver //Tools and Algorithms for the Construction and Analysis of Systems. – Springer Berlin Heidelberg, 2008. – С. 337-340.

A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels

[1]A. Borodin <alexey.borodin@ispras.ru>
[1,2]A. Belevancev <abel@ispras.ru>
[1]Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. The paper describes a practical approach for finding bugs in the source code of programs using static analysis. The approach is implemented in the Svace tool that is developed by ISP RAS. Svace performs defect detection for different error types including null pointer dereferences, buffer overruns and underruns, uninitialized variables usages, memory leaks, double locks and missing locks, unreachable code, division by zero, use after free and others.

The analysis goal is to find as many defects as possible while minimizing false positives with acceptable analysis time. As a result, on large programs the approach inevitably results in missing some defects.

Even critical program defects exist because of various reasons, and the right analysis approach should be detected based on a defect type. A good analyzer will include both simple detectors using only semantic analysis on an abstract syntax tree (AST) and complex detectors using interprocedural context and path sensitive analyzers. The Svace analyzer is designed for that purpose as a collection of analyzers having various levels: an AST analyzer, a conservative data flow analyzer, flow, context and path sensitive interprocedural analysis that makes a few assumptions losing conservativeness. The interprocedural analysis is annotation based: each function is analyzed only once, and its annotation created to summarize the analysis results is used when simulating this function's call. All described algorithms are presented and illustrated using examples of various detectors and their real warnings found on a number of open source projects.

Keywords: static analysis; C language; defects in source code; abstract syntax tree; flow-sensitivity; path-sensitivity; interprocedural analysis; unsound analysis; null pointer dereference

DOI: 10.15514/ISPRAS-2015-27(6)-8

For citation: Borodin A., Belevancev A.. A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8.

References:

[1]. S. C. Misra, V. C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality //Computational Science and Its Applications—ICCSA 2003. – Springer Berlin Heidelberg, 2003. – pp. 724-732.

[2]. https://msdn.microsoft.com/library/cc307416

[3]. M. Tim Jones. Static and dynamic testing in the software development life cycle. 26 August 2013 (http://www.ibm.com/developerworks/library/se-static/)

[4]. A. S. Markov, V. L. Cirlov, A. V. Barabanov. Metody ocenki nesootvetstvija sredstv zawity informacii [Methods for assessing non-compliance means of information protection] //M.: Radio i svjaz' [Radio and Communication] – 2012. (in Russian)

[5]. T. Kremenek, D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations //Static Analysis. – Springer Berlin Heidelberg, 2003. – pp. 295-315.

[6]. V.N. Ignat'ev. Ispol'zovanie legkovesnogo staticheskogo analiza dlja proverki nastraivaemykh semanticheskikh ogranichenij jazyka programmirovanija [Static analysis usage for customizable checks of programming languages semantic constraints]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 22, 2012, pp. 169–188. DOI: 10.15514/ISPRAS-2012-22-11. (in Russian)

[7]. V.P. Ivannikov, A.A. Belevancev, A.E. Borodin, V.N. Ignat'ev, D.M. Zhurikhin, A.I. Avetisjan, M.I. Leonov. Staticheskij analizator Svace dlja poiska defektov v iskhodnom kode programm [Svace: static analyzer for detecting of defects in program source code] // Trudy ISP RAN [The Proceedings of ISP RAS], volume 26, issue 1, pp. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7. (in Russian)

[8]. A.I. Avetisjan, A.E. Borodin. Mekhanizmy rasshirenija sistemy staticheskogo analiza Svace detektorami novykh vidov ujazvimostej 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], volume 21, 2011, pp. 39–54. (in Russian)

[9]. A.I. Avetisjan, A.A. Belevancev, A.E. Borodin, V.S. Nesov. Ispol'zovanie staticheskogo analiza dlja poiska ujazvimostej i kriticheskikh oshibok v iskhodnom kode programm [Using static analysis for searching vulnerabilities and critical errors in the source code of programs]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 21, 2011, pp. 23–38. (in Russian)

[10]. M. Shapiro, S. Horwitz. Fast and accurate flow-insensitive points-to analysis //Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – ACM, 1997. – pp. 1-14.

[11]. L. De Moura, N. Bjørner. Z3: An efficient SMT solver //Tools and Algorithms for the Construction and Analysis of Systems. – Springer Berlin Heidelberg, 2008. – pp. 337-340.

Инструментация и оптимизация выполнения транзакционных секций многопоточных программ

[1]И.И. Кулагин <ivan.i.kulagin@gmail.com>
[2]М.Г. Курносов <mkurnosov@gmail.com>
[1] Сибирский государственный университет телекоммуникаций и информатики, 630102, Россия, г. Новосибирск, ул. Кирова, дом 86.
[2]Санкт-Петербургский государственный электротехнический университет «ЛЭТИ», 197376, Россия, г. Санкт-Петербург, ул. Профессора Попова, дом 5.

Аннотация. В работе выполнено исследование эффективности реализации программной транзакционной памяти (software transactional memory) в компиляторе GCC, предложен метод инструментации параллельных программ, использующих транзакционную память, для осуществления задач профилирования, а также предложен подход к сокращению числа ложных конфликтов, возникающих при выполнении транзакционных секций. Суть подхода заключается в варьировании параметров реализации транзакционной памяти в runtime-библиотеке компилятора GCC по результатам предварительного профилирования программы (profile-guided optimization). Предложенный метод инструментации позволяет оптимизировать динамические характеристики выполнения транзакционных секций. Эффективность подхода к сокращению числа ложных конфликтов исследована на тестовых многопоточных программах из пакета STAMP.

Ключевые слова: программная транзакционная память, инструментация, оптимизация по результатам предварительного профилирования, многопоточное программирование, компиляторы.

DOI: 10.15514/ISPRAS-2015-27(6)-9

Для цитирования: Кулагин И.И., Курносов М.Г. Инструментация и оптимизация выполнения транзакционных секций многопоточных программ. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 135-150. DOI: 10.15514/ISPRAS-2015-27(6)-9.

1. Введение

Синхронизация доступа к разделяемым ресурсам является одной из важных и сложных задач при разработке параллельных алгоритмов для многопоточных программ. Используя примитивы синхронизации (семафоры, мьютексы и др.) программист должен обеспечить не только корректность программы – отсутствие взаимных блокировок (deadlock, livelock) и состояний гонок за данными (data race), но и минимизировать время ожидания доступа к критическим секциям (разделяемым ресурсам).

Классические методы синхронизации, основанные на механизме блокировок, позволяют создавать в коде программы критические секции, выполнение которых возможно только одним потоком в каждый момент времени [1]. Такие примитивы синхронизации позволяют *защитить участок кода программы*, одновременно выполняющийся множеством потоков.

Анализ многопоточных программ показывает, что при одновременном выполнении потоками одной критической секций в ней может происходить обращения по разным адресам памяти и, следовательно, возникновение состояния гонки данных маловероятно. Например, такая ситуация наблюдается если в критическую секцию помещен код добавления элемента в хеш-таблицу (рис. 1). Если все потоки в один момент времени обратятся к функции для добавления нового элемента с хеш-кодом *i*, то с большой долей вероятности ключи *key* будут иметь разные хеш-коды и, как следствие, каждый поток будет выполнять добавления нового узла в отдельный связный список *h*[*i*]. Здесь блокировка мьютексом всей функции является избыточной и неэффективной.

```
function hashtable_add(h, key, value)
    lock_acquire()
    i = hash(key)
    list_add_front(h[i], key, value)
    lock_release()
end function
```

Рис. 1. Добавление пары (key, value) в хеш-таблицу h.

Активно развиваются два альтернативных подхода к созданию потокобезопасных и масштабируемых многопоточных программ – это *неблокирующие алгоритмы и структуры данных* (lock-free data structures) [2] и транзакционная память (ТП, transactional memory) [3, 4].

Использование неблокирующих структур данных, как правило, требует глубокой переработки многопоточных программ и совместно используемых потоками объектов в памяти [2].

Менее трудоемким и прозрачным для программиста видится использование технологии транзакционной памяти, основная идея которой заключается в *защите от конкурентного доступа области памяти* программы, а не участка кода, как в случае использования блокировок на базе мьютексов.

Известны как программные реализации транзакционной памяти (software transactional memory – STM): LazySTM, TinySTM, GCC TM, DTMC, RSTM, STMX, STM Monad, так и аппаратные реализации в процессорах (hardware transactional memory): Intel TSX, AMD ASF, Oracle Rock, IBM POWER8, IBM PowerPC A2.

В рамках программной транзакционной памяти программисту предоставляются языковые конструкции или API для формирования в программе *транзакционных секций* (transactional section) – участков кода, в которых осуществляется защита совместно используемых областей памяти. Выполнение потоками таких секций осуществляется без их блокирования. На среду выполнения (runtime) ложатся задачи по контролю за корректностью выполнения транзакций. Если во время выполнения транзакции другие потоки одновременно с ней не модифицировали защищенную область памяти, то транзакция считается корректной, и она фиксируется. Если же два или более потока, при выполнении транзакций, обращаются к одной и той же области памяти и как минимум один из них выполняет операцию записи, то возникает *конфликт* (аналог состояния гонки данных). Для его разрешения выполнение одной или нескольких транзакций может быть либо приостановлено (до завершения конфликтующей транзакции), либо прервано, а все модифицированные ими (их потоками) области памяти приведены в исходное состояние (на момент старта транзакции) – *отмена транзакции и восстановление* (cancel and rollback).

Для того, чтобы обнаруживать конфликты runtime-система должна отслеживать попытки одновременного доступа к одной и той-же области памяти. Это реализуется путем поддержки информации о состоянии защищаемых регионов памяти. Возможны два уровня гранулярности контролируемых областей: *уровень программных объектов* (object-based STM) и *уровень слов памяти* (word-based STM).

Уровень программных объектов подразумевает поддержку runtime-системой метаданных о состоянии каждого объекта программы. Например, объектов в C++-программе.

Для реализации уровня слов памяти в простейшем случае требуется каждый байт линейного адресного пространства процесса сопровождать метаданными, что является практически невозможным. Вместе этого линейное адресное пространство процесса разбивается на фиксированные блоки, каждый из которых сопровождается метаданными о состоянии (подход, подобный прямому отображению физических адресов на кеш-память процессора) [5, 6]. Это приводит к тому, что множеству областей памяти соответствуют одни метаданные, что является источником возникновения ложных конфликтов. *Ложный конфликт* (false conflict) – это ситуация, при которой два или более потока во время выполнения транзакции обращаются к разным участкам линейного адресного пространства, но отображаемым на одни и те же

метаданные. Поэтому runtime-система воспринимает такую ситуацию как конфликт (data race), хотя на самом деле таковой отсутствует.

Ложные конфликты существенно снижают эффективность параллельных STM-программ. Поэтому остро стоит задача разработки алгоритмов обнаружения и сокращения числа ложных конфликтов в реализациях STM.

В данной работе предлагается метод оптимизации ложных конфликтов по результатам предварительного профилирования C/C++ STM-программы. Для чего разработан модуль расширения компилятора GCC, который реализует инструментацию оптимизируемой STM-программы. Результаты выполнения инструментированной STM-программы поступают на вход созданного модуля анализа и варьирования параметров реализации runtime-библиотеки STM в компиляторе GCC (libitm).

2. Программная транзакционная память

Международным комитетом ISO по стандартизации языка C++, в рамках рабочей группы WG21, ведутся работы по внедрению транзакционной памяти в стандарт языка. Окончательное внедрение планируется в стандарт C++17. На сегодняшний день предложен черновой вариант спецификации поддержки транзакционной памяти в C++ [7]. Она реализована в компиляторе GCC начиная с версии 4.8 и предоставляет ключевые слова `__transaction_atomic`, `__transaction_relaxed` для создания транзакционных секций, а также `__transaction_cancel` для принудительной отмены транзакции.

Для выполнения транзакционных секций runtime-системой создаются транзакции. *Транзакция* (transaction) – это конечная последовательность операций транзакционного чтения/записи памяти. Операция транзакционного чтения выполняет копирование содержимого указанного участка общей памяти в соответствующий участок локальной памяти потока. Транзакционная запись копирует содержимое указанного участка локальной памяти в соответствующий участок общей памяти доступной всем потокам.

Инструкции транзакций выполняются потоками параллельно (конкурентно). После завершения выполнения транзакция может быть либо *зафиксирована* (commit), либо *отменена* (cancel). Фиксация транзакции подразумевает, что все сделанные в рамках нее изменения памяти становятся необратимыми. При отмене транзакции ее выполнение прерывается, а состояние всех модифицированных областей памяти восстанавливается в исходное с последующим перезапуском транзакции (*откат транзакции*, rollback).

Отмена транзакции происходит в случае *обнаружения конфликта* – ситуации, при которой два или более потока обращаются к одному и тому же участку памяти и как минимум один из них выполняет операцию записи.

Для разрешения конфликта разработаны различные походы, например, можно приостановить на некоторое время или отменить одну из конфликтующих транзакций.

На рис. 2 представлен пример создания транзакционной секции, в теле которой выполняется добавление элемента в хэш-таблицу множеством потоков. После выполнения тела транзакционной секции каждый поток приступит к выполнению кода, следующего за ней, в случае отсутствия конфликтов. В противном случае поток повторно будет выполнять транзакцию до тех пор, пока его транзакция не будет успешно зафиксирована.

Основными аспектами реализации транзакционной памяти в runtime-системах являются:

- политика обновления объектов в памяти;
- стратегия обнаружения конфликтов;
- метод разрешения конфликтов.

```
/* Совместно используемая хеш-таблица */
hashtable_t *h;

/* Код потоков */
void *thread_start(void *arg) {
    struct data *d = (struct data *)arg;
    prepareData(d);

    /* Транзакционная секция */
    __transaction_atomic {
        /* Добавление элемента в хеш-таблицу */
        struct data *d = (struct data *)arg;
        hashtable_insert(h, d);
    }

    saveData(d);
    return NULL;
}
```

Рис. 2. Использование транзакционной памяти в языке C/C++ (GCC libitm).

Политика обновления объектов в памяти определяет, когда изменения объектов в рамках транзакции будут записан в память. Распространение получили две основные политики – ленивая и ранняя. *Ленивая* политика обновления объектов в памяти (lazy version management) откладывает все операции с объектами до момента фиксации транзакции. Все операции записываются в специальном журнале (redo log), который при фиксации используется для отложенного выполнения операций. Очевидно, что это замедляет операцию фиксации, но существенно упрощает процедуры ее отмены и восстановления. Примером реализаций ТП, использующих данную политику являются RSTM-LLT [8] и RSTM-RingSW[9].

Ранняя политика обновления (eager version management) предполагает, что все изменения объектов сразу записываются в память. В журнале отката (undo log) фиксируются все выполненные операции с памятью. Он используется для восстановления оригинального состояния модифицируемых участков памяти в случае возникновения конфликта. Эта политика характеризуется быстрым выполнением операции фиксации транзакции, но медленным выполнением процедуры ее отмены. Примерами реализаций, использующими раннюю политику обновления данных являются GCC (libitm), TinySTM [5], LSA-STM [6], Log-TM [10], RSTM [8] и др.

Момент времени, когда инициируется алгоритм обнаружения конфликта, определяется *стратегией обнаружения конфликтов*. При *отложенной стратегии* (lazy conflict detection), алгоритм обнаружения конфликтов запускается на этапе фиксации транзакции [11]. Недостатком этой стратегии является то, что временной интервал между возникновением конфликта и его обнаружением может быть достаточно большим. Эта стратегия используется в RSTM-LLT [8] и RSTM-RingSW [9].

Пессимистичная стратегия обнаружения конфликтов (eager conflict detection) запускает алгоритм их обнаружения при каждой операции обращения к памяти. Такой подход позволяет избежать недостатков отложенной стратегии, но может привести к значительным накладным расходам, а также, в некоторых случаях, может привести к увеличению числа откатов транзакций. Стратегия реализована в TinySTM [5], LSA-STM [6] и TL2 [12].

В компиляторе GCC (libitm) реализован комбинированный подход к обнаружению конфликтов – отложенная стратегия используется совместно с пессимистической.

3. Обнаружение конфликтов

Выбор гранулярности обнаружения конфликтов – один из ключевых моментов при реализации программной транзакционной памяти. На сегодняшний день используются два уровня гранулярности: уровень программных объектов (object-based STM) и уровень слов памяти (word-based STM). Уровень программных объектов подразумевает отображение объектов модели памяти языка (объекты C++, Java, Scala) на метаданные runtime-библиотеки. При использовании уровня слов памяти осуществляется отображение блоков линейного адресного пространства процесса на метаданные. Метаданные хранятся в таблице, каждая строка которой соответствует объекту программы или области линейного адресного пространства процесса. В строке содержатся номер транзакции, выполняющей операцию чтения/записи памяти; номер версии отображаемых данных; их состояние и др. Модификация метаданных выполняется runtime-системой с помощью атомарных операций процессора.

В данной работе рассматривается реализация программной транзакционной памяти в компиляторе GCC, использующий уровень слов памяти (в версиях GCC 4.8+ размер блока – 16 байт).

На рис. 3 представлен пример организации метаданных транзакционной памяти с использованием уровня слов памяти (GCC 4.8+). Линейное адресное пространство процесса фиксированными блоками циклически отображается на строки таблицы, подобно кешу прямого отображения. Выполнение операции записи приведет к изменению поля «состояние» соответствующей строки таблицы на «заблокировано». Доступ к области линейного адресного пространства, у которой соответствующая строка таблицы помечена как «заблокировано», приводит к конфликту.

Основными параметрами транзакционной памяти с использованием уровня слов памяти являются число S строк таблицы и количество B адресов линейного адресного пространства, отображаемых на одну строку таблицы. От выбора этих параметров зависит число ложных конфликтов – ситуаций аналогичных ситуации ложного разделения данных при работе кеша процессора. В текущей реализации GCC (4.8-5.1) эти параметры фиксированы [13].

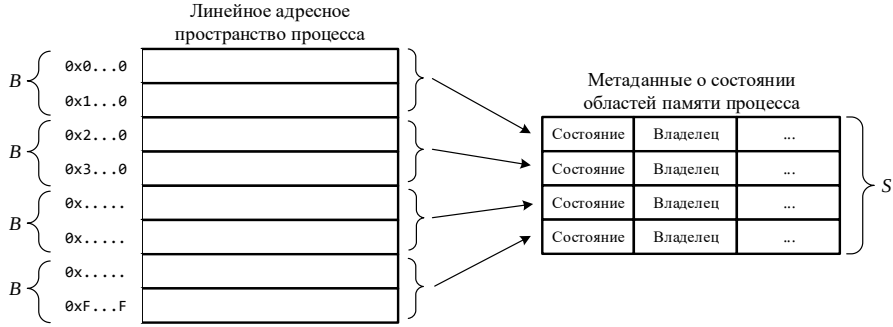


Рис. 3. Таблица с метаданными транзакционной памяти GCC 4.8+ (word-based STM): $B = 16$, $S = 2^{19}$.

4. Ложные конфликты

При отображении блоков линейного адресного пространства процесса на метаданные runtime-библиотеки возникают коллизии. Это неизбежно, так как размер таблицы метаданных гораздо меньше размера линейного адресного пространства процесса. Коллизии приводят к возникновению ложных конфликтов. *Ложный конфликт* – ситуация, при которой два или более потока во время выполнения транзакции обращаются к разным участкам линейного адресного пространства, но сопровождаемые одними и теме же метаданными о состоянии, и как минимум один поток выполняет операцию записи. Таким образом, ложный конфликт – это конфликт, который

происходит не на уровне данных программы, а на уровне метаданных runtime-библиотеки.

Возникновение ложных конфликтов приводит к откату транзакций, так же, как и возникновение обычных конфликтов, несмотря на то, что состояние гонки за данными не возникает, что влечет за собой увеличение времени выполнения STM-программ. Сократив число ложных конфликтов можно существенно уменьшить время выполнения программы.

На рис. 4 показан пример возникновения ложного конфликта в результате коллизии отображения линейного адресного пространства на строку таблицы. Поток 1 при выполнение операции записи над областью памяти с адресом A1 захватывает соответствующую строку таблицы. Выполнение операции чтения над областью памяти с адресом A2 потоком 2 приводит к возникновению конфликта, несмотря на то что операции чтения и записи выполняются над различными адресами. Последнее обусловлено тем, что 1 и 2 отображены на одну строку таблицы метаданных.

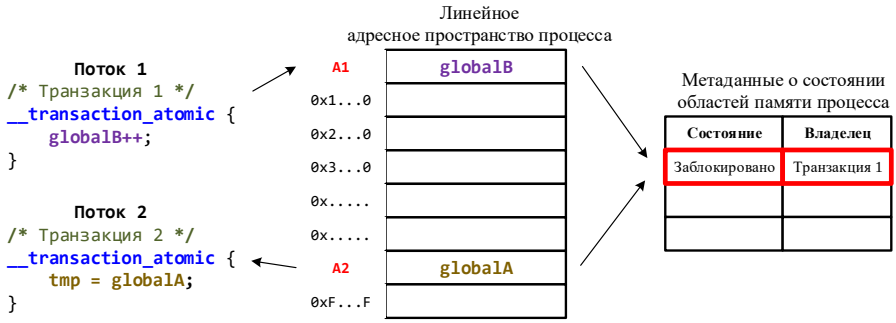


Рис. 4. Пример возникновения ложного конфликта при выполнении двух транзакций (GCC 4.8+).

5. Сокращение числа ложных конфликтов

В работе [14] для минимизации числа ложных конфликтов предлагается использовать вместо таблицы с прямой адресацией (как в GCC 4.8+), в которой индексом является часть линейного адреса, хеш-таблицу, коллизии в которой разрешаются методом цепочек. В случае отображения нескольких адресов на одну запись таблицы каждый адрес добавляется в список и помечается тэгом для идентификации (рис. 5). Такой подход позволяет избежать ложных конфликтов, однако накладные расходы на синхронизацию доступа к метаданным существенно возрастают, так как значительно увеличивается количество атомарных операций «сравнение с обменом» (compare and swap –
CAS).

Рис. 5. Хеш-таблица для хранения метаданных.

Авторами предложен метод, позволяющий сократить число ложных конфликтов в STM-программах. Предполагается, что метаданные организованы в виде таблицы с прямой адресацией. Суть метода заключается в автоматической настройке параметров S и B таблицы под динамические характеристики конкретной STM-программы. Метод включает три этапа.

Этап 1. Инструментация транзакционных секций с целью профилирования. На первом этапе выполняется компиляция C/C++ STM-программы с использованием разработанного модуля инструментации транзакционных секций (модуль расширения GCC). В ходе статического анализа транзакционных секций STM-программ выполняется внедрение кода для регистрации обращений к функциям Intel TM ABI (_ITM_beginTransaction, _ITM_comitTransaction, _ITM_LU4, _ITM_WU4 и др.). Детали реализации модуля инструментации описаны ниже.

Этап 2. Профилирование программы. На данном этапе выполняется запуск STM-программы в режиме профилирования. Профилировщик регистрирует все операции чтения/записи памяти в транзакциях. В результате формируется протокол (trace), содержащий информацию о ходе выполнения транзакционных секций:

- адрес и размер области памяти, над которой выполняется операция;
- временная метка (timestamp) начала выполнения операции.

Этап 3. Настройка параметров таблицы. По протоколу определяются средний размер W читаемой/записываемой области памяти во время выполнения транзакций. По значению W подбираются субоптимальные параметры B и S таблицы, с которыми STM-программа компилируется.

Эксперименты с тестовыми STM-программами из пакета STAMP (6 типов STM-программ), позволили сформулировать эвристические правила для подбора параметров B и S по значению W.

Значение параметра S целесообразно выбирать из множества $\{2^{18}, 2^{19}, 2^{20}, 2^{21}\}$. Значение параметра B выбирается следующим образом:

- если $W = 1$ байт, то $B = 2^4$ байт;

- если $W = 4$ байт, то $B = 2^6$ байт;
- если $W = 8$ байт, то $B = 2^7$ байт;
- если $W >= 64$ байт, то $B = 2^8$ байт.

6. Инструментация транзакционных секций

STM-компилятор осуществляет трансляцию транзакционных секций в последовательность вызовов функций runtime-системы поддержки TM [15].

Компания Intel предложила спецификацию ABI для runtime-систем поддержки транзакционной памяти – Intel TM ABI [16]. Компилятор GCC, библиотека libitm, реализует этот интерфейс начиная с версии 4.8.

На рис. 6 представлен пример трансляции компилятором GCC транзакционной секции в обращения к функциям Intel TM ABI.

```
int a, b;
...
__transaction_atomic {
    if (a == 0)
        b = 1;
    else
        a = 0;
}
...
```

→

```
    ...
    state = _ITM_beginTransaction()
<L1>:
    if (state & a_abortTransaction)
        goto <L3>;
    else
        goto <L2>;
<L2>:
    if (_ITM_LU4(&a) == 0)
        _ITM_WU4(&b, 1);
    else
        _ITM_WU4(&a, 0);
    _ITM_commitTransaction();
<L3>:
    ...
```

Рис. 6. Инструментация транзакционной секции компилятором GCC.

В общем случае последовательность выполнения транзакции следующая:

1. Создание транзакции (вызов `_ITM_beginTransaction`) и анализ ее состояния. Если состояние транзакции содержит флаг принудительной отмены, то выполнение продолжается с метки `<L3>`, т.е. осуществляется выход из транзакции, иначе выполнение тела транзакции начинается с метки `<L2>`.

2. Выполнение транзакции. Если выполняется принудительная отмена транзакции, то в состоянии устанавливается флаг принудительной отмены (`a_abortTransaction`) и управление передается метке `<L1>`.

3. Попытка фиксации транзакции (вызов `_ITM_commitTransaction`), в случае возникновения конфликта транзакция отменяется, в состояние

транзакции записывается причина отмены и выполнение транзакции повторяется начиная с метки `<L1>`.

Разработанный модуль анализирует промежуточное представления GIMPLE транзакционных секций и добавляет функции регистрации обращений к функциям Intel TM ABI: регистрация начала транзакции и ее фиксации, транзакционное чтения/запись областей памяти. Функции регистрации заносят в протокол адреса и размер областей памяти, над которыми выполняются операции, а также время начала выполнения операций.

На рис. 7 представлен пример инструментации транзакционных секций. Функции с префиксом `tm_prof_` выполняют регистрацию событий.

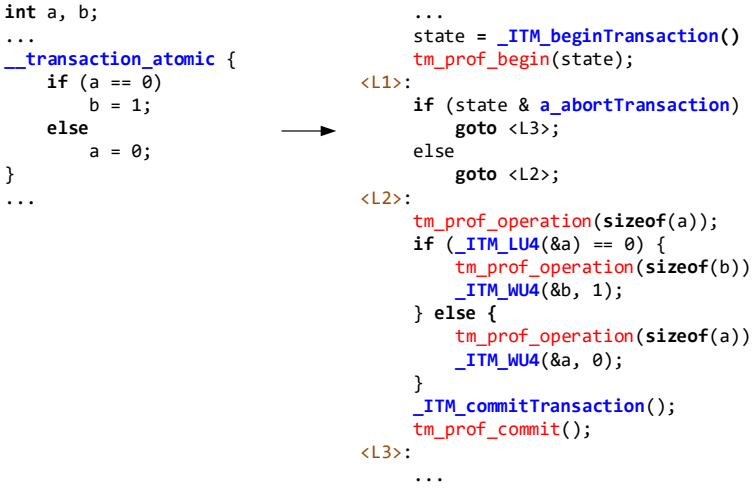


Рис. 7. Инструментация транзакционной секции.

Модуль инструментации в связке с библиотекой профилирования предоставляют достаточно сведений о динамических характеристиках транзакционных секций для того чтобы ответить на вопрос: «Фиксации каких транзакций или операции над какими данными приводят к отмене других транзакций?». Кроме этого, метод позволяет определить значения субоптимальных значений параметров реализации runtime-системы ТП, а именно число строк таблицы метаданных о состоянии областей памяти и количество адресов линейного адресного пространства, отображаемых на одну строку таблицы.

8. Эксперименты

Экспериментальное исследование проводилось на вычислительной системе, оснащенной двумя четырехъядерными процессорами Intel Xeon E5420. В

данных процессорах отсутствует поддержка аппаратной транзакционной памяти (Intel TSX).

В качестве тестовых программ использовались многопоточные STM-программы из пакета STAMP [9, 11, 12]. Число потоков варьировалось от 1 до 8. Тесты собирались компилятором GCC 5.1.1. Операционная система GNU/Linux Fedora 21 x86_64.

В рамках экспериментов измерялись значения двух показателей:

- время *t* выполнения STM-программы;
- количество *C* ложных конфликтов в программе.

На рис. 8 и 9 показана зависимость количества *C* ложных конфликтов и времени *t* выполнения теста от числа потоков при различных значениях параметров *B* и *S*. Результаты приведены для программы genome из пакета STAMP. В ней порядка 10 транзакционных секций, реализующих операции над хеш-таблицей и связными списками. Видно, что увеличение значений параметров *S* и *B* приводит к уменьшению числа возможных коллизий (ложных конфликтов), возникающих при отображении адресов линейного адресного пространства процесса на записи таблицы.

При размере таблицы 2^{21} записей, на каждую из которых отображается 2^6 адресов линейного адресного пространства, достигается минимум времени выполнения теста genome, а также минимум числа ложных конфликтов.

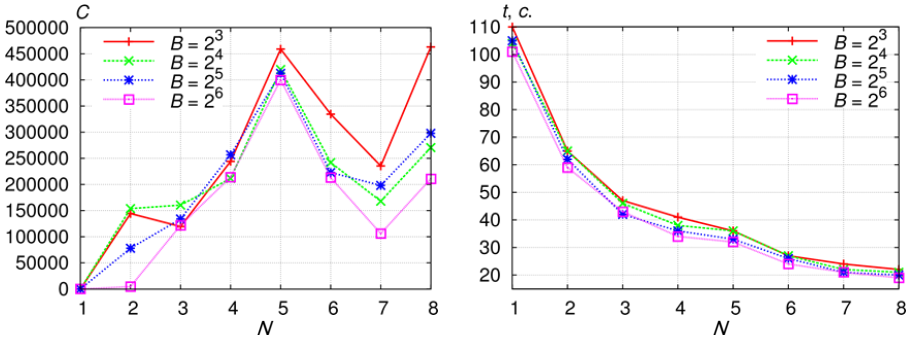


Рис. 8. Зависимость числа C ложных конфликтов (слева) и времени t выполнения теста (справа) от числа N потоков: $S = 2^{19}$.

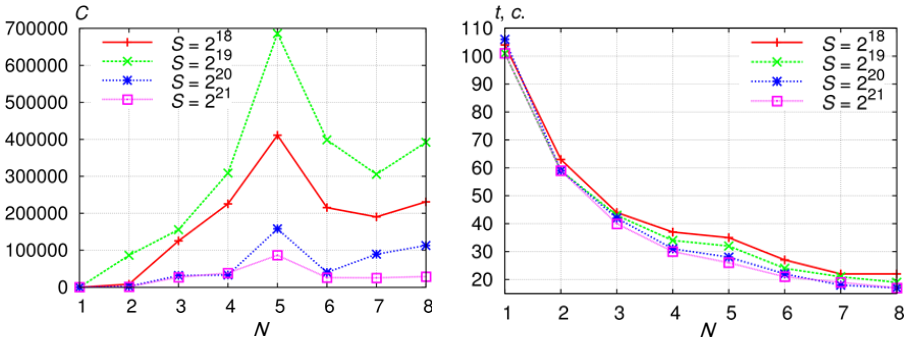


Рис. 9. Зависимость числа C ложных конфликтов (слева) и времени t выполнения теста (справа) от числа N потоков: $B = 2^6$.

Время выполнения теста genome удалось сократить в среднем на 20% за счет минимизации числа ложных конфликтов.

9. Заключение

В рамках данной работы создан модуль компилятора GCC для инструментации транзакционных секций STM-программ. Предложена библиотека профилирования STM-программ и оптимизации параметров внутренних структур данных runtime-библиотеки транзакционной памяти компилятора GCC (libitm) под конкретное приложение. Используя предложенный метод время выполнения теста genome удалось сократить на 20% за счет минимизации числа ложных конфликтов.

В будущем, планируется разработать алгоритмы выбора способа реализации программной транзакционной памяти во время компиляции программы. Дополнительно планируется провести исследование реализаций программной транзакционной памяти без централизованного хранения метаданных о состоянии областей памяти процесса.

Работа выполнена при поддержке РФФИ (гранты 15-37-20113, 15-07-00653), а также Министерства образования и науки Российской Федерации в рамках договора 02.G25.31.0058 от 12.02.2013.

Список литературы

[1]. M. Herlihy, N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[2]. Hendler D., Shavit N., Yerushalmi L. A scalable lock-free stack algorithm // Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. SPAA ’04. 2004. P. 206–215.

[3]. Кузнецов С.Д. Транзакционная память [HTML]. http://citforum.ru/programming/digest/transactional_memory/.

[4]. N. Shavit, D. Touitou. Software Transactional Memory. In PODC'95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA, Aug. 1995. ACM, 204–213.

[5]. Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel, Time-based Software Transactional Memory, IEEE Transactions on Parallel and Distributed Systems, Volume 21, Issue 12, pp. 1793-1807, December 2010.

[6]. Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation, 20th International Symposium on Distributed Computing (DISC), 2006.

[7]. Victor Luchango, Jens Maurer, Mark Moir. Transactional memory for C++ [PDF]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3718.pdf

[8]. Rochester Software Transactional Memory Runtime. Project web site [HTML]. www.cs.rochester.edu/research/synchronization/rstm/.

[9]. Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2009, P – 141-150.

[10]. Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In HPCA '06: Proc. 12th International Symposium on High-Performance Computer Architecture, February 2006, P. – 254-265.

[11]. Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: scalable transactions with a single atomic instruction. In SPAA '08: Proc. 20th Annual Symposium on Parallelism in Algorithms and Architectures, June 2008, P. 275–284.

[12]. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In DISC '06: Proc. 20th International Symposium on Distributed Computing, September 2006. Springer Verlag Lecture Notes in Computer Science volume 4167, P. – 194-208.

[13]. Pascal Felber, Christof Fetzer, Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. PPOPP 2008. P. – 237-246.

[14]. Craig Zilles and Ravi Rajwar. Implications of false conflict rate trends for robust software transactional memory. In IISWC '07: Proc. 2007 IEEE.

[15]. Olszewski M., Cutler J., Steffan J. G. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. PACT '07. 2007. P. 365–375.

[16]. Intel Corporation. Intel Transactional Memory Compiler and Runtime Application Binary Interface. Revision: 1.0.1, November 2008.

Instrumentation and Optimization of Transactional Sections Execution in Multithreaded Programs

[1]I. Kulagin <ivan.i.kulagin@gmail.com>
[2]M. Kurnosov <mkurnosov@gmail.com>
[1]SibSUTIS, 86 Kirova Str., Novosibirsk, 630102, Russian Federation
[2] Saint Petersburg Electrotechnical University "LETI", 5 Professora Popova Str., St. Petersburg, 197376, Russian Federation

Abstract. Software transactional memory (STM) is approach to develop thread-safe programs. In contrast to locking a block of code by mutex, the main idea of this approach is to protect memory areas from concurrent access by threads. In this paper, we investigate efficiency of software transactional memory implementation in GCC compiler. Software tools for instrumentation and profiling STM-programs are proposed. Profile-guided method for reducing false conflicts. in STM-programs is presented. False conflict is conflict that exist on the level of runtime library but not when the memory accessing happens. The instrumentation module analyzes the code of transactional sections and puts calls for registration of the some events (begin transactions, transactional read/write, commit transactions and abort transactions). The profiling of the instrumented program allows get dynamic properties of execution transactional code like size of used data, read/write addresses, timestamp of events, etc. The static instrumentation allows to optimize dynamic properties of execution transactional sections. The method of reducing false conflicts performs the tuning of transactional memory parameters value in GCC implementation (libitm runtime-library) by using the profiling results (profile-guided optimization). The efficiency of reducing false conflicts is investigated on the STAMP benchmarks. These benchmarks contains eight tests that operate with hash tables, lists, arrays protected by software transactional memory. Using the proposed method of the tests time is reduced approximately to 20%.

Keywords: software transactional memory; instrumentation; profile-guided optimization; multithreaded programming; compilers.

DOI: 10.15514/ISPRAS-2015-27(6)-9

For citation: Kulagin I.**,** Kurnosov M. Instrumentation and Optimization of Transactional Sections Execution in Multithreaded Programs. *Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015*, pp. 135-150 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-9

References

[1]. M. Herlihy, N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[2]. Hendler D., Shavit N., Yerushalmi L. A scalable lock-free stack algorithm // Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. SPAA '04. 2004. P. 206–215.

[3]. Kuznetsov S.D. Transaktsionnaya pamat. [Transactional memory]. http://citforum.ru/programming/digest/transactional_memory/. (in Russian).

[4]. N. Shavit, D. Touitou. Software Transactional Memory. In PODC'95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA, Aug. 1995. ACM, 204–213.

[5]. Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel, Time-based Software Transactional Memory, IEEE Transactions on Parallel and Distributed Systems, Volume 21, Issue 12, pp. 1793-1807, December 2010.

[6]. Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation, 20th International Symposium on Distributed Computing (DISC), 2006.

[7]. Victor Luchango, Jens Maurer, Mark Moir. Transactional memory for C++ [PDF]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3718.pdf

[8]. Rochester Software Transactional Memory Runtime. Project web site [HTML]. www.cs.rochester.edu/research/synchronization/rstm/.

[9]. Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2009, P – 141-150.

[10]. Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In HPCA '06: Proc. 12th International Symposium on High-Performance Computer Architecture, February 2006, P. – 254-265.

[11]. Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: scalable transactions with a single atomic instruction. In SPAA '08: Proc. 20th Annual Symposium on Parallelism in Algorithms and Architectures, June 2008, P. 275–284.

[12]. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In DISC '06: Proc. 20th International Symposium on Distributed Computing, September 2006. Springer Verlag Lecture Notes in Computer Science volume 4167, P. – 194-208.

[13]. Pascal Felber, Christof Fetzer, Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. PPOPP 2008. P. – 237-246.

[14]. Craig Zilles and Ravi Rajwar. Implications of false conflict rate trends for robust software transactional memory. In IISWC '07: Proc. 2007 IEEE.

[15]. Olszewski M., Cutler J., Steffan J. G. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. PACT '07. 2007. P. 365–375.

[16]. Intel Corporation. Intel Transactional Memory Compiler and Runtime Application Binary Interface. Revision: 1.0.1, November 2008.

Использование различных представлений java-программ для статического анализа

Е.А. Карпулевич <karpulevich@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва,
ул. А. Солженицына, дом 25.

Аннотация. Статический анализ исходного кода используется для автоматизированного обнаружения дефектов программного обеспечения. Особо ощутима польза статического анализа при разработке больших проектов, состоящих из сотен тысяч строк кода, поскольку такой объем кода практически невозможно проверить вручную.
Статический анализатор, в отличие от компилятора, не так сильно ограничен по времени. Благодаря этому, можно реализовать более сложные и точные алгоритмы, которые выдают больше истинных и меньше ложных срабатываний, чем алгоритмы анализа компилятора. В основе работы любого
алгоритма лежит внутреннее представление программного кода. В статье рассматриваются различные варианты внутреннего представления программ и детекторы программных ошибок, работающие на этих представлениях. Анализ внутреннего представления в виде абстрактного синтаксического дерева (АСТ) позволяет быстро обнаруживать несложные ошибки, например опасное преобразование типов. С помощью абстрактного синтаксического дерева удобно искать ошибки, связанные с повторным использованием кода. Анализ графа потока управления (ГПУ) позволяет находить более сложные ошибки, для обнаружения, которых требуется проход по коду программы. Вместо прохода по коду анализ выполняется с помощью обхода ГПУ. С помощью анализа на ГПУ можно обнаружить такие дефекты, как, например, утечка ресурса, повторное освобождение ресурса, переполнение буфера.
Существуют и другие внутренние представления, на которых удобно проводить определенные классы анализов. В статье, в качестве примера, описаны принципы работы нескольких детекторов анализатора SVACE на соответствующих внутренних представлениях.

Ключевые слова: статический анализ, java, FindBugs, SVACE

DOI: 10.15514/ISPRAS-2015-27(6)-10

Для цитирования: Карпулевич Е.А. Использование различных представлений java-программ для статического анализа. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 151-158. DOI: 10.15514/ISPRAS-2015-27(6)-10.

1. Введение

В настоящее время программный код в больших проектах исчисляется миллионами строк кода. Неудивительно, что в таких проектах присутствует огромное количество ошибок разной степени критичности. Обнаружить среди миллионов строк несколько строк кода, содержащих дефект, непросто, а отследить утечку ресурса или возможное разыменование нулевого указателя вручную иногда практически невозможно. Для автоматизированной проверки программ и выдачи предупреждений об ошибках необходим инструмент, специализирующийся на поиске ошибок - статический анализатор. Для языка программирования Java существует несколько известных статических анализаторов: FindBugs[1], SVACE[2], Jlint[3] и др.

Во время работы компилятор также может выдавать большое количество предупреждений о потенциальных дефектах, однако, большое количество из них не приводят к сбоям в работе программы (например, когда в новой версии компилятора функция объявлена устаревшей). Кроме того, анализ производимый во время компиляции ограничен по времени и ресурсам, так как основная задача java-компилятора - предоставить байткод максимально быстро.

Для получения адекватных предупреждений об ошибках при анализе программ важны быстрые и корректные алгоритмы поиска ошибок, продуманная архитектура анализатора и информативное промежуточное представление исходного кода программы, на котором строится весь анализ[4].

Различные промежуточные представления подходят для выявления в коде программы определенных типов ошибок. В статье проводится сравнение нескольких внутренних представлений в контексте статического анализа.

2. Получение различных внутренних представлений

Построение промежуточных представлений статического анализатора схоже с построением промежуточных представлений при компиляции исходного кода. На этапе лексического анализа исходного кода компилятор разбирает исходный код на последовательность лексем, из которых формирует абстрактное синтаксическое дерево. После этого происходят оптимизации на уровне абстрактного синтаксического дерева. На этапе кодогенерации по абстрактному синтаксическому дереву генерируется java-байткод. Для проведения анализа программы и дальнейшей её оптимизации по байткоду строится граф потока управления, граф вызовов.

Для статического анализа программ используются все перечисленные представления. На основе анализа байткода работают детекторы поиска

ошибок по шаблонам, на абстрактном синтаксическом дереве – детекторы клонов кода и поиск ошибок несоответствия отступов. Построение графа потока управления не является необходимым этапом статического анализа. Но существует класс ошибок, которые неудобно искать, анализируя абстрактное синтаксическое дерево. Такие ошибки как утечки памяти и разыменование нулевых указателей позволяет обнаружить анализ графа потока управления.

3. Поиск ошибок через анализ байткода и абстрактного синтаксического дерева.

Большая часть предупреждений FindBugs является результатом работы алгоритмов поиска ошибок на байткоде по характерным шаблонам. С помощью библиотеки ASM[5] происходит обход инструкций байткода.

Библиотека ASM позволяет работать с байткодом, она позволяет считывать байткод, а также предоставляет возможность по генерации и модификации байткода классов на лету.

Простой пример поиска по байткоду – детектор FindBugs ICAST_INTEGER_MULTIPLY_CAST_TO_LONG. Этот детектор проверяет на возможное переполнение типа integer до расширения до long. Такая ошибка содержится, например, в следующем фрагменте кода:

*long convertDaysToMilliseconds(int days) { return 1000*3600*24*days; }*

Чтобы найти ошибку такого рода достаточно проверить наличие в байткоде последовательности из двух инструкций IMUL (умножение двух целых) I2L(преобразование в long). Исправленный код может выглядеть так:

*long convertDaysToMilliseconds(int days) { return 1000L*3600*24*days; }*

Сигнатурный поиск по байткоду позволяет найти достаточно простые ошибки, такие как проверка знака битовой операции и ошибка форматирования строки и пр. Алгоритмы поиска шаблонов таких ошибок хорошо работают на внутреннем представлении в виде последовательности команд байткода. Но существуют ошибки, для поиска которых анализ байткода не является достаточным или является крайне неэффективным. Например, для поиска ошибок повторного использования (клонов) кода лучше подходит абстрактное синтаксическое дерево. Для работы алгоритмов поиска утечек ресурсов необходима информация, которую можно восстановить из байткода, но которая в нем не содержится: где ресурс выделяется и освобождается, функция какого объекта вызывается. Также, для поиска утечек ресурсов требуются пользовательские спецификации, полная или частичная девиртуализация и межпроцедурный анализ.

Алгоритмы, реализованные на байткоде можно успешно реализовать на абстрактном синтаксическом дереве, однако реализация алгоритмов может стать значительно сложнее.

Рассмотрим детектор ICAST_INTEGER_MULTIPLY_CAST_TO_LONG. Для того чтобы найти ошибку переполнения в абстрактном синтаксическом дереве

необходимо проверять типы, значения и эмулировать арифметические операции, требуется значительно больше усилий.

Задачи из класса обнаружения клонов кода наоборот проще решаются на основе анализа абстрактного синтаксического дерева или анализа графа программных зависимостей[6]. Например, поиск ошибки повторного использования на байткоде затруднен необходимостью восстанавливать структуру условных операторов и операторов цикла, отсутствием информации об отступах и позициях идентификаторов.

Ошибки повторного использования возникают при дублировании кода копированием с внесением последующих изменений. Один из вариантов такой ошибки – копирование условного оператора с последующей заменой одной переменной на другую. Иногда программист успешно меняет копию фрагмента кода в пяти местах, а в одном забывает.

Чтобы найти ошибку повторного использования необходимо проверить похожесть двух фрагментов кода и провести анализ идентификаторов на наличие неполных замен. Замена идентификатора A будет неполной если везде, кроме одного места, идентификатор A заменен на идентификатор B.

Кроме того, необходимо каким-то образом выявить похожие фрагменты кода, на которых и будет работать алгоритм. Для поиска похожих фрагментов кода в лоб можно использовать суффиксные деревья для поиска двух одинаковых подстрок в последовательности лексем, полученных из исходного кода. Но такой поиск достаточно медленный (сложность построения суффиксного дерева O(n), проверка похожих частей O(m)), так как поиск идет по всему коду файла.

Около 40% случаев копирования кода – копирование базовых блоков и функций[7]. Программист копирует небольшую функцию, условный оператор целиком, одно из условий условного оператора, заголовок цикла или весь цикл, заменяя один или несколько идентификаторов. К тому же чаще всего вставка происходит рядом копированием. Полная информация о таких фрагментах кода и о порядке их следования содержится в абстрактном синтаксическом дереве. В статическом анализаторе SVACE поиск повторного использования кода реализован через поиск похожих частей кода на абстрактном синтаксическом дереве (сложность обнаружения фрагментов для проверки O(1), проверка похожих частей O(m)) с последующим анализом на повторное использование. В поиске повторного использования в анализаторе SVACE учитывается не только похожесть соседних фрагментов кода, но и соответствие отступов строк в них.

4. Анализ графа потока управления

Для поиска ошибок разыменования нулевого указателя или поиска утечки ресурса требуется анализировать пути исполнения программы. В таком случае можно использовать граф потока управления. Алгоритмы поиска ошибок на графе потока управления похожи на алгоритмы компиляторного анализа.

Анализ графа потока управления используется в статическом анализаторе SVACE, что позволяет анализатору находить утечки ресурсов. Для корректной работы алгоритма поиска утечек необходима возможность хранения и передачи межпроцедурной информации, так как часто ресурс выделяется в одной функции, а используется и освобождается в других. В статическом анализаторе SVACE после анализа функции составляется и сохраняется её аннотация (информация о поведении функции). Кроме того пользователю необходимо указать какие функции выделяют, а какие закрывают ресурсы. Для этих целей в SVACE существует возможность добавлять пользовательские спецификации, описывающие поведение функций.

Поиск утечки ресурса идет в процессе обхода графа управления. Фиксируются моменты выделения и освобождения ресурса и, если было выделение ресурса а после не было его освобождения, выдается предупреждение об утечке.

В языке java все нестатические неприватные (то есть, protected, package и public) методы являются виртуальными. Для анализа графа потока управления на предмет ошибок нужно понимать метод какого класса вызывается в коде. Провести точную девиртуализацию не всегда представляется возможным. В таком случае можно прибегнуть к частичной девиртуализации. При частичной девиртуализации вместо одного кандидата методу соответствует некоторый набор кандидатов. Точность анализа при частичной девиртуализации для ряда детекторов снижается, но и время необходимое для девиртуализации сокращается. Понимать какой метод вызывается нужно, в том числе, для поиска утечек. В java возможна ситуация, когда базовый класс не выделяет ресурс в отличие от своих потомков. В этом случае необходима девиртуализация, чтобы понять был ли выделен ресурс.

5. Поиск ошибок в многопоточных программах

Многие программы выполняются в несколько потоков с использованием примитивов синхронизации. При таком выполнении в процессе работы программы могут возникать взаимные блокировки или состояние гонки. Для задачи поиска взаимных блокировок граф потока управления является слишком громоздким и содержит информацию, которая не пригодится для поиска ошибок синхронизации. В инструменте Jlint много внимания уделили многопоточности. В качестве внутреннего представления Jlint использует граф зависимостей блокировок[8], для того чтобы алгоритмы поиска ошибок синхронизации были проще и точнее.

6. Объединение представлений

Существуют и другие внутренние представления программ, например граф программных зависимостей и граф вызовов. Они подходят для других задач, например, граф программных зависимостей удобен для поиска клонов кода.

Наличие нескольких представлений, на которых можно реализовать анализы различных классов ошибок заставляет задуматься о создании статического анализатора с несколькими внутренними представлениями. И действительно, многие анализаторы работают с несколькими представлениями. Например, статический анализатор SVACE ищет ошибки с помощью алгоритмов работающих на представлениях в виде байткода, графа вызовов, графа потока управления и др.

Кроме простого наличия нескольких представлений важна возможность безболезненного перехода от одного внутреннего представления к другому. При наличии такого перехода возможно не только повысить информативность навигации и сообщений об ошибках, но и делать несколько анализов на предмет одной ошибки на различных представлениях, а потом объединять результаты.

Очень важна связь между исходным кодом и внутренними представлениями программы. В процессе компиляции исходного кода в байткод происходит потеря некоторой части полезной информации о программе, например, полностью пропадает информация о наличии и количестве пробельных символов. Анализ ошибок такого рода можно проводить только на этапе компиляции.

В компиляторе javac есть внутреннее представление исходного кода в виде абстрактного синтаксического дерева. В таком случае достаточно интересной выглядит идея построить статический анализатор кода на основе компилятора, добавив в него несколько дополнительных внутренних представлений и реализацию алгоритмов для поиска ошибок.

В код компилятора javac успешно добавлены детекторы, работающие на абстрактном синтаксическом дереве, обнаруживающие ошибки повторного использования, ошибки выбора объекта для синхронизации, одинаковые ветки в условном или тернарном операторе. Полнота (отношение количества обнаруженных ошибок к общему количеству предупреждений) анализа для этих детекторов составила около 80%.

Список литературы

[1]. FinBugs – http://findbugs.sourceforge.net/findbugs2.html

[2]. В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды Института системного программирования РАН Том 26. Выпуск 1. 2014 г. Стр. 231-250.

[3]. Cyrille Artho. Finding faults in multi-threaded programs. March 15, 2001. (http://artho.com/jlint/mthesis.pdf)

[4]. Nick Rutar, Christian B. Almazan, Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. (http://www.cs.umd.edu/~jfoster/papers/issre04.pdf)

[5]. ASM framework – http://asm.ow2.org/index.html

[6]. Sevak Sargsyan, Shamil Kurmangaleev, Vahagn Vardanyan, Vachagan Zakaryan. Code Clones Detection Based on Semantic Analysis for JavaScript Language. October 1, 2015 (https://csit.am/2015/9a.html)
[7]. Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. (http://opera.ucsd.edu/paper/OSDI04-CPMiner.pdf)
[8]. Jurgen Graf, Martin Hecker, Martin Mohr, and Benedikt Nordhoff. Lock-sensitive Interference Analysis for Java: Combining Program Dependence Graphs with Dynamic Pushdown Networks. 2013. (https://pp.ipd.kit.edu/uploads/publikationen/pdgwithdpn2013id.pdf)

Using Different Views Java-Programs for Static Analysis

E.A. Karpulevitch <karpulevich@ispras.ru>
Institute for System Programming of the RAS,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Absract. Static analysis of the source code used for the automated detection of software defects. Particularly noticeable benefits of static analysis in the development of large projects, consisting of hundreds of thousands of lines of code, because this amount of code is almost impossible to check manually.
Static analyzer of the compiler in contrast, not so much limited in time. Because of this, you can implement more complex and accurate algorithms that give more truth, and less false positives than the compiler's analysis algorithms. At the heart of any algorithm is an internal representation of the program code. The article discusses the various options for the internal representation of programs and software bug detectors that work on these ideas. Analysis of the internal representation of an abstract syntax tree (AST) allows you to quickly detect simple errors, such as a dangerous type conversions. By using abstract syntax tree is convenient to look for errors associated with re-use of code. An analysis of the control flow graph (CFG) allows you to find a more sophisticated error detection which requires passage by the program code. Instead pass code analysis is executed using the CFG bypass. Through analysis of the CFG can detect defects such as, for example, a resource leak, double release of the resource, buffer overflow. There are also other internal representations, which is convenient to carry out certain tests classes. The article, by way of example, the principles of operation described SVACE analyzer several detectors corresponding internal representations.

Keywords: static analysis, java, FindBugs, SVACE

DOI: 10.15514/ISPRAS-2015-27(6)-10

For citation: Karpulevitch E.A. Using Different Views Java-Programs for Static Analysis. *Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015,* pp. 151-168 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-10

References

[1]. FinBugs – http://findbugs.sourceforge.net/findbugs2.html

[2]. V.P. Ivannikov, A.A. Belevancev, A.E. Borodin, V.N. Ignat'ev, D.M. Zhurikhin, A.I. Avetisjan, M.I. Leonov. Staticheskij analizator Svace dlja poiska defektov v iskhodnom kode programm [Svace: static analyzer for detecting of defects in program source code] Trudy ISP RAN [The Proceedings of ISP RAS], volume 26, issue 1, pp. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7. (in Russian)

[3]. Cyrille Artho. Finding faults in multi-threaded programs. March 15, 2001. (http://artho.com/jlint/mthesis.pdf)

[4]. Nick Rutar, Christian B. Almazan, Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. (http://www.cs.umd.edu/~jfoster/papers/issre04.pdf)

[5]. ASM framework – http://asm.ow2.org/index.html

[6]. Sevak Sargsyan, Shamil Kurmangaleev, Vahagn Vardanyan, Vachagan Zakaryan. Code Clones Detection Based on Semantic Analysis for JavaScript Language. October 1, 2015 (https://csit.am/2015/9a.html)

[7]. Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. (http://opera.ucsd.edu/paper/OSDI04-CPMiner.pdf)

[8]. Jurgen Graf, Martin Hecker, Martin Mohr, and Benedikt Nordhoff. Lock-sensitive Interference Analysis for Java: Combining Program Dependence Graphs with Dynamic Pushdown Networks. 2013. (https://pp.ipd.kit.edu/uploads/publikationen/pdgwithdpn2013id.pdf)

Использование ABI для интроспекции виртуальных машин

Н.И. Фурсова <natalia.fursova@ispras.ru>
П.М. Довгалюк <pavel.dovgaluk@ispras.ru>
И.А. Васильев <ivan.vasiliev@ispras.ru>
Новгородский государственный университет имени Ярослава Мудрого,
173000, Россия, г. Великий Новгород, ул. Лазаревская, дом 11

Аннотация. В статье предлагается подход к интроспекции виртуальных машин с использованием двоичного интерфейса приложений. Основная цель метода - получать информацию о работе системы, имея минимальные знания об ее внутреннем устройстве. Наша система основана на эмуляторе QEMU и имеет модульное строение, единицей в котором является плагин.

Существующие подходы (RTKDSM, DECAF) получают данные из операционной системы с помощью структур ядра. Эти инструменты вынуждены хранить большое количество профилей с данными, потому что все адреса и смещения в структурах меняются от версии к версии. Мы предлагаем использовать редко изменяющиеся части двоичного интерфейса приложений, такие как соглашения о вызовах и номера и параметры системных вызовов. Основная идея метода - перехватывать системные функции и считывать параметры и возвращаемые значения.

Для осуществления системного вызова у процессора есть специальная инструкция. Расширив возможности QEMU механизмом инструментирования, мы имеем возможность отслеживать каждую выполняющуюся инструкцию и отфильтровывать нужную. При возникновении системного вызова мы передаем управление в детектор системных вызовов, который проверяет номер произошедшего вызова и, в соответствии с ним, принимает решение какому плагину перенаправить это задание.

В механизме перехвата системных вызовов важно не только определить что вызов произошел, но и корректно определить его завершение, чтобы считать значения выходных параметров и возвращаемое значение. Для окончания системного вызова тоже есть специальные инструкции, но нам так же нужно верно сопоставить начало вызова с его концом, для чего мы определяем текущий контекст.

Таким образом мы реализовали мониторинг файловых операций, процессов и создали прототип монитора API функций.

Мы планируем расширить набор плагинов для анализа и мониторинга. Наша система будет извлекать информацию о загруженных модулях, приложениях, а также отладочную информацию.

Ключевые слова: интроспекция; виртуальные машины; динамический анализ; системные вызовы.

DOI: 10.15514/ISPRAS-2015-27(6)-11

Для цитирования: Фурсова Н.И., Довгалюк П.М., Васильев И.А. Использование ABI для интроспекции виртуальных машин. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 159-168. DOI: 10.15514/ISPRAS-2015-27(6)-11.

1. Введение

Динамический анализ - это важная технология для исследования программного обеспечения (ПО). Он используется для профилирования, анализа вредоносного кода, обнаружения вторжений, тестирования ПО и для многого другого.

Полносистемный анализ дает представление о всей системе. Он может помочь анализировать деятельность ядра ОС и взаимодействие между процессами, коммуникацию с аппаратным обеспечением или определить поведение вредоносного кода без влияния на работу системы.

Для реализации анализа мы используем интроспекцию. Интроспекция - это извлечение данных из операционной системы, которые она использует для своей работы и которые скрыты от пользователя. Такими данными могут быть идентификаторы процессов и потоков, значения переменных, содержимое памяти. Большое количество этих данных сосредоточены в структурах ядра системы.

Основная идея нашего метода - использование минимальных знаний о системе, которые не включают структуры ядра. Мы будем получать информацию с разных уровней работы системы, используя лишь данные с открытой структурой. Наш подход предполагает модульное строение, единицей в котором является плагин, выполняющий определенные функции.

Наша работа основана на мультиплатформенном симуляторе QEMU [1]. Мы расширили QEMU новой функциональностью, позволяющую загружать внешние плагины и производить динамическое инструментирование. Мы так же создали несколько плагинов для мониторинга системных вызовов, файловых операций и процессов.

2. Обзор существующих подходов

Рассмотрим несколько исследований на тему инструментирования и полносистемного анализа.

Real-time kernel data structure monitoring (RTKDSM) использует возможности анализа Volatility, является фреймворком с открытым исходным кодом, упрощает и автоматизирует анализ состояний выполняющейся виртуальной машины [3]. Система RTKDSM выполняет мониторинг состояния операционной системы в виртуальной машине в реальном времени. Она использует Xen как монитор виртуальной машины и некоторые плагины

Volatility. Volatility специализируется на анализе дампов памяти. RTKDSM анализирует память гостевой системы напрямую, не используя дампы.

Основанная на дампах архитектура Volatility не эффективна в режиме мониторинга, поскольку нужно анализировать целый дамп каждый раз, как только понадобится информация. Таким образом, RTKDSM использует Volatility только для поиска структур данных исследуемых операционных систем. После того как адреса структур получены, RTKDSM использует собственный агент мониторинга, который отслеживает изменения в этих структурах.

DECAF это платформо-независимый полносистемный фреймворк для динамического анализа [2]. DECAF расшифровывается как Dynamic Executable Code Analysis Framework. Он разработан на основе QEMU и предоставляет интерфейс для программирования анализирующих плагинов. DECAF восстанавливает семантику уровня операционной системы с помощью интроспекции. Восстановление происходит, когда такой запрос приходит с уровня аппаратного обеспечения.

Инструмент RTKDSM основан на Xen, что ограничивает его работу архитектурой x86. DECAF является мощным средством динамического анализа, имеет модульную структуру и открытый исходный код, что позволяет пользователям дополнять его нужными функциями, однако имеет некоторые недостатки. Во-первых, DECAF основан на QEMU версии 1.0, что не позволяет запускать новые операционные системы, такие как Windows 7-8. Во-вторых, метод получения данных у этого инструмента - исследование структур ядра системы, из-за чего ему необходимо генерировать и хранить профили для каждой версии исследуемой операционной системы.

3. Поход и уникальность

Мы предлагаем новый подход к интроспекции, который не требует модификаций гостевой операционной системы и приложений и сделает зависимые от операционной системы части минимальными. Например, DECAF вынужден иметь профиль на каждую версию операционной системы с указанием всех адресов структур и смещений в этих структурах, что не очень удобно, потому что адреса и смещения отличаются для разных версий и сборок ядра ОС.

Мы предлагаем использовать для извлечения данных из операционной системы некоторые части ABI (Application Binary Interface). ABI - это интерфейс, описывающий взаимодействие между операционной системой, библиотеками и приложениями. Как правило, в ABI входят такие сведения как соглашения о вызовах, формат исполняемых и библиотечных файлов, количество и формат системных вызовов и другое. ABI редко подвергается изменениям что позволяет получать данные, используя минимальное количество параметров. Это свойство важно, если речь идет о встроенных системах, в которые невозможно загрузить приложение, которое считает из

системы все необходимые смещения (таким образом работает DECAF). Для работы нашего метода используются такие части ABI как соглашения о вызовах и системные вызовы (их номера и параметры).

Одной из важных составляющих нашей системы является перехватчик системных вызовов. Системный вызов это запрос из операционной системы к ядру. Для этих запросов обычно предусмотрены специальные инструкции (например, syscall для x86/64 или svc #0 для ARM). Эти инструкции переводят процессор в режим ядра и выполняют соответствующий системному вызову код. Перехватчик системных вызовов это плагин, процесс работы которого представлен на рисунке 1.

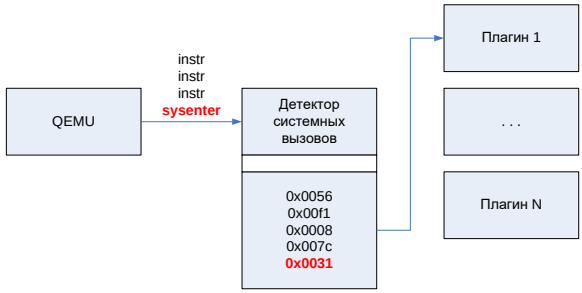


Рис. 1. Перехватчик системных вызовов.

С помощью перехвата системных вызовов мы имеем возможность отслеживать файловые операции, создание новых процессов и потоков, создание объектов ядра, операций отображения в память и другие.

Для выполнения требований о минимальном вмешательстве мы используем виртуальную машину и исследуем ее память. Также используются зависимые от ABI части кода для преобразования платформо-зависимых данных в платформо-независимое представление.

Мы используем наш подход для реализации алгоритмов анализа. Каждый алгоритм представляется в виде плагина. В итоге мы получаем систему с модульным строением, где каждая единица отвечает за один из видов анализа. Плагины могут работать на разных уровнях, таких как уровень операционной системы, уровень аппаратного обеспечения. Информация извлекается из исполняемого кода.

Плагины на разных уровнях взаимодействуют посредством сообщений. Сообщения соответствуют событиям, которые обозначают внешние коммуникации или изменения в состоянии виртуальной машины.

Аппаратный уровень включает события и данные, генерируемые симулятором, например, может быть вызвана функция обратного вызова, когда произошло прерывание, трансляция инструкции, выполнение инструкции, получение сетевого пакета, чтение/запись ячеек памяти и так далее. Эти события могут быть использованы для построения плагинов,

которые исследуют системные вызовы, адресные пространства и контекст исполнения.

Следующий уровень включает плагины для мониторинга файлов, процессов, потоков и других объектов операционной системы. Файловые операции могут быть исследованы путем перехвата соответствующего системного вызова, который рассматривался на предыдущем уровне. Создание процессов может исследоваться таким же способом.

Уровень приложений содержит плагины, которые исследуют модули, приложения и другие объекты уровня пользователя. Плагины уровня исходных кодов извлекают символьную и отладочную информацию для предоставления понятной пользователю информации об отладке и анализе приложения.

3.1 Мониторинг файловых операций

Мы создали плагин для мониторинга файлов. Этот плагин не зависит от гостевой операционной системы и гостевого аппаратного обеспечения.

Так как Windows и Linux имеют разные наборы системных вызовов, мы создаем плагин для перехвата системных вызовов для каждой операционной системы.

Для анализа файловых операций перехватываются следующие функции: NtCreateFile, NtOpenFile, NtReadFile, NtWriteFile, NtClose в Windows и creat, open, read, write, close в Linux.

Когда плагин обнаруживает платформо-зависимый системный вызов, он преобразовывает параметры и возвращает данные системного вызова в платформо-независимой форме.

Мы извлекаем следующие параметры операций: Create/Open file (возвращается хэндл файла, имя файла, тип доступа), Close file (хэндл файла), Read/Write file (хэндл файла, адрес буфера, количество прочитанных/записанный байт). Пример полученного абстрактного журнала представлен на рисунке 2.

```
open(name \SystemRoot\bootstat.dat, mode READ | WRITE,
      handle 0x1c)
read(handle 0x1c, buffer 0x15f88c, length 0x4)
write(handle 0x1c, buffer 0x15f8cb, length 0x1)
```

Рис. 2.Фрагмент абстрактного журнала.

Так же для каждой операционной системы записывается подробный лог всех вызывавшихся функций с полным набором параметров. Платформо-независимый вариант нужен для реализации единого механизма анализа лога, работающего со всеми операционными системами.

Плагин для системных вызовов использует две функции обратного вызова для обработки системного вызова. Первая функция обратного вызова требуется,

чтобы проверить какой системный вызов произошел и соответствует ли он нужному. Вторая функция обратного вызова выполняется, когда системная функция завершилась. Он читает возвращаемые значения и выходные параметры функций.

Механизм вызова функции обратного вызова по завершению выполнения функции с целью получения возвращаемых значений является одной из задач нашей работы. Для получения возвращаемого значения системного вызова мы вызываем функцию обратного вызова после выполнения системной функции. Для выполнения гостевого кода QEMU использует динамическую трансляцию. Мы не можем просто вставить любой код после выполнения инструкции в этом же блоке трансляции, потому что выполнение текущего блока закончится и начнется выполнение следующего блока, что лишит нас возможности отследить завершение системного вызова.

Для решения этой задачи мы используем контекст выполнения. Функции обратного вызова вызываются перед каждой инструкцией, и мы можем запомнить некоторые данные на старте системного вызова и затем сверять их с текущими данными на выходе. Для определения начала и окончания выполнения системного вызова в архитектуре x86 используются регистры esp и cr3.

3.2 Мониторинг процессов

Системные вызовы для процессов разные у разных операционных систем, поэтому большинство их параметров мы не можем унифицировать. Мы трассируем наиболее важные данные, доступные во всех реализациях системных вызовов.

Процессы в Linux могут быть созданы с помощью функций fork или clone. Windows использует NtCreateProcess, которая сама вызывает NtCreateThread. Поскольку функции для создания процессов разные, то привести их к общему виду является нетривиальной задачей. На данном этапе для каждой операционной системы свой плагин и абстрактный журнал для процессов не пишется.

Одним из очень важных параметров при создании процессов и работе с ними является идентификатор процесса. В Windows на уровне системных вызовов в большинстве функций идентификатор заменен хэндлом процесса, в Linux же возвращается настоящий идентификатор процесса. Первой идеей было использовать хэндл вместо идентификатора, однако выяснилось, что хэндл ненадежный источник: он может закрыться, а процесс продолжит жить или он может быть продублирован, поэтому было решено найти идентификатор процесса и для Windows. Функция NtCreateThread заполняет структуру ClientID, которая содержит идентификатор процесса и потока.

Для того, чтобы использовать идентификаторы для анализа системы и приложений необходимо связать их с адресным пространством. На данный момент мы работаем с архитектурой x86 и используем для этой цели регистр

cr3. Процесс создается из контекста другого процесса, поэтому на первом этапе мы сможем связать только cr3 создающего процесса с идентификатором создаваемого. Эта схема представлена на рисунке 3.

Рис. 3.Связь cr3 и идентификатора процесса.

Для того чтобы решить проблему с контекстом было решено использовать системный вызов NtQueryProcessInfo, который тоже при определенных условиях может запрашивать структуру ClientID. Если эта системная функция вызывается для текущего процесса, то мы можем корректно соотнести адресное пространство с идентификатором процесса. На данный момент непонятно, достаточно ли этих данных, поэтому мы рассмотрели возможность использования альтернативных путей получения этой информации, например, встраиваемые системные вызовы. Это значит, что мы будем вызывать некоторые системные вызовы в тот момент, когда необходимо будет получить информацию о процессе, например, NtQueryProcessInfo/getpid. Однако такой подход возможен только с использованием записи/воспроизведения [4], потому что встраивания изменят состояние системы.

3.3 Мониторинг API функций

Следующим источником получения данных являются API функции. Исследуя их выполнение, мы можем получать доступ к более высокоуровневой информации.

Для того, чтобы получить доступ к этим функциям нужно определить базовый адрес, по которому загрузится dll, а также необходимо знать смещения в файле, по которым расположены адреса функций из библиотеки.

Мы пробовали решить эту задачу на примере библиотеки kernel32.dll с помощью системных вызовов. В загрузке dll файлов обычно принимают участие три системных вызова: NtOpenFile, NtCreateSection, NtMapViewOfSection. Выходным параметром последней системной функции является исковый базовый адрес. Теперь, зная адрес, можно определить смещения интересующих функций и, используя выражение "базовый адрес + смещение" получить адрес функции. Принцип работы с адресами может быть такой же, как с системными вызовами, только вместо опкодов инструкций рассматривать адреса функций и, при переходе на заданный адрес, передавать управление в нужный плагин.

4. Заключение

В статье описан подход для интроспекции виртуальных машин. Мы предлагаем анализировать операционные системы и приложения в виртуальной машине с минимальными вмешательствами в их работу. Для

этого мы используем некоторые аспекты ABI, а именно коды системных вызовов и соглашения о передаче их параметров. Мы создали основу нашей системы - перехватчик системных вызовов для файловых операций и процессов. Перехватчик системных вызовов анализирует инструкции симулятора и определяет в потоке системные вызовы. Монитор файлов может быть использован для трассировки файловых операций в операционных системах Windows и Linux.

Мы планируем расширить набор плагинов для анализа и мониторинга. Плагины должны предоставлять возможности для интроспекции для операционных систем. Наша система будет извлекать информацию о загруженных модулях, приложениях, вызовах API функций, а также отладочную информацию.

Одной из возможностей улучшения инструмента анализа является анализ оффлайн, основанный на записи сценария работы виртуальной машины. У нас уже есть реализация записи/воспроизведения в QEMU [4]. Анализ будет выполняться в процессе воспроизведения и не окажет никакого влияния на систему, потому что все входные данные записаны в файле журнала. Соединив запись/воспроизведение с подходом к интроспекции, мы получим мощный инструмент анализа без оказания влияния на работу системы.

Список литературы

[1]. F. Bellard, QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, pages 41-41, Berkeley, CA, USA, 2005. USENIX Association.

[2]. A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pages 248-258, New York, NY, USA, 2014. ACM.

[3]. J. Hizver, T-c Chiueh. Real-time Deep Virtual Machine Introspection and Its Applications. In Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, pages = 3-14, York, NY, USA, 2014. ACM.

[4]. P. Dovgalyuk, Pavel. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. In Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, pages 553-556, CSMR '12, Washington, DC, USA, 2012. IEEE Computer Society.

Using ABI for Virtual Machines Introspection

N.I. Fursova <natalia.fursova@ispras.ru>
P.M. Dovgalyuk <pavel.dovgaluk@ispras.ru>
I.A. Vasiliev <ivan.vasiliev@ispras.ru>
Yaroslav-the-Wise Novgorod State University,
173000, Russia, Velikiy Novgorod, Lasarevskaya street, 11

Abstract. The paper proposes an approach to introspection of virtual machines using the applications binary interface. The purpose of the method is to get information about the system, while having a minimum knowledge about its internal structure. Our system is based on QEMU emulator and has a modular structure.

Existing approaches (RTKDSM, DECAF) receive data from the operating system using the kernel structures. Those instruments have to store a large number of data profiles, because all addresses and offsets in the kernel structures vary from version to version. We offer the use of the rarely changing application binary interfaces, such as calling conventions and the numbers and parameters of system calls. The idea of the method is to intercept system functions and read parameters and return values.

Processor uses a special instruction to implement a system call. We expand QEMU with instrumentation engine, so we are able to monitor each executing instruction and to filter desired ones. In the event of a system call, we pass the control to the detector of system calls, that checks the number of occurred call and according to it decides to which plugin the job should be redirected to. In the mechanism of system calls interception, it is important not only to determine that the call occurred, but also to correctly determine its completion. That is needed to obtain the values of output parameters and return values.

To determine the end of the system call, the system also has special instructions, but we need to collate the beginning of the call to its end correctly. And to do so we are using the current context.

Thus, we have implemented monitoring of file operations and processes, and created a prototype of API functions monitor. We plan to expand the set of plugins for analysis and monitoring.

Keywords: introspection; virtual machines; dynamic analysis; system calls.

DOI: 10.15514/ISPRAS-2015-27(6)-11

For citation: Fursova N.I., Dovgalyuk P.M., Vasiliev I.A. Using ABI for Virtual Machines Introspection**.** Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 159-168 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-11

References

[1]. F. Bellard, QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, pages 41-41, Berkeley, CA, USA, 2005. USENIX Association.

[2]. A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pages 248-258, New York, NY, USA, 2014. ACM.

[3]. J. Hizver, T-c Chiueh. Real-time Deep Virtual Machine Introspection and Its Applications. In Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, pages = 3-14, York, NY, USA, 2014. ACM.

[4]. P. Dovgalyuk, Pavel. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. In Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, pages 553-556, CSMR '12, Washington, DC, USA, 2012. IEEE Computer Society.

Концепция наследования в современных языках программирования

А.В. Канатов <a.kanatov@samsung.com>
Е.А. Зуев <e.zouev@samsung.com>
Исследовательский центр Samsung,
127018, Россия, г. Москва, ул. Двинцев, дом 12, корпус 1

Аннотация. Статья содержит обзор и анализ реализаций понятия наследования в современных промышленных языках программирования. Исследуются достоинства и недостатки механизмов наследования в таких языках, как C++, Java, C# и Eiffel и других, анализируются их особенности и ограничения моделей наследования, реализованных в этих языках.

На основе проведенного анализа в статье предлагается альтернативный подход к трактовке наследования, который сочетает общность и гибкость множественного наследования и простоту практического применения для целей повторного использования кода. Суть предлагаемого подхода заключается в переносе контроля валидности полного графа наследования на этап обработки обращений к свойствам класса на основе анализа перекрытий (overriding) и контроля подобия (conformance) сигнатур свойств.

Предложенный подход может быть реализован как дополнение к какому-либо существующему языковому инструменту, так и в виде независимой реализации.

Ключевые слова: множественное наследование, переопределение (overriding) конфликт имен и версий, понятие источника свойства (origin and seed).

DOI: 10.15514/ISPRAS-2015-27(6)-12

Для цитирования: Канатов А.В., Зуев Е.А. Концепция наследования в современных языках программирования. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 169-188. DOI: 10.15514/ISPRAS-2015-27(6)-12.

1. Введение

Понятие наследования является одной из фундаментальных концепций в современных ЯП, поддерживающих парадигму объектно-ориентированного программирования.

Понятие наследования служит адекватной концептуальной моделью широкого множества различных схем отношений между сущностями реального мира. С

инженерной точки зрения, языковой механизм, обеспечивающий расширение возможностей классов вместе с заданием полиморфного поведения их свойств в производных классах, служит удобной и надежной основой повторного использования кода при разработке сложных программных систем.

Практически все современные языки программирования реализуют понятие наследования и содержат соответствующие языковые механизмы. В то же время, конкретные реализации наследования в распространенных языках либо заметно ограничены по сравнению с общей теоретической концепцией, которая просто рассматривает наследование как форму отношений между сущностями, не накладывая никаких ограничений, либо следуют определенной модели объектов времени выполнения, вводя в свою очередь решения, продиктованные реализацией (class layout, virtual method table structure).

Также необходимо отметить, что основные проблемы, которые возникают при реализации наследования,– это конфликт имен, неоднозначность версий свойства при полиморфном присваивании, а также согласование статусов видимости свойств. Отмеченные проблемы возникают в основном при множественном наследовании, поэтому во многих современных языках возможности наследования ограничивают единичным наследованием, предлагая в качестве паллиативного решения понятие интерфейса (C# [3], Java [2], Kotlin [12]) или «протокола» (Swift [11]).

2. Обзор существующих подходов к поддержке наследования.

2.1 Полная поддержка наследования: язык C++

Наиболее известной реализацией модели множественного наследования служит язык C++. Несмотря на полноту реализации всех аспектов, связанных с организацией множественного наследования, и соответствующим богатством изобразительных возможностей, этот язык подвергается серьезной и обоснованной критике в связи со сложностью реального программирования и недостаточной надежностью создаваемых программ, прямо вытекающих из намерения поддержать все мыслимые потребности прогораммистов в рамках единого языка.

Две схемы ниже иллюстрируют различные аспекты организации множественного наследования в C++. Первый пример (рис. 1) представляет условную схему обычного множественного наследования, при которой в каждом из двух подобъектах базовых классов Man и Woman присутствует, в свою очередь, подобъект базового класса Person.

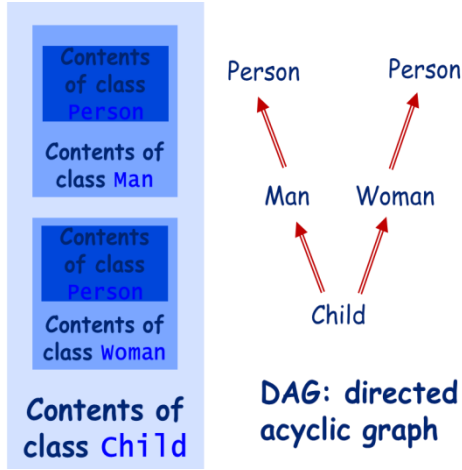


Рис. 1. Пример обычного (невиртуального) наследования в C++

Второй пример представляет модификацию схемы из предыдущего примера, когда подобъект базового класса Vehicle присутствует в объекте производного класса только в одном экземпляре, несмотря на то, что формально оба подобъекта Car и Plane (дважды) наследуют один и тот же класс Vehicle.

Такая композиция иллюстрирует понятие виртуального наследования, а соответствующая схема носит название diamond scheme (из-за ее сходства с кристаллической решеткой алмаза).

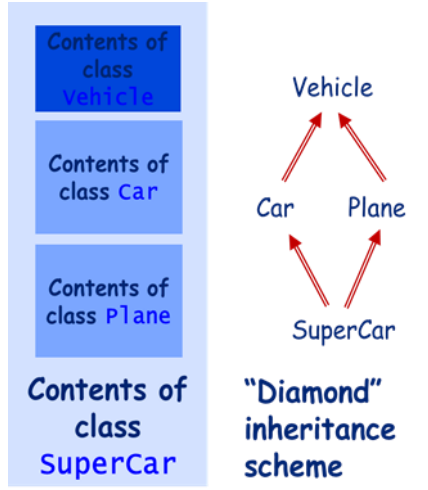


Рис. 2. Пример виртуального наследования в C++

2.2 Oberon и Zonnon

Философия языка Oberon [10] заключается в предельноем упрощении, как самого механизма наследования, так и его реализации. В языке поддерживаются лишь базовые возможности единичного наследования и полиморфизма. С одной стороны, это позволяет весьма эффективно реализовать данный механизм, что необходимо по причине ориентации языка на реализацию встроенных систем с ограниченными ресурсами. В то же время минимализм модели наследования Oberon дает возможность использовать язык для целей обучения.

Язык Zonnon [4], позиционируемый как непосредственный наследник языковой линии Pascal – Modula-2 – Oberon, переосмысливает традиционный взгляд на объектный подход в целом и на механизм наследования, в частности. Вместо привычной схемы – базовый и производный классы, возможно, реализующие некоторые интерфейсы – этот язык предлагает концептуально более чистую модель, согласно которой ключевым понятием и одновременно единственным объектом (множественного) наследования является абстрактное понятие интерфейса, а объекты выступают только как конечные «реализации» определенного интерфейса или группы интерфейсов.

Более подробно, Zonnon определяет следующие базовые единицы программы, на основе которых строится его объектно-ориентированная модель:

- Описание (definition): единый элемент абстракции. Он задает абстрактный интерфейс, а также может уточнять (refine) другое описание.
- Реализация (implementation): предоставляет реализацию некоторого описания «по умолчанию», а также может выступать как независимая коллекция ресурсов, агрегируемая в объекты.
- Объект (object): программно-управляемый ресурс, который реализует абстрактный интерфейс некоторого описания (описаний), а также может определять параллельное поведение.
- Модуль (module): специальный объект, управляемый системой (singleton object). Включает коллекцию ресурсов (данных и функциональностей), в том числе объектов и реализаций.

Рис.3 иллюстрирует модель наследования, реализованную в языке Zonnon.

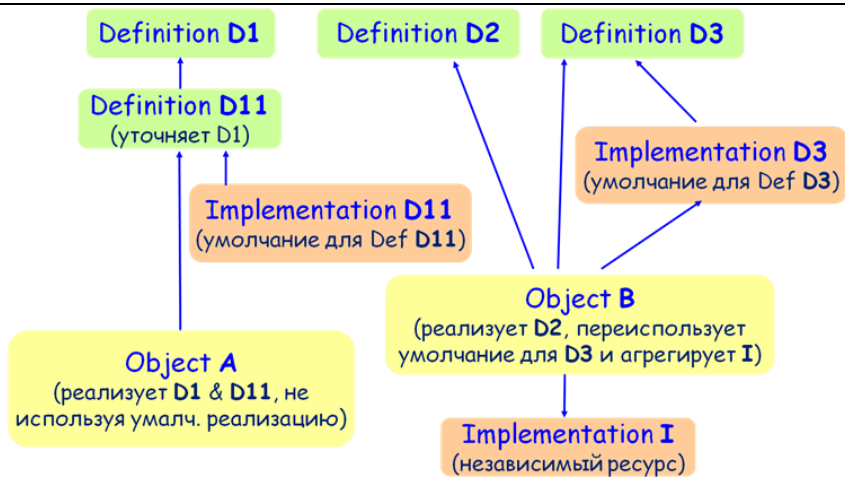


Рис. 3 Модель наследования языка Zonnon

2.3 C#, Java и Scala

Как говорилось выше, единичное наследование является в настоящее время доминирующей реализаций ООП в современных языках, предоставляя разумный компромисс между сложностью реализации и использования множественного наследования и поддержкой важнейших преимуществ наследования как базового принципа проектирования программ.

В качестве частичной компенсации за отказ от «настоящего» множественного наследования такие языки, как Java и C# предлагают в качестве альтернативы понятие интерфейса. Это понятие, будучи существенно легче для понимания и реализации, нежели сходное понятие «абстрактных классов» C++, служит в то же время и адекватной моделью многих реальных отношений.

На следующем рисунке схематически показаны отношения, характерные для таких языков.

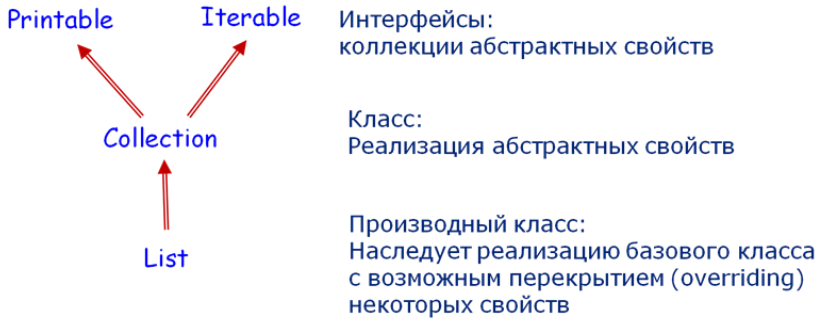


Рис. 4 Модель с единичным наследованием и множественными интерфейсами

Язык Scala [7] сохраняет в целом подход, принятый в Java (единичное наследование и интерфейсы), однако заметно расширяет возможности наследования за счет введения понятия трейта (trait), позволяющего создавать более гибкие и функционально богатые схемы отношений между объектами. Кроме того, в языке имеются возможности гибкого задания различных схем наследования (ковариантность или контравариантость), которые, однако, весьма сложны для практического использования и ориентированы больше на разработчиков прикладных библиотек, нежели на конечных пользователей языка.

2.4 Нишевые модели наследования: Ada и Eiffel

Ada95 и последующие редакции стандарта [8] добавляют объектно-ориентированные возможности к предыдущей версии языка (Ada83) при сохранении максимально возможной обратной совместимости Ада-программ.

Объектная модель Ады основана на расширении традиционного для императивных языков понятия записи (record) введением так называемых тегированных записей (tagged record), «расширяющих» некоторую другую запись. Один из атрибутов тегированной записи (‘Class) предоставляет средства, традиционные для объектного подхода, включая наследование, полиморфное присваивание и др.

В целом, ООП в этом языке, предоставляя достаточно полный спектр механизмов единичного наследования, реализован способом, не слишком привычным для разработчиков, использующих «мейнстримные» C-подобные языки, и потому практическое использование механизма наследования в этом языке представляется для них весьма затруднительным.

Для языка Eiffel характерна полная реализация полного варианта множественного наследования, которая сочетается с поддержкой настраиваемых классов (generics). Характерный для Eiffel механизм предикатов естественным образом интегрирован с механизмом наследования. Eiffel не подразумевает понятие подобъекта (характерное, например, для C++), а интерпретирует каждый метод и атрибут как отдельную сущность, к которой при наследовании могут применяться различные адаптации. Эти адаптации позволяют менять имя свойства, задавать новое тело, делать унаследованный метод абстрактным, изменять его видимость и разрешать неоднозначность при множественном наследовании.

Подобный механизм адаптации для многих программистов, знакомых с C-подобными языками, образует повышенный «барьер вхождения»; эти особенности, а также некоторые другие непривычные свойства языка вызывают определенное отторжение у современного программистского сообщества.

2.5 Нестандартные модели: JavaScript

Другая группа современных ЯП воплощает нестандартные подходы к реализации ООП и, в частности, к понятию наследования. Так, язык JavaScript [5] предлагает альтернативный подход, получивший название «прототипного наследования», при котором отношения между объектами устанавливаются динамически на основе общего для них предопределенного свойства («прототипа»).

На следующем рисунке схематически показано различие в подходах между статической и динамической моделями наследования.

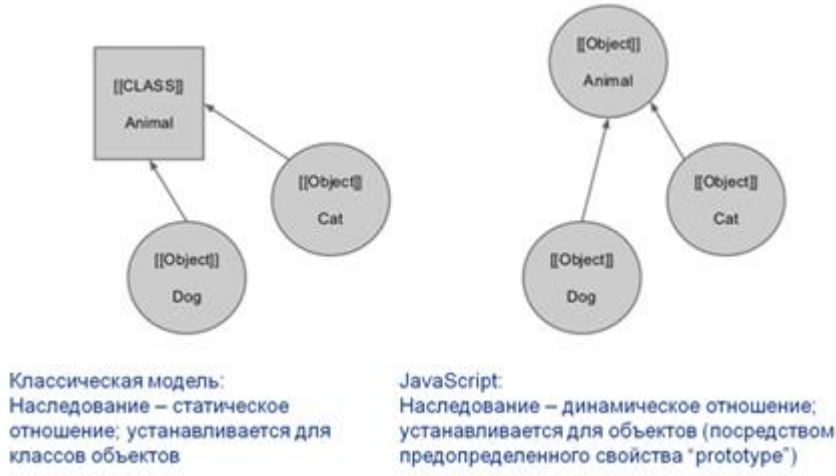


Рис.5 Различные модели наследования

Подобный подход, давая определенные преимущества в сфере построения интерактивных программ, не может считаться универсальным из-за слабых средств статического контроля, легкости внесения ошибок и трудностей в отладке.

В заключение краткого обзора следует упомянуть о языке Python [6], который не вполне вписывается в представленную выше классификацию и представляет в целом удачную попытку совмещения традиционного единичного наследования и динамической типизации, расширяя тем самым привычную парадигму «ООП = Статическая система типов».

3. Предлагаемая модель наследования

На основе анализа существующих моделей реализации наследования в современных языках программирования предлагается подход, который, как представляется авторам, преодолевает недостатки и ограничения существующих схем, сохраняя при этом механизм множественного

наследования со всеми его преимуществами. Для начала рассмотрим несколько известных определений, которые понадобятся для раскрытия сути концепции.

Класс: поименованная совокупность свойств, где свойство может быть либо атрибутом (переменным или константным) или подпрограммой (функцией, методом).

Наследование: направленное отношение между классами, при котором все свойства класса-родителя (базового) переходят к классу-наследнику (производному).

Отношение подобия (conformance): определяется наличием пути в графе наследования от одного класса до другого. Направленность графа наследования совпадает с отношением подобия.

Класс-источник (origin): это класс, в котором данное свойство было описано впервые.

Свойство-источник (seed): это та версия свойства, где это свойство было первый раз описано.

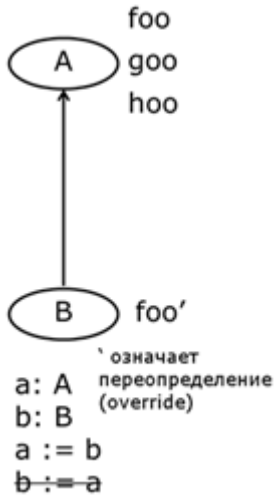


Рис. 6 Основные понятия, связанные с наследованием

На рис. 6 источником для свойства foo из класса B является свойство foo из класса A, где свойство foo было описано первый раз.

Очевидно, что для класса B класс A является родителем или непосредственным предком (базовым классом), а для класса A класс B

является наследником (производным классом). Понятия предка и потомка суть лишь транзитивные отношения от родителя и наследника соответственно.

Отношение подобия задает возможность операции присваивания для объектов типа B или его потомков объектам типа A.

Теперь рассмотрим, какие проблемы приходится решать при наличии множественного наследования в общем случае.

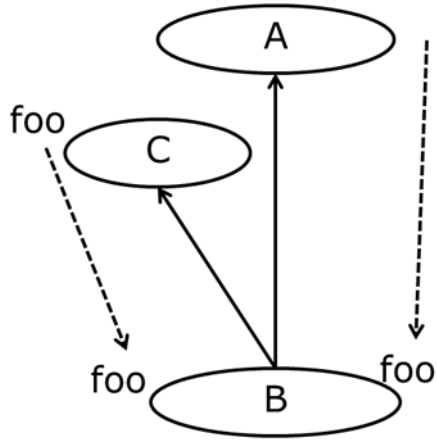


Рис. 7 Конфликт имен при множественном наследовании.

Первая задача заключается в определении того, сколько версий свойства foo будет в наследнике B, если в него наследуется более одной версии свойства с именем foo. В данном случае (см. рис. 7) у нас есть две версии foo из классов A и C, соответственно. Такая ситуация является примером конфликта имен и должна иметь определённое решение.

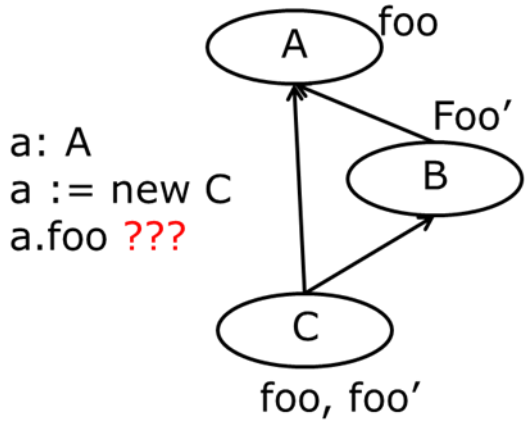


Рис. 8 Неоднозначность версий при множественном наследовании.

Для данного случая возникает неоднозначность между разными версиями свойства foo так как класс C имеет две версии – одна которая была получена непосредственно из класса A и вторая из класса B. И тогда возникает неоднозначность при полиморфном присваивании. Какая версия свойство foo должна быть вызвана при выполнении кода слева. Этот вопрос тоже должен иметь свое решение.

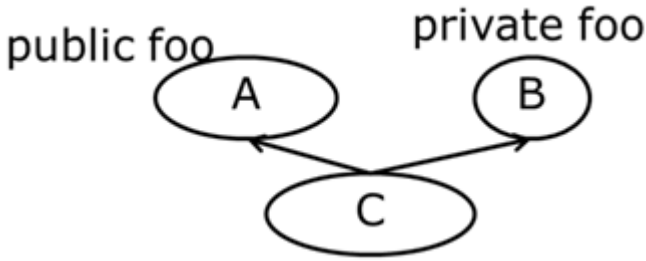


Рис. 9 Видимость свойств при множественном наследовании

Третий случай несет некоторую проблематичность. Если наследуются версии foo с разным уровнем видимости, то если они сливаются в одно свойство, необходимо решить, какая должна быть видимость у этог свойства? Эта проблема гораздо проще, чем первые две, но и она должна иметь свое решение.

Итак, начнем с конфликта имен. Рассмотрим пример

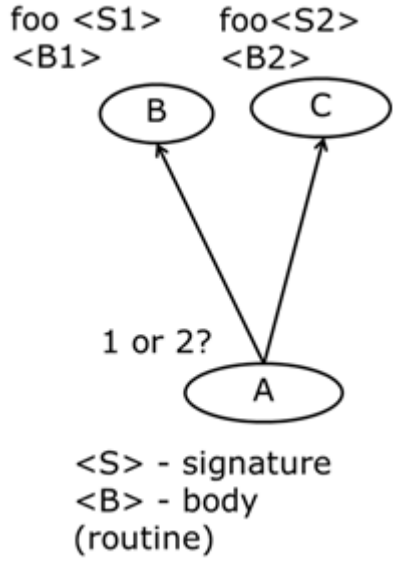


Рис. 10 Частный случай конфликта имен и его разрешение

Из классов B и C в класс A наследуются два свойства с одинаковым именем foo. Разрешение этого конфликта предлагается на основе следующего подхода. Если сигнатуры <S1> и <S2> идентичны, тела <B1> и <B2> идентичны и оба свойства имеют один и тот же источник (origin & seed), то это одно и тоже свойство и, следовательно, оно будет присутствовать в A в единственном экземпляре. Иными словами, если одно и тоже свойство приходит в класс разными путями, то можно считать, что это одно свойство (по сути, это практически то же самое, что виртуальное наследование в C++). А вот если не выполняется хотя бы одно из условий, перечисленных выше то, тогда класс B будет иметь два свойства с именем foo. Таким образом, допускается перекрытие имен (overloading).

Теперь рассмотрим короткий пример использования класса A.

```
a: A

a.foo(<parameters)>
```

Из примера видно, что в зависимости от того, как связаны между собой сигнатуры версий и их источники, обращение к свойству может быть либо однозначно разрешено компилятором, либо квалифицировано им как неоднозначность. В случае, если foo существует в A в единственной версии, то неоднозначности нет. Если же присутствует более одной версии, то в зависимости от типов параметров, которые передаются данному свойству,

компилятор пробует определить, какая версия из класса A должна быть вызвана. Если такое определение свойства дает единственный результат, то неоднозначности не возникает и именно это свойство и должно быть вызвано. В случае неоднозначности компилятор выдает сообщение об ошибке. Таким образом, мы уходим от тотальной проверки правильности всего графа наследования для всех свойств и проверяем правильность обращений к свойствам. Если обращения могут быть однозначно разрешены компилятором, то программа будет считаться корректной.

Рассмотрим еще один пример (рис. 11), когда у нас в классе наследнике есть новая версия свойства – т.е. мы производим переопределение свойств при наследовании (overriding).

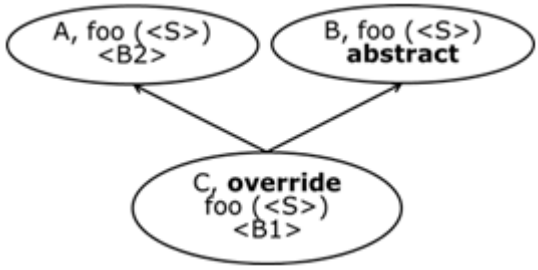


Рис. 11 Простой пример переопределения

Начнём с простого случая, когда все сигнатуры идентичны. Класс C вводит новую версию свойства foo с идентичной сигнатурой S и некоторым новым телом B1 и переопределяет обе версии foo, которые наследуются из A и B соответственно. Хотелось бы обратить внимание, на то что в классе A свойство foo имело реально тело, в то время как в классе B тело отсутствовало – foo являлось абстрактным методом. И свою очередь foo из C может быть как конкретным свойством так и абстрактным.

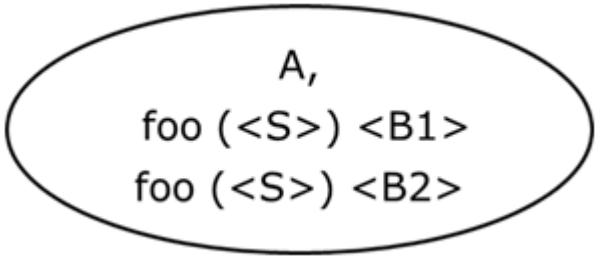


Рис. 12 Перекрытие имен свойств – недопустимая ситуация

Если еще раз обратиться к концепции перекрытия имен, то возникает вопрос, какое перекрытие является допустимым. Предлагается, что в рамках одного

класса два свойства являются различными, если у них разные имена или если у них одно имя и различные сигнатуры. Тем самым в рамках одного класса предлагается трактовать как ошибочную ситуацию, представленную на рисунке 12.

Однако, такого рода ситуация может возникнуть при наследовании и мы ее рассмотрим отдельно.

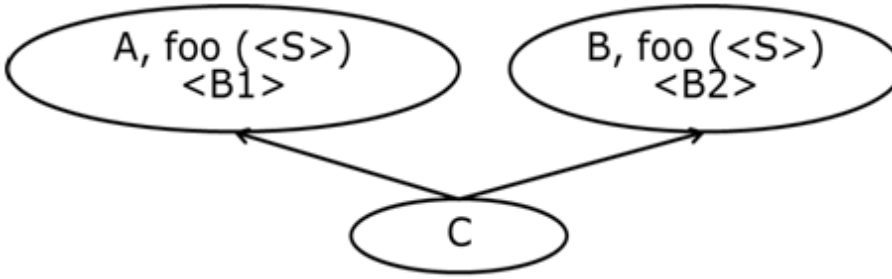


Рис. 13 Неоднозначность при множественном наследовании.

В данном случае на рис. 13 свойства foo из A и foo из B не должны иметь общего источника и тогда в классе C у нас есть две перегруженные версии с идентичными сигнатурами и разными телами. И теперь давайте рассмотрим следующие обращения. Если обращение текстуально находиться в C или его наследниках, то обращения A.foo (<exprS>) и B.foo (<exprS>) являются полностью валидными и очевидно понятными – в первом случае вызывается версия foo из класса A и из B во втором. Обращение к свойству foo просто по имени является неоднозначным и при такой попытке - foo (<exprS>) – компилятор выдает сообщение об ошибке. Аналогичная ситуация происходит, если обращение идет из внешнего кода по отношению к классу, как в примере ниже

c: C

c.foo (<exprS>)

Компилятор не может однозначно определить версию foo и выдаёт соответствующее сообщение об ошибке.

Если рассмотреть общий случай как разрешается конфликт имен при помощи перекрытия и/или переопределения, то получается следующая схема:

Пусть есть два метода (подпрограммы, функции) с именем foo и сигнатурами <S1> и <S2> и телами <B1> и <B2> соответственно, которые наследуются в некоторый класс, то в случаях если

1. <S1> = <S2> - сигнатуры совпадают
    a. <B1> = <B2> - это один и тот же метод и он присутствует в единственном экземпляре

b. <B1> != <B2> - это два разных метода. Ситуация неоднозначности и разрешается она при попытке обращение, как было описано выше.

2. <S1> != <S2> - сигнатуры не совпадают – это два разных метода и при обращении к ним компилятор либо однозначно определяет версию исходя из статических типов передаваемых параметров или выдает сообщение о неоднозначности конкретного обращения.

a. Если оба метода происходят из одного источника, то необходимо разрешить неоднозначность, выбрав ту или другую версию для поддержки полиморфных присваиваний.

3. <S1> != <S2> - сигнатуры не совпадают и присутствует переопределение с сигнатурой <S3>, которая подобна обеим сигнатурам <S1> и <S2>. Неоднозначности нет, если только одна версия для foo.

Пусть есть два атрибута с именем attr и типами T1 и T2, которые наследуются в некоторый класс, то

1. T1 = T2 - это один и тот же атрибут и он присутствует в единственном экземпляре

2. T1 != T2 - это два разных атрибута. Ситуация неоднозначности и разрешается компилятором исходя из вида использования атрибута.

a. Если оба атрибута происходят из одного источника, то необходимо разрешить неоднозначность, выбрав ту или другую версию для поддержки полиморфных присваиваний.

3. Если присутствует переопределение с типом T3, который подобен обоим типам T1 и T2, то неоднозначности нет, так как есть одна версия для attr.

Таким образом, существует решение для конфликта имен при наличии перекрытий имен и переопределении версий при наследовании. Причем надо отметить, что оно существует как для методов, так и для атрибутов и принципиально схема не различается для них.

Если обобщить схему на произвольное количество свойств, то общая схема выглядит, как показано на рис. 14.

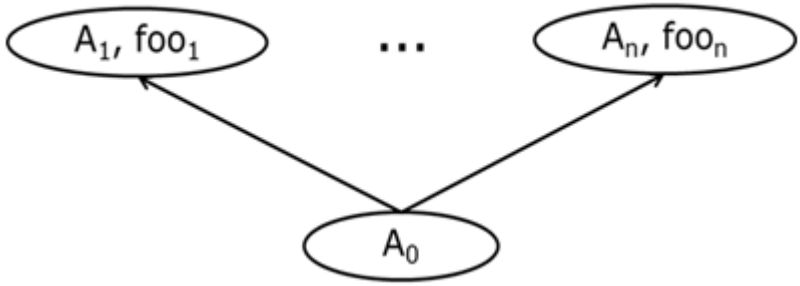


Рис. 14 Общая схема перекрытия при множественном наследовании

Есть класс A, который наследует n классов A_1 .. A_n и каждый из них содержит свойство foo. Тогда множество свойств foo в классе A имеет потенциально 4 составляющие

- Если в A0 для свойства foo из A1 применена операция абстрагирования (синтаксически она может выглядеть как A1.foo is abstract), то среди всех версий foo из классов A2 .. An должна быть версия fooi из класса Ai, которая подобна foo1. Другими словами версия foo1 становится абстрактной и заменятся версией fooi.
- Если в A0 свойство foo из A2 задается как переопределяющее (синтаксически это может выглядеть как override A2.foo), то эта версия переопределяет все версии из множества A3 .. An которым она подобна.
- Пусть свойства foo3 .. fook – это одно и тоже свойство. Тогда все они «сливаются» в одно свойство в классе A0.
- Остается множество свойств fook+1 .. foon, которое определяет различные перекрытые версии foo.

И, наконец, второй вариант: имеется переопределение в A_0.

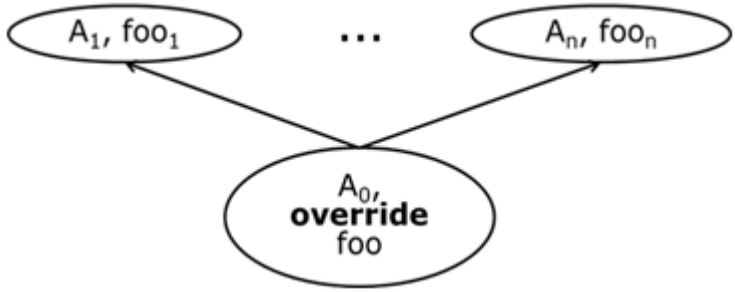


Рис. 15 Общая схема переопределения при множественном наследовании

Тогда опять часть версий может быть сделана абстрактными (как описано выше), а оставшиеся делятся на два множества (потенциально пустые).

- foo_1 .. foo_k – переопределены версией foo из A0.

- foo_{k+1} .. foo_n – наследованные и перекрытые версии foo, которые теперь являются частью A_0.

Есть еще один случай, который является ахиллесовой пятой ковариантного переопределения при наличии полиморфного присваивания – это так называемый cat calls. К сожалению, авторам неизвестен адекватный русский перевод термина, так что оставим англоязычный вариант. Суть проблемы заключается в нарушении статической типизации при следующей схеме наследования.

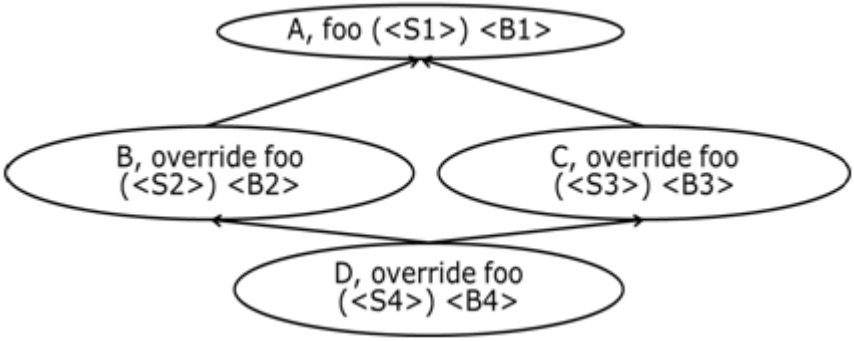


Рис. 16 Пример нарушения системы типизации при множественном наследовании

Данная схема наследования валидна, если сигнатуры <S1> и <S2> различны, и следующие сигнатуры попарно подобны: <S2>→<S1>, <S3>→<S1>, <S4>→<S2> & <S4>→<S3>, где знак → означает подобие. Рассмотрим вариант полиморфного присваивания

a: A **is new** D()

Иными словами, некоторая сущность a была описана как имеющая статический тип A, а ей при выполнении программы будет присвоен объект типа D. И тогда обращение вида

a.foo (<parameters>)

будет работать правильно если типы <parameters> подобны сигнатуре <S4>, и приводить к нарушению системы типов, если они подобны сигнатуре <S1>. Эта проблема не нова, и лобовое ее решение заключается в выполнении проверки правильности обращения в контексте всей программной системы, а

не одного класса (нечто схожее с концепцией LTO – только тут будет производиться консервативная проверка семантики при наличии информации о всей системе). Одно из возможных решений состоит в запрете полиморфного присваивания при ковариантном переопределении, но это несколько сужает возможности применения множественного наследования.

Последняя проблема, которую мы рассмотрим, заключается в возможном конфликте областей видимости при наследовании. Обратимся к рис. 17.

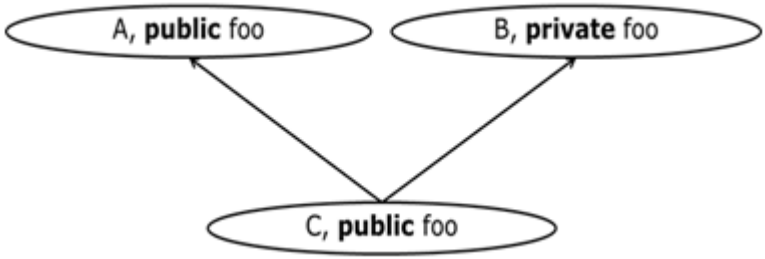


Рис. 17 Области видимости при множественном наследовании

Вне зависимости от того, каким способом (из тех что были описаны выше) мы добиваемся того, что в классе C у нас остается свойство foo в единственном экземпляре, область его видимости не может заужаться. Это можно выразить следующей схемой.

private … private => public или private

public … private = > public

4. Заключение

В статье представлены подходы к реализации концепции наследования в современных языках программирования. Предложена альтернативная концепция, которая ставит во главу угла особый вид множественного наследования с возможностью перегрузки методов и атрибутов, их переопределения при наследовании. Вводится понятие неоднозначности обращений к свойствам классов, которые либо однозначно разрешаются компилятором при проверке всех обращений, либо отвергаются им как неразрешимые.

Список литературы

[1]. International Standard: ISO/IEC 14882:2011(E) Information technology – Programming Languages – C++

[2]. J.Gosling, B.Joy, G.Steele, G.Bracha, A.Buckley, The Java Language Specification, 2015-02-13, https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

[3]. C# Language Specification, Version 5.0, Microsoft Corporation, https://www.microsoft.com/en-us/download/details.aspx?id=7029
[4]. Gutknecht J., Romanov V., Zueff E. The Zonnon Project: A .NET Language and Compiler Experiment, in V.Skala, P.Nienaltowski (Eds.) .NET Technologies 2005 Conference Proceedings, May 30 – June 1, 2005, University of West Bohemia, Plzen, Czech Republic, ISBN 80-86943-01-1.
[5]. International Standard: ISO/IEC 16262:2011(E) Information technology – Programming Languages, their environments and system software interfaces – ECMAScript language specification.
[6]. The Python Language Reference, https://docs.python.org/3.3/reference/.
[7]. Martin Odersky, Lex Spoon, and Bill Venners: Programming in Scala, Second Edition, Artima Press, 2010.
[8]. International Standard: ISO/IEC 8652:2012 Information technology – Programming Languages – Ada.
[9]. Bertrand Meyer: Object-Oriented Software Construction, Second Edition. Prentice Hall. ISBN 0-13-629155-4.
[10]. N.Wirth: The Programming Language Oberon, http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon.Report.pdf
[11]. The Swift Programming Language Reference: https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AboutTheLanguageReference.html.
[12]. The Kotlin Language Reference. http://kotlinlang.org/docs/reference/

The Concept of Inheritance in Modern Programming Languages

A. Kanatov <a.kanatov@samsung.com>
E. Zouev <e.zouev@samsung.com>
Samsung R&D Institute Russia, Dvintsev 12, 127018 Moscow, Russia.

Abstract. The paper gives a brief overview of existing approaches to inheritance implemented in some mainstream and experimental programming languages including Ada, Eiffel, C++, Java, Scala, Oberon, and Zonnon.Advantages and limitations of the approaches are analyzed and discussed.The paper claims that multiple inheritance model is one of the most important features that should be supported by any modern programming language because it reflects some fundamental relationships in the real world. However, only a few industrial languages do support the modelwherein such a support is far from being comfortable for users and leads to many troubles in development and maintenance. An alternative approach in the design of inheritance support is presented. The suggested approach keeps multiple inheritance as the general, powerfuland flexible mechanism for software design and reuse and is based on overloading and overriding with conflicts resolution at call sites based on conformance instead of full validity of the system inheritance graph. All typical problems in multiple inheritance including name clashes, name resolution,

signature conformance, covariance and contravariance are carefully consideredand discussed. The detailed explanation of how the problems are solved within the approach is presented. The paper describes a completely research project and is not supported by Samsung R&D Institute.

Keywords: multiple inheritance, overriding, name clashes, version conflict, origin and seed.

DOI: 10.15514/ISPRAS-2015-27(6)-12

For citation: Kanatov A., Zouev E. The Concept of Inheritance in Modern Programming Languages. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 169-188 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-12

References

[1]. International Standard: ISO/IEC 14882:2011(E) Information technology – Programming Languages – C++

[2]. J.Gosling, B.Joy, G.Steele, G.Bracha, A.Buckley, The Java Language Specification, 2015-02-13, https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

[3]. C# Language Specification, Version 5.0, Microsoft Corporation, https://www.microsoft.com/en-us/download/details.aspx?id=7029

[4]. Gutknecht J., Romanov V., Zueff E. The Zonnon Project: A .NET Language and Compiler Experiment, in V.Skala, P.Nienaltowski (Eds.) .NET Technologies 2005 Conference Proceedings, May 30 – June 1, 2005, University of West Bohemia, Plzen, Czech Republic, ISBN 80-86943-01-1.

[5]. International Standard: ISO/IEC 16262:2011(E) Information technology – Programming Languages, their environments and system software interfaces – ECMAScript language specification.

[6]. The Python Language Reference, https://docs.python.org/3.3/reference/.

[7]. Martin Odersky, Lex Spoon, and Bill Venners: Programming in Scala, Second Edition, Artima Press, 2010.

[8]. International Standard: ISO/IEC 8652:2012 Information technology – Programming Languages – Ada.

[9]. Bertrand Meyer: Object-Oriented Software Construction, Second Edition. Prentice Hall. ISBN 0-13-629155-4.

[10]. N.Wirth: The Programming Language Oberon, http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon.Report.pdf

[11]. The Swift Programming Language Reference: https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AboutTheLanguageReference.html.

[12]. The Kotlin Language Reference. http://kotlinlang.org/docs/reference/

Агрессивная инлайн-подстановка функций для VLIW-архитектур

А.В. Ермолицкий <era@mcst.ru>
М.И. Нейман-заде <muradnz@mcst.ru>
О.А. Четверина <chetverina_o@mcst.ru>
А.Л. Маркин <markin_a@mcst.ru>
В.Ю.Волконский <vol@mcst.ru>
АО "МЦСТ", 119991, Россия, г. Москва, Ленинский проспект, дом 51

Аннотация. Достижение высокой производительности на микропроцессорах с VLIW-архитектурой возможно лишь при использовании агрессивной инлайн-подстановки. Предложенный в настоящей работе алгоритм оптимизации явно учитывает время компиляции, что делает его эвристику более сбалансированной и позволяет значительно сократить рост кода и ускорить компиляцию по сравнению с известными алгоритмами. Кроме того, нам удалось достичь высоких показателей производительности благодаря ряду факторов: учёт в эвристике ключевых оптимизаций, использование клонирования функций, частичной инлайн-подстановки и компиляции в режиме «вся программа». Реализация нашего алгоритма в оптимизирующем компиляторе для архитектуры Эльбрус позволила ускорить задачи SPEC CPU2006 в среднем в 1.41 раз.

Ключевые слова: оптимизация, оптимизирующий компилятор, инлайн-подстановка, VLIW.

DOI: 10.15514/ISPRAS-2015-27(6)-13

Для цитирования: Ермолицкий А.В., Нейман-заде М.И., Четверина О.А., Маркин А.Л., Волконский В.Ю. Агрессивная инлайн-подстановка функций для VLIW-архитектур. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 189-198. DOI: 10.15514/ISPRAS-2015-27(6)-13.

1. Введение

Большинство современных программ написано с использованием высокоуровневых языков программирования. При этом распространённой практикой является разбиение исходного кода на множество мелких функций. Это упрощает разработку, однако препятствует эффективному исполнению программы, поскольку код различных функций не может перемешиваться на этапе компиляции. Широко известным решением указанной проблемы

является инлайн-подстановка (inline expansion), выполняемая оптимизирующим компилятором [1-5]. Суть данной оптимизации заключается в подстановке тела функции в точку её вызова. Наряду с очевидным устранением операций вызова, передачи параметров и результата, инлайн-подстановка позволяет увеличить эффективность других оптимизаций, таких как наложение итераций цикла и планирование инструкций. В частности, инструкции из разных процедур могут быть спланированы одновременно. Основным недостатком инлайн-подстановки является рост размера кода, что приводит к увеличению времени компиляции. Чрезмерный рост кода приводит к возникновению блокировок из-за более частых промахов в кэш кода. Кроме того, в некоторых случаях перемешивание холодного и горячего кода может приводить к неоптимальному планированию последнего.

Указанные проблемы особенно актуальны для VLIW-архитектур со статическим планированием, где в случае небольших функций сложно использовать возможности параллельного исполнения инструкций. Для VLIW необходима очень агрессивная инлайн-подстановка [6], которая в случае современных программ может выполнять подстановку десятков тысяч вызовов. Известные алгоритмы инлайн-подстановки приводят к слишком сильному росту кода и замедлению компиляции. Как правило, авторы подобных алгоритмов решают задачу о ранце (Knapsack problem), т.е. пытаются найти такое множество подстановок, при котором время исполнения программы минимально, а коэффициент увеличения размера кода не превышает заданной величины. Такая постановка задачи на практике нередко приводит к избыточному росту кода, либо алгоритм останавливается до достижения существенного прироста производительности. Частичная подстановка функций [7-12] решает указанные проблемы лишь отчасти, в общем случае её область применимости сильно ограничена. Кроме того, эти алгоритмы показывают хорошие результаты лишь при наличии профильной информации и компиляции в режиме "вся программа" (whole program), на практике такие режимы компиляции используются исключительно редко.

2. Сбалансированная инлайн-подстановка

В настоящей работе представлен алгоритм агрессивной инлайн-подстановки, в котором за счёт тщательного подбора эвристик удалось достичь оптимального баланса между уменьшением времени исполнения *Te* и увеличением времени компиляции *Tc*. Эвристика алгоритма основана на минимизации функционала: $Te^3Tc \rightarrow min$ (1). За счёт учёта *Tc*, предложенный алгоритм обладает хорошей масштабируемостью, позволяя оптимизировать в режиме whole огромные программы, содержащие миллионы строк исходного кода. В теории такая эвристика не ограничивает рост времени компиляции, однако на практике она позволяет значительно уменьшить в среднем *Tc* по сравнению с известными алгоритмами агрессивной инлайн-подстановки.

В общем случае задача минимизации функционала (1) является NP-полной [13]. Мы используем жадный алгоритм для приближённого решения этой задачи, суть его заключается в следующем. Формируется множество всех операций вызова в программе, для которых возможно выполнение инлайн-подстановки (*call_set*). Для каждого вызова из *call_set* вычисляется его вес: $W = -3\Delta Te/Te - \Delta Tc/Tc$, где ΔTe и ΔTc - изменение времени исполнения и компиляции программы соответственно, вызванное данной инлайн-подстановкой. Подстановка считается полезной только при $W > 0$. Инлайн-подстановки выполняются в порядке уменьшения веса до тех пор, пока есть хоть один вызов с $W > 0$. При этом в процессе работы оптимизации для подставленных вызовов вычисляется вес и они добавляются в множество *call_set*. Кроме того, учитывается, что подстановка одной функции может изменить веса других функций - поддерживается динамическое перевычисление весов и переупорядочивание множества вызовов.

Основной сложностью при реализации эвристики является оценка величин *Te* и *Tc*. Для оценки *Te* каждая функция разбивается на регионы - циклы и линейные участки. Для каждого региона оценивается время его исполнения в зависимости от количества инструкций в нём, и затем эти времена суммируются. При этом используется профильная информация: статическая (на основе оценки компилятором вероятностей переходов) либо динамическая (полученная в результате тренировочного запуска программы). Для корректной оценки *Te* и ΔTe необходимо, чтобы профильная информация была согласованной, т.е. сумма счётчиков вызовов функции должна равняться счётчику стартового узла этой функции. На практике статический профиль часто бывает несогласованным, это происходит в первую очередь из-за наличия рекурсии и вызовов по указателю. Поэтому перед оценкой *Te* выполняется локальное согласование профиля за счёт умножения счётчиков функции на корректирующий множитель. Время компиляции оценивается по формуле: $Tc = a + bN^2$, где a и b - константы, подобранные эмпирически, а N - количество инструкций в функции. Подобная оценка *Tc* является очень грубой, поскольку реальное время компиляции зависит от множества факторов, однако для наших целей такое приближение оказалось достаточным.

Ключевой особенностью алгоритма является учёт различных оптимизаций, которые становятся возможными в результате инлайн-подстановки, делается это следующим образом. При оценке веса вызова $f \rightarrow g$ вычисляется изменение времени исполнения: $\Delta Te = (Te(f^*) + Te(g^*)) - (Te(f) + Te(g))$, где $Te(f^*)$ и $Te(g^*)$ - оценка времени исполнения функций f и g соответственно после инлайн-подстановки функции g в f. Оптимизации учитываются при вычислении $Te(f^*)$. Например, если функция g вызывается с константными параметрами, учитывается, что после подстановки g результат некоторых выражений можно будет вычислить статически, некоторые ветвления управления можно будет удалить, соответственно, количество подставленных

в f инструкций будет меньше, чем изначально было в g. Таким образом, при оценке $Te(f^*)$ мы учитываем только те инструкции, которые останутся после инлайн-подстановки и выполнения оптимизаций. Аналогичным образом учитываются оптимизации при оценке $Tc(f^*)$ и ΔTc. Мы учитываем следующие оптимизации:

- планирование кода (code scheduling)
- пропагация значений (global copy propagation)
- пропагация констант (constant propagation)
- удаление избыточных ветвлений управления
- удаление мёртвого кода (dead code elimination)
- наложение итераций цикла (softpipe)
- прочие цикловые оптимизации [14,15].

Кроме того, вес дополнительно увеличивается в случае, когда в результате подстановки уточняется количество итераций цикла (если оно задаётся параметром функции) либо удаётся порвать больше зависимостей между указателями - параметрами подставляемой функции.

3. Экспериментальные результаты

Предложенный алгоритм был внедрён в промышленный оптимизирующий компилятор для архитектуры Эльбрус [16,17]. Для эмпирического подбора коэффициентов в функциях оценки *Te* и *Tc* использовались задачи из пакета SPEC CPU2000 [18]. Коэффициенты подбирались таким образом, чтобы минимизировать среднеквадратичное отклонение оценочных и реальных величин *Te* и *Tc*. Для проверки эффективности предложенного алгоритма использовались 29 задач из пакета SPEC CPU2006 [18]. Замеры времени исполнения задач проводились на четырёхъядерном микропроцессоре Эльбрус 4С с тактовой частотой 800 МГц. В помодульном режиме сборки без динамической профильной информации (опции -O3 -ffast) за счёт инлайн-подстановки время исполнения в среднем уменьшилось на 29%, при этом время компиляции увеличилось всего на 13%, а размер бинарного файла увеличился на 9.1% (см. таблицу 1). В режиме "вся программа" с использованием профиля (опции -O3 –ffast –fwhole –fprofile-use) результаты получились ещё более впечатляющими: исполнение ускорилось на 41%, компиляция замедлилась на 12%, размер бинарного файла увеличился на 7.7% в среднем (см. таблицу 2). Наибольшее ускорение было достигнуто на задаче 447.dealII, которая ускорилась в 7.3 раза при незначительном замедлении компиляции (0.9%) и уменьшении размера кода (1.5%).

Таблица 1: результаты для помодульного режима компиляции

название теста	коэффициент ускорения теста	замедление компиляции	рост размера бинарного кода
400.perlbench	1,00	7,9%	7,3%
401.bzip2	1,00	0%	4,0%
403.gcc	1,03	11,4%	8,3%
410.bwaves	1,00	3,9%	0%
416.gamess	1,01	5,5%	3,1%
429.mcf	1,09	6,0%	19,2%
433.milc	1,00	21,4%	11,3%
434.zeusmp	4,39	17,3%	-1,4%
435.gromacs	1,07	14,5%	13,9%
436.cactusADM	1,00	7,3%	5,2%
437.leslie3d	1,03	1,4%	0,2%
444.namd	2,43	44,4%	25,0%
445.gobmk	1,02	10,9%	4,8%
447.dealII	4,32	46,6%	34,0%
450.soplex	1,41	10,0%	4,7%
453.povray	1,70	31,6%	30,6%
454.calculix	1,00	0%	1,5%
456.hmmer	1,00	7,0%	11,5%
458.sjeng	1,04	10,5%	11,3%
459.GemsFDTD	1,00	0%	0%
462.libquantum	1,03	14,5%	10,4%
464.h264ref	1,01	2,7%	4,8%
465.tonto	1,01	3,1%	5,4%
470.lbm	1,00	2,3%	5,7%
471.omnetpp	1,56	15,5%	10,7%
473.astar	1,43	20,2%	10,3%
481.wrf	1,52	2,3%	2,6%
482.sphinx3	1,00	9,1%	8,9%
483.xalancbmk	3,27	11,1%	20,1%
gmean	**1,29**	**13,0%**	**9,1%**

Таблица 2: результаты для режима компиляции «вся программа»

название теста	коэффициент ускорения теста	замедление компиляции	рост размера бинарного кода
400.perlbench	1,03	13,9%	12,9%
401.bzip2	1,00	8,0%	5,7%
403.gcc	1,10	21,2%	22,2%
410.bwaves	1,00	0%	0%
416.gamess	1,02	0%	0%
429.mcf	1,28	0%	-1,7%
433.milc	1,88	41,8%	23,9%
434.zeusmp	4,38	9,4%	-7,3%
435.gromacs	1,06	8,2%	7,8%
436.cactusADM	1,00	2,0%	2,8%
437.leslie3d	1,11	0%	1,4%
444.namd	2,43	50,7%	48,3%
445.gobmk	1,16	10,8%	7,3%
447.dealII	7,30	0,9%	-1,5%
450.soplex	1,52	20,0%	6,3%
453.povray	2,32	23,3%	23,1%
454.calculix	1,01	10,3%	8,9%
456.hmmer	1,01	7,2%	3,0%
458.sjeng	1,10	16,3%	14,8%
459.GemsFDTD	1,00	1,9%	0,5%
462.libquantum	1,03	3,9%	7,7%
464.h264ref	1,01	3,5%	2,7%
465.tonto	1,03	2,3%	2,3%
470.lbm	1,00	-2,1%	0,7%
471.omnetpp	1,81	23,7%	21,6%
473.astar	1,74	23,3%	7,1%
481.wrf	1,62	1,8%	0,8%
482.sphinx3	1,06	9,3%	10,2%
483.xalancbmk	3,27	-1,8%	7,7%
gmean	**1,41**	**11,8%**	**7,7%**

Список литературы

[1]. Robert W. Scheifler. An analysis of inline substitution for a structured programming language. Communications of the ACM, Sept. 1977, Volume 20, Issue 9, p. 647-654
[2]. P. C. Chang, Wen-mei W. Hwu. Inline function expansion for compiling C programs. ACM SIGPLAN Notices, 1989, vol. 24, no. 7, p. 246-257
[3]. Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu. Profile-guided automatic inline expansion for C programs. Software - Practice & Experience, May 1992, Volume 22, Issue 5, p. 349 - 369
[4]. Matthew Arnold, Stephen J. Fink, Vivek Sarkar, Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. ACM SIGPLAN Notices, 2000, vol. 35, no. 7, p. 52-64
[5]. Peng Zhao, Jose Nelson Amaral. To inline or not to inline? Enhanced inlining decisions. Workshop on Languages and Compilers for Parallel Computing (LCPC) , Oct. 2003, p. 405-419
[6]. Andrew Ayers, Richard Schooler, Robert Gottlieb. Aggressive inlining. ACM SIGPLAN Notices, 1997, vol. 32, no. 5, p. 134-145
[7]. Robert Muth, Saumya Debray. Partial Inlining. Technical report, Department of Computer Science, University of Arizona, 1997
[8]. Tom Way, Ben Breech, Lori Pollock. Region Formation Analysis with Demand-Driven Inlining for Region-Based Optimization. Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques, 2000
[9]. Tom Way, Lori L. Pollock. A Region-based Partial Inlining Algorithm for an ILP Optimizing Compiler. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2002, Volume 2, p. 552-556
[10]. Dibyendu Das. Function inlining versus function cloning. ACM SIGPLAN Notices, June 2003, Volume 38, Issue 6, p. 23 - 29
[11]. Peng Zhao, Jose Nelson Amaral. Function outlining and partial inlining. Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing, 2005, p. 101 - 108
[12]. Jun-Pyo Lee, Jae-Jin Kim, Soo-Mook Moon, Suhyun Kim. Aggressive Function Splitting for Partial Inlining. Proceedings of the 2011 15th Workshop on Interaction between Compilers and Computer Architectures, 2011, p. 80-86
[13]. M.R. Garey, D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York.1979. 338 p.
[14]. Steven S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers, San Francisco. 1997. 888 p.
[15]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Pearson Education, Boston. 2006. 1000 p.
[16]. Краткое описание архитектуры Эльбрус, http://www.elbrus.ru/arhitektura_elbrus
[17]. В.Ю. Волконский, А.В.Брегер, А.Ю.Бучнев, А.В.Грабежной, А.В.Ермолицкий, Л.Е.Муханов, М.И.Нейман-заде, П.А.Степанов, О.А.Четверина. Методы распараллеливания программ в оптимизирующем компиляторе. Вопросы радиоэлектроники, серия ЭВТ, выпуск 3, 2012, стр.63-88
[18]. Standard Performance Evaluation Corporation. SPEC CPU Benchmarks, http://www.spec.org/

Aggressive Inlining for VLIW

A.Ermolitckii <era@mcst.ru>
M.Neiman-Zade <muradnz@mcst.ru>
O.Chetverina <chetverina_o@mcst.ru>
A.Markin <markin_a@mcst.ru>
V.Volkonskii <vol@mcst.ru>
MCST, 51 Leninskii avenue, Moscow, 119991, Russian Federation

Abstract. Inline expansion is very important for high performance VLIW, especially for microprocessors with static scheduling. Optimizations in optimizing compilers for VLIW duplicate code aggressively and lead to long compile time. Our inlining algorithm is based on heuristics that takes into account compile time explicitly. This made optimization more balanced and significantly reduced code growth and compile time compared to common inlining approach based on minimization of runtime within constraints. Instead of using hard constraints we are trading run time for compilation time in some proportion. Our heuristics predicts several key optimizations in evaluation of runtime and compile time: code scheduling, global copy propagation, dead code elimination and different loop optimizations. Optimizations prediction reduces the need in profile information which is rarely available in practice. Our implementation of inlining includes cloning, partial inlining and inlining across compilation modules in whole program mode. All this factors make dramatic impact on performance: our inlining implementation in the Elbrus optimizing compiler boost SPEC CPU2006 benchmark performance by factor of 1.41 at the cost of 12% increase of compile time and 7.7% increase of code size on average.

Keywords: optimization, optimizing compiler, inline expansion, VLIW.

DOI: 10.15514/ISPRAS-2015-27(6)-13

For citation: Ermolitckii A., Neiman-Zade M., Chetverina O., Markin A., Volkonskii V. Aggressive Inlining for VLIW. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 189-198 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-13

References

[1]. Robert W. Scheifler. An analysis of inline substitution for a structured programming language. Communications of the ACM, Sept. 1977, Volume 20, Issue 9, p. 647-654
[2]. P. C. Chang, Wen-mei W. Hwu. Inline function expansion for compiling C programs. ACM SIGPLAN Notices, 1989, vol. 24, no. 7, p. 246-257
[3]. Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu. Profile-guided automatic inline expansion for C programs. Software - Practice & Experience, May 1992, Volume 22, Issue 5, p. 349 - 369
[4]. Matthew Arnold, Stephen J. Fink, Vivek Sarkar, Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. ACM SIGPLAN Notices, 2000, vol. 35, no. 7, p. 52-64

[5]. Peng Zhao, Jose Nelson Amaral. To inline or not to inline? Enhanced inlining decisions. Workshop on Languages and Compilers for Parallel Computing (LCPC) , Oct. 2003, p. 405-419
[6]. Andrew Ayers, Richard Schooler, Robert Gottlieb. Aggressive inlining. ACM SIGPLAN Notices, 1997, vol. 32, no. 5, p. 134-145
[7]. Robert Muth, Saumya Debray. Partial Inlining. Technical report, Department of Computer Science, University of Arizona, 1997
[8]. Tom Way, Ben Breech, Lori Pollock. Region Formation Analysis with Demand-Driven Inlining for Region-Based Optimization. Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques, 2000
[9]. Tom Way, Lori L. Pollock. A Region-based Partial Inlining Algorithm for an ILP Optimizing Compiler. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2002, Volume 2, p. 552-556
[10]. Dibyendu Das. Function inlining versus function cloning. ACM SIGPLAN Notices, June 2003, Volume 38, Issue 6, p. 23 - 29
[11]. Peng Zhao, Jose Nelson Amaral. Function outlining and partial inlining. Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing, 2005, p. 101 - 108
[12]. Jun-Pyo Lee, Jae-Jin Kim, Soo-Mook Moon, Suhyun Kim. Aggressive Function Splitting for Partial Inlining. Proceedings of the 2011 15th Workshop on Interaction between Compilers and Computer Architectures, 2011, p. 80-86
[13]. M.R. Garey, D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York.1979. 338 p.
[14]. Steven S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers, San Francisco. 1997. 888 p.
[15]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Pearson Education, Boston. 2006. 1000 p.
[16]. Kratkoe opisanie arkhitektury Elbrus [Short description of Elbrus architecture], http://www.elbrus.ru/arhitektura_elbrus (in Russian)
[17]. V.Volkonskii, A.Breger, A.Buchnev, A.Grabezhnoi, A.Ermolitckii, L.Mukhanov, M.Neiman-Zade, P.Stepanov, O.Chetverina. Metody rasparallelivaniia program v optimiziruiushchem kompiliatore [Program parallelization methods in optimizing compiler]. Voprosy radioelektroniki, seriia EVT, vypusk 3 [Questions of radioelectronics, Computer Technology series, Volume 3], 2012, p.63-88 (In Russian)
[18]. Standard Performance Evaluation Corporation. SPEC CPU Benchmarks, http://www.spec.org/

Comparative Analysis of Frameworks for the Performance Evaluation of Multi-tier Cloud Applications[1]

[1]G.R. Garay <godofredo.garay@reduc.edu.cu>
[2]A. Tchernykh <chernykh@cicese.mx>
[3]A.Yu. Drozdov < alexander.y.drozdov@gmail.com>
[1]University of Camaguey,
Carretera de Circunvalacíon, km 5, 74650 Camagüey, Cuba
[2]CICESE Research Center, Carretera Ensenada-Tijuana No. 3918, Zona Playitas, Código Postal 22860, Apdo. Postal 360, Ensenada, B.C. México
[3] Moscow Institute of Physics and Technology (State University)
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia

Abstract. In early stages of a hardware design, when a lot of options need to be consideredquickly, analytic modeling is used. It allows the performance evaluation of proposed systems without requiring complexand costly detailed simulations. Analytical approaches for the performance evaluation of cloud computing environments include Queuing Theory and Control Theory models. Real-Time Calculus (RTC) is a high-level analysistechnique previously proposed for stream-processing hard real-time systems and frequently used toevaluate trade-offs in packet stream processing architectures.The central idea of theModular Performance Analysis with RTC (MPA-RTC) is to build an abstract performance model of a system that bundles all information needed for performance analysis with RTC. In this paper, we address the performance evaluation of multi-tier clouds applications, and compare a Real-Time Calculus-based framework with two classical analytical approaches such as queuing theoretic approaches and control theoretic approaches. We focus on the capabilities of these alternatives for estimating the key Quality of Service parameter - the application response-time. In addition, we discuss the capabilities of each analytical approach for modeling other aspects of cloud computingenvironment such as workload models, task processing models, virtual machine (VM) provisioning, VMs performance interference, autonomic resource management, server consolidation, and cloud scaling strategies (horizontal and/or vertical). The capabilities of MPA-RTCas a valuable tool for the performance evaluation of cloud computing platforms are exposed.

Keywords: Real-Time Calculus, queuing, control, QoS, response-time, cloud computing

DOI: 10.15514/ISPRAS-2015-27(6)-14

[1] The work is partially supported by the Ministry of Education and Science of Russian Federation under contract No02.G25.31.0061 12/02/2013 (Government Regulation No 218 from 09/04/2010).

For citation: Garay G.R., Tchernykh A., Drozdov A.Yu. Comparative Analysis of Frameworks for the Performance Evaluation of Multi-tier Cloud Applications. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 6, 2015, pp. 199-224 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-14

1. Introduction

Virtualization-based resource management in cloud computing environments is usually related to performance improvement, including QoS guaranteeing, energy saving, and others parameters specified in the SLAs.

A number of researchers have focused on SLA (Service Level Agreement)-based objectives (e.g., client-perceived response time, throughput, dependability, reliability, availability, costs, security, confidentiality, etc.).

In order to optimize the system performance, some methods have to be exploited to estimate the possible metrics based on the input of the system. To this end, analytical performance models can be established for the examined applications running upon the virtualized environment.

After the objectives and proper performance estimation approaches are determined (e.g., analytical frameworks), performance analysis need to figure out the best configuration for the placement of virtual machines [3].

In a previous work [32], we discussed a Real-Time Calculus-based approach for the performance evaluation of multi-tier cloud applications, where we only focused on the capabilities of RTC for estimating the Quality of Service parameters such as response time.

In this considerably extended version of the paper, we compare the previously proposed analytical framework with two classical analytical approaches commonly used for the performance evaluation of multi-tier cloud Web applications (see [3-5]) such as queuing theoretic approaches and control theoretic approaches. In particular, we focus on the capabilities of these alternatives that can be employed for estimating Web application response-time. In addition, specific VMs management issues are also analyzed.

The paper is organized as follow. In Section 2, we present the motivation of the work, and give some background information. Existing analytical approaches are presented in Section 3, and the main features of Real-Time Calculus are presented in Section 4. A discussion of the principal findings is presented in Section 5. The paper is concluded in Section 6.

2. Motivation

As a motivation example (Fig. 1), let us consider a system under test (SUT) consisting a three-tier web application [4, 6]. The three-tiers include presentation-tier, application (business)-tier and data-tier, implemented in actual systems as a web server process (P), application server process (B), and database server process (D), respectively (Fig. 2).

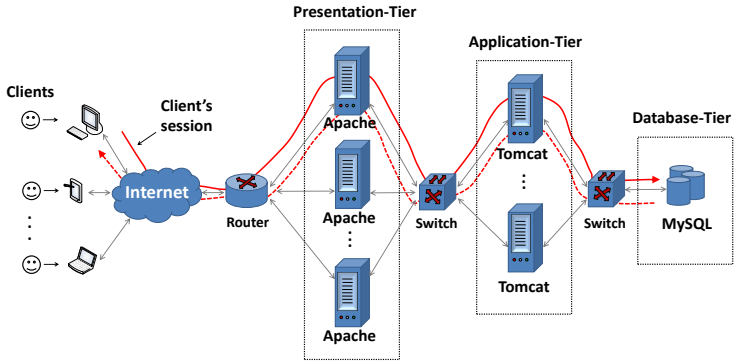


Fig. 1. Imaginary example of a client session on a basic multi-tier application architecture (note that in virtualized cloud platforms, each software server, i.e., Apache, Tomcat, and MySQL, is run inside of a virtual machine).

The first tier named presentation-tier consists of Web server. It displays what is presented to the user on the client side within their Web browsers. For the Web server-tier, it mainly has three functions: (1) Admitting/denying requests from the clients and services Web requests; (2) Passing requests to the application server; and finally, (3) receiving response from application server and sending it back to clients. In this paper, all these tiers will be modeled as software servers.

In our SUT (Fig. 1), a state-full web application is considered. For this reason, the session-based data-access client requests and responses are processed by the same virtual machines (VMs) instances (see Fig. 2).

In practice, multiple deployment scenarios of VMs on physical machines (PMs) may exist. In this paper, we want to answer the following question: can we predict whether the application's response time will violate (or surpass) a pre-specified deadline when application's characteristics at each single tier in isolation are known in advance with certain levels of confidence?

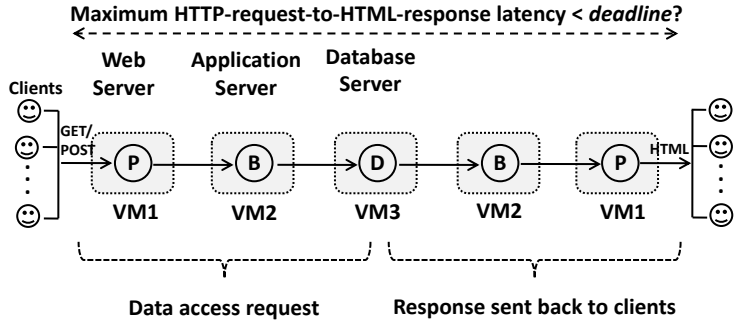


Fig. 2. Focus of attention: Predicting Web-application response-time in cloud computing platform, e.g., does maximum request-to-response latency of a client request will not exceed application deadline (with 95% confidence interval)

3. Existing Approaches

3.1 Queuing models

One of the most popular analytical approaches for the performance evaluation of cloud computing environments [4, 5] is Queuing Theory (QT) [7]. Here, we present a short introduction to QT [8], which summarizes the most important issues of this analytical approach.

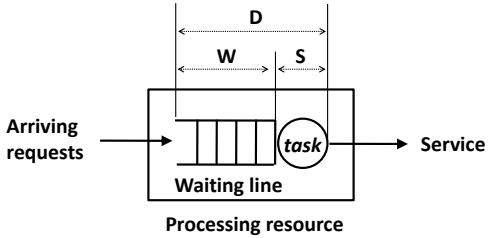


Fig 3. Single queue parameters in the context of the classical QT: mean waiting time (W), mean service time (S), mean request-response delay time (D).

QT can be seen as a branch of probability theory applied to different fields, e.g., communication networks, computer systems, and so forth. QT tries to estimate parameters like e.g., the mean system response time (waiting time in the queue plus service times), distribution of the number of customers in the queue, distribution of the number of customers in the system, and so forth. This analysis is mainly studied in stochastic scenarios (Fig. 3).

Queuing systems may not only be different in distributions of the inter-arrival and service times, but also in the number of servers, size of the waiting queues (infinite or finite), service discipline, and so on.

To analyze multi-tier web applications, one can represent web applications as a network of queuing systems. One basic classification of queuing networks is the distinction between open and closed queuing networks.

In an open network, new customers may arrive from outside of the system (coming from a conceptually infinite population) and, later on, leave the system. In a closed queuing network, the number of customers is fixed, and no customers enter or leave the system. Examples of queuing models that could be used to capture and analyze the behavior of cloud systems and their applications are M/M/1, M/G/1, M/M/m, M/G/m/K, M/M/c, or Erlang formulas (Fig. 4; see [4, 5] for references).

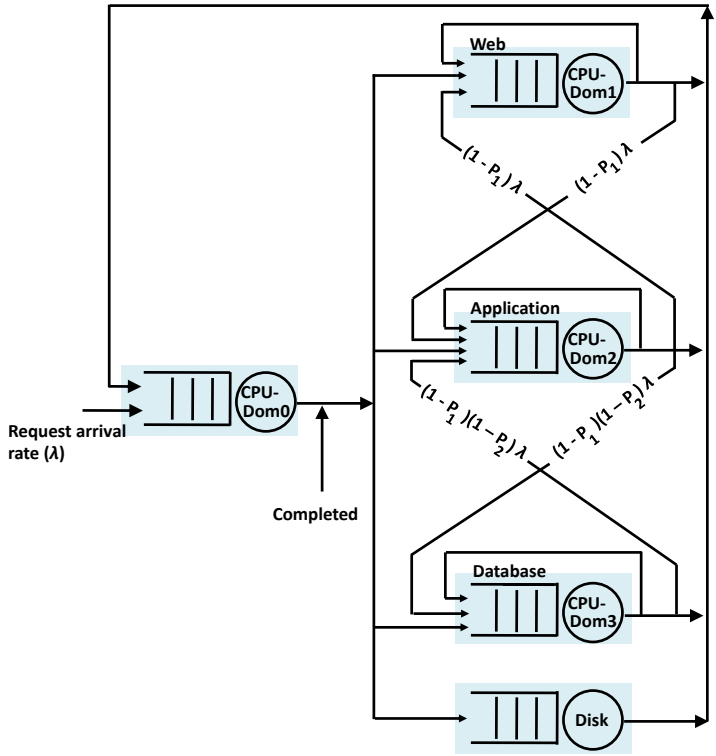


Fig 4. Example of a closed-queueing system based on M/G/1queuing modeling for virtualized three-tier applications, as shown in Fig. 1 (Adapted from [2]).

3.2 Control theory models

Control theory (CT) is another popular technique [4, 5]. It provides a systematic approach for designing closed-loop systems that are one of the basic type of control system, which uses feedback signals to control itself. They are designed to automatically achieve and maintain the desired output condition by comparing it with the actual condition. Such systems are designed to be stable by trying to avoid wild oscillations, accurate by achieving the desired outputs (e.g., response time), and settle quickly to steady state values (e.g., to adjust the workload dynamics) [9] (Fig. 5).

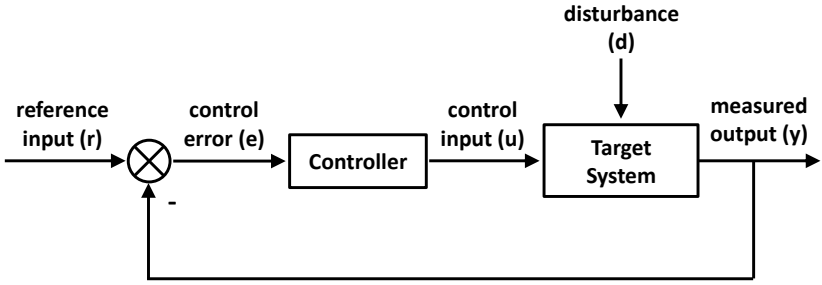


Fig 5. Standard feedback control loop (Adapted from [1]).

The target system provides a set of performance variables referred to as measured outputs or simply outputs.

Sensors monitor the outputs of the target system, and actuators can adjust control inputs, or simply inputs, to change the system behavior.

The feedback controller is the decision-making unit of the control system. The main objective of the controller is to maintain the outputs of the system sufficiently close to the desired values by adjusting the inputs under disturbances. This desired value is translated by the control system to the set point signals, which gives the option for the control system designer to specify the goals or values of the outputs that have to be maintained at runtime.

The feedback control system is a reactive decision making mechanism, because it waits until a disturbance affects the outputs of the system to make the necessary decisions.

Another type of control systems is feed-forward control system (considered as a proactive control mechanism).

Also, it is used a combination of the two previous types, i.e., feedback and feed-forward control system (which addresses the limitations of both schemes) [10].

Recently, CT has been used in the analysis of many aspects of cloud computing environments [4, 5] (Fig. 6).

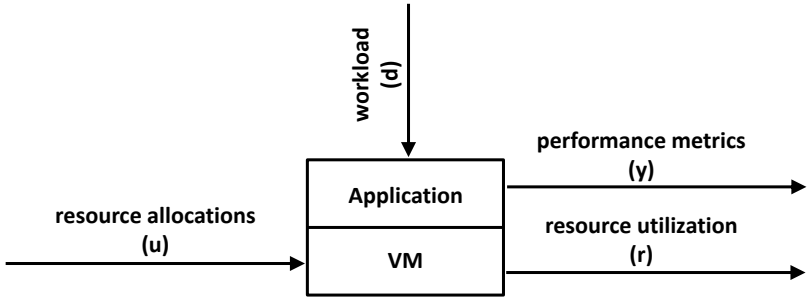


Fig 6. Example of the application of control theory to automated resource and service level management in shared virtualized infrastructures with three nodes hosting multiple multi-tier applications (Adapted from [1]).

4. Modular Performance Analysis with RTC

In addition to the analytical approaches described in the previous section, in this paper, we analyze the features provided by RTC.

The central idea of "Modular Performance Analysis with RTC" (MPA-RTC) [11] is to build an abstract performance model of a system that bundles all information needed for performance analysis with RTC.

The abstract performance model unifies essential information about the environment, about the available computation and communication resources, about the application tasks (or dedicated HW/SW components), as well as about the system architecture itself.

For performance analysis by using MPA-RTC, a real system (e.g., a multi-tier web application) can be decomposed into abstract performance analysis components (i.e., RTC components) whose behavior can be deterministic or non-deterministic. For instance, Fig. 2 shows that the system can be decomposed into five concatenated queuing subsystems, which can be analytically modeled as RTC components with non-deterministic behavior.

4.1 Deterministic analysis

RTC is a formal method developed in embedded systems domain [12-14]. In [15], RTC is compared with the analytical approaches commonly used for the performance evaluation of network interfaces. A case study of the applicability of RTC in the context of performance evaluation of network interfaces is presented in [16].

Basically, the RTC framework primary consists of a task model, resource model, and calculus (i.e., Real-Time Calculus) that allows reasoning about event streams and their processing.

In this work, we consider the problem of the evaluation of cloud computing environments. In the mentioned framework, the input event stream might be composed by a finite number of different event types, e.g., HTTP requests issued by clients, service requests issued the web server to the application server, or service requests issued the application server to the database server.

On the other hand, the processing resources that we model are the virtual machines in which the application tiers are deployed, and the task model, considered in this work, consists of software servers.

In RTC, the resource model captures the information about the available processing capacity of different hardwares involved in the processing of requests, and the possible mappings of processing functions to these resources (e.g., mapping application tiers to virtual machines).

The analytical framework also considers characteristics of the event stream entering the system (e.g., clients requests in Fig. 2), which are specified by using their arrival curves.

Thus, given the infrastructure of a data center, the calculus associated with the RTC-based framework can be used to analytically determine properties such as the maximum delay (latency) experienced by an event stream, and take into consideration the underlying scheduling disciplines at the different processing resources.

In this paper, we estimate the impact of the data center resource pool parameters (e.g., servers speed), and stochastic behavior of both web applications workload and application tiers processing time on the application response time by analytical methods.

Other specific VMs management issues are also analyzed and discussed (Section 5).

In RTC, the basic model is characterized by a processing resource that receives incoming requests and executes them using the available resource (processing or communication) capacity. To this end, some non-decreasing functions of resource provisioning are introduced.

Definition 1 (Arrival and Service Function). An event stream can be described by an arrival function R, where R(t) denotes the number of events that have arrived in the interval [0, t).

A computing or communication resource can be described by a service function C, where C(t) denotes the number of events that could have been served in the interval [0, t).

Definition 2 (Arrival and Service Curves). The upper and lower arrival curves, $\alpha^u(\Delta),\ \alpha^l(\Delta)\ \in \mathbb{R}\geq 0$ of an arrival function R(t) satisfy the following inequality:

$$\alpha^l(t-s) \leq R(t) - R(s) \leq \alpha^u(t-s), \forall\, s,t : 0 \leq s \leq t$$

The upper and lower service curves,

$$\beta^u(\Delta),\ \beta^l(\Delta) \in \mathbb{R}^{\geq 0}$$

of a service function C(t) satisfy

$$\beta^l(t-s) \leq C(t) - C(s) \leq \beta^u(t-s)\ \forall\, s,t : 0 \leq s \leq t$$

As described in [12], α_f^u and β_r^l bounding-functions can be defined using a piecewise linear approximation (Fig. 7).

For example, given a trace representing the processing capabilities of a VM running an application tier, two-slopes piecewise linear functions (i.e., LR functions, Section 4.2) can be used for describing a lower bound of the processing service at VMs over any time interval of length Δ (Fig. 7a).

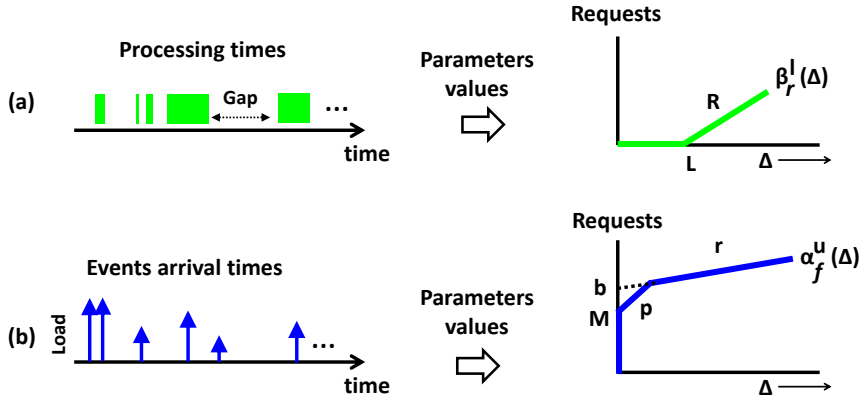


Fig 7. Obtaining the parameters values required for constructing the straight line segments of the upper and lower bounding-curves by using a software server trace and an arrival trace, respectively. In (a), the slope L represents the latency (i.e., longest gap in the trace), and the slope R can be interpreted as the average (long-term) processing rate. In (b), M represents the maximum possible load (measured e.g. in time units) on a resource for processing one token (i.e., one request); the slope p of the middle segment can be interpreted as the (load on a resource due to short-term) peak/burst rate, the slope r as the (load on a resource due to the) long-term request arrival rate, and the value b, as the burst tolerance of events stream.

Similarly, arrival curves defined by using piecewise linear segments with three pieces (three slopes) can be used for expressing an upper bound of the number of events that may arrive over any time interval of length Δ. This allows us to model an arrival curve in the form of a T-SPEC specification (p, r, M, b). For instance, a token bucket is used to specify event streams (i.e., traffic), which is widely used in the area of communication networks [17] (Fig. 7b).

Then, by using the RTC-based analytical framework, we can compute the maximum delay experienced by an event stream passing through a single resource processing the flow (e.g., a single application tier), and passing through a multiple processing resources (e.g., the entire application tiers).

When α_f^l and α_f^u describe the arrival curves of an event stream f, and if, β_r^l and β_r^u, describe the processing capability of r in terms of the same units, then, the maximum delay suffered by the event stream f at the resource r can be given by the following inequality:

$$\text{delay} \leq \sup_{t \geq 0}\{\inf\{\tau \geq 0 : \alpha_f^u(t) \leq \beta_r^l(t + \tau)\}\}$$

A physical interpretation of this inequality can be given as follows: the maximum delay experienced by an event stream (e.g., client data access requests in multi-tier cloud web applications) waiting to be served by r (e.g., a web, application, or database server) can be bounded by the maximum horizontal distance between the bounding-functions α_f^u and β_r^l (Fig. 8).

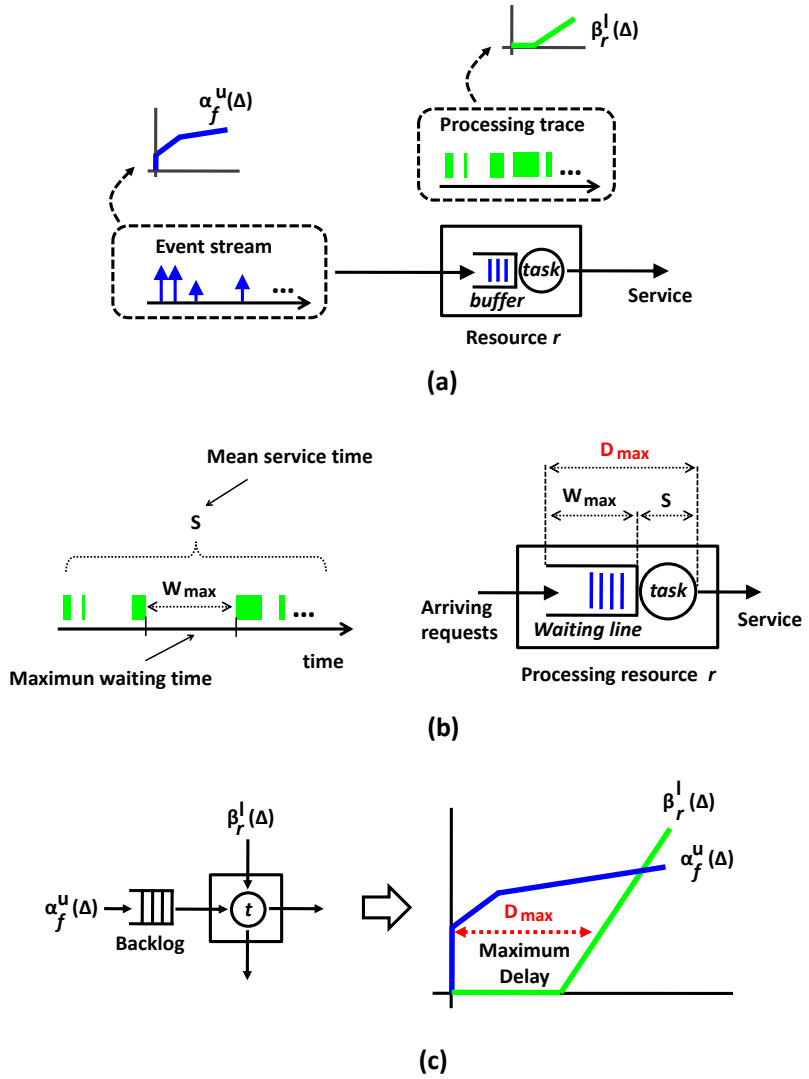


Fig 8. (a) Deriving the α_f^u and β_r^l bounding-functions of the processing resource r. (b) RTC model parameters and our metric of interest (D_{max}). (c) Modeling the resource r and obtaining its maximum request-response delay time (D_{max}) by using RTC.

According to [12], if the event stream passes through multiple resources, such as a tandem of software servers involved in processing incoming event stream using a FIFO discipline (Fig. 2), which have their input lower service curves equal to β_1^l, β_2^l, β_3^l, ..., β_n^l, then, an accumulated lower service curve β^l for serving this event

stream can be computed through an iterated convolution (as defined in the network calculus domain [18] (Fig. 9):

$$\beta^l = (((\beta_1^l \otimes \beta_2^l) \otimes \beta_3^l) \otimes \dots) \otimes \beta_n^l \qquad (1)$$

Thus, the maximum delay experienced by this stream can be given by

$$delay \leq sup_{t \geq 0}\{\inf\{\tau \geq 0 : \alpha_f^u(t) \leq \beta^l(t + \tau)\}\}$$

In the analytical framework, depending on the context, in which these bounding-functions are used, the delay can be computed in terms of different time units, e.g., cycles, seconds, etc.

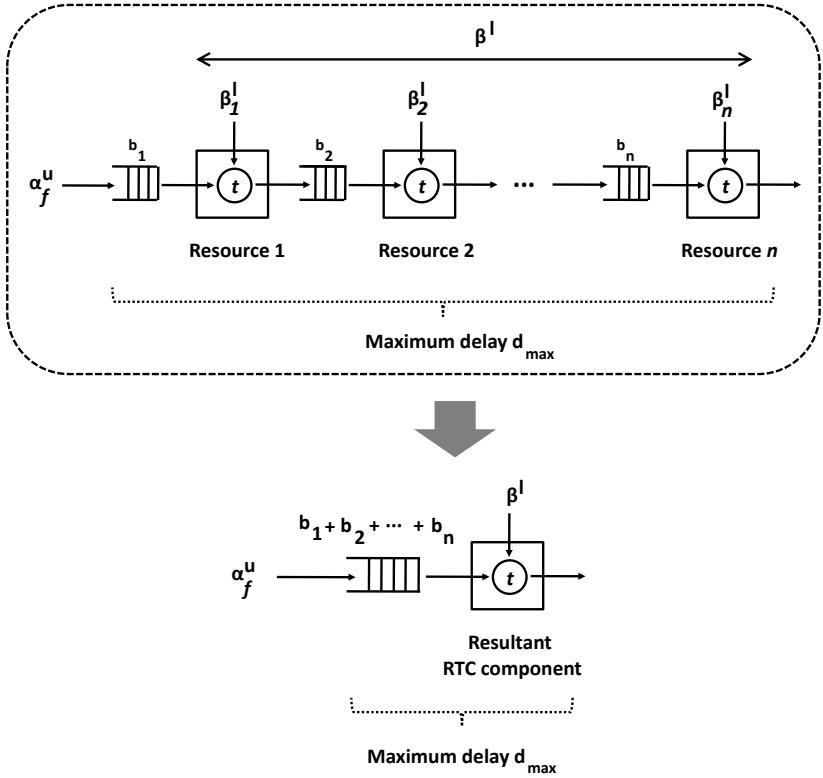


Fig 9. A tandem of processing resources (as in Fig. 2) modeled each one of them by means of an RTC component (upper part), and representation of a resultant RTC component for the system (bottom), which uses as input the β^l accumulated lower service curve computed for the tandem of processing resources using the equation (1).

In general RTC-based analysis, components are specified as transformers of input arrival and service curves into output arrival and service curves through a set of equations (Fig. 10; see [11]). Thus, RTC-based analytical approaches are compositional in the sense that they use local parameters about processing resources

(such as the arrival rate of event stream, long-term average service rate, longest gap in a trace of processing availability), which can be determined without taking into account any interference with other resources.

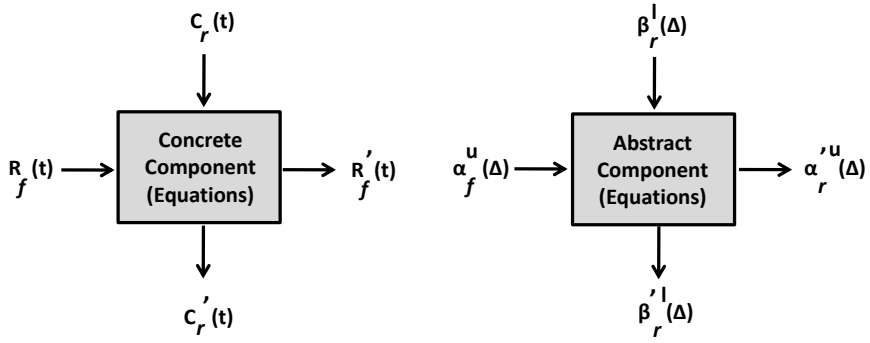


Fig 10. Transforming input functions into output functions. (a) Specific arrival and service functions, $R_f(t)$ and $C_r(t)$, enter into a concrete processing resource and are transformed into the $R'_f(\Delta)$ and $C'_r(\Delta)$ output-functions. (b) Abstract arrival and service curves, α_f^u and β_r^l, enter into an abstract component (RTC component), and are transformed into the $\alpha_f'^u$ and $\beta_r'^l$ output service curves.

Hence, by using this local information, we can predict how global parameters (such as end-to-end latency) will behave in a given system that combines the analytical models (RTC components) of these individual processing resources. This approach shows how to reduce the complexity of the system by combining the analysis of single components.

4.2 Stochastic analysis

The analytical framework described in the previous sections allows us to obtain hard real-time guarantees on delays and backlog. To this end, a finite trace of an event stream and a sliding window approach are applied to derive the arrival and service curves [14].

Contrary to the classical MPA-RTC, the RTC-based probabilistic analysis presented in [16] provides soft real-time guarantees, i.e., guarantees on delays and backlogs that are valid up to a certain level of confidence, as opposed to the hard guarantees commonly derived by formal methods.

In [16], the α_f^u and β_r^l bounding-curves are not deduced by sliding a window of length Δ over the trace and recording the minimum and maximum number of events lying within the window. Stochastic models for the service and arrival curves are considered. These models are stochastic in the sense that they consider uncertainties in the estimation of the parameters required for constructing the pieces of line for α_f^u and β_r^l.

This approach is most suitable in the context of our work (Fig. 2). For example, processing tasks at presentation, application and data layers could be modeled as latency-rate servers (LR servers). In such a case, the β_r^l lower service curve can be represented as a $\beta_{L,R}(t)$ latency-rate function (LR function). In the network calculus domain, it is defined as [18]:

$$\beta_{L,R}(t) = \begin{cases} R(t-L), & if\ t > L \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

for some $L \geq 0$ ("latency") and $R \geq 0$ ("rate").

4.3 RTC model calibration

In general, an RTC model for multi-tier cloud web applications can be calibrated (parameterized) using different alternatives. For example, the value of the input parameters of analytical model, which are needed for constructing the pieces of line of the arrival and service curves (mathematical functions), can be obtained from direct measurement on real systems [19], simulation results [20] e.g., by using trace/model-based simulations, or by synthetic models [21].

It should be noted that deriving the parameters for constructing the $\beta_{r_i}^l$ lower service curve of a concrete system component with non-deterministic behavior (e.g., a web, application or database server) from simulations or real traces may give the case where the following assumption holds (see [16]).

$$\exists\, i, \Delta :\ \beta_{r_i}^l(\Delta) < \beta_{\{r_i, reality\}}^l(\Delta) \quad (3)$$

where $i \in (1, 2, 3, \ldots)$, and $\beta_{r_i}^l$ is a resultant lower service curve derived from a set of lower service curves.

The elements of this set are a family of service curves of the component obtained by using alternatives for model calibration described above. Notice that the value of the L and R are parameters of an aggregated (resultant) bounding-curve.

Let us say that $\beta_{r_i}^l$, can be computed using aggregation functions like "AVERAGE", "MINIMUM", or "MAXIMUM", given a list of parameter values (Fig. 11; see [16] for details).

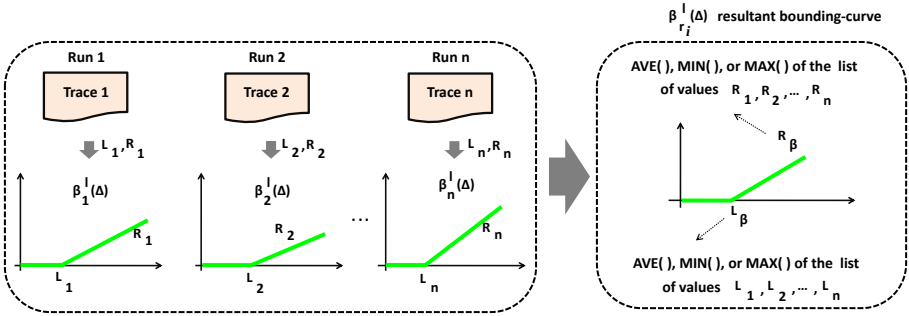


Fig 11. Family of service curves corresponding to a system component with non-deterministic behavior (left part), and procedure for obtaining its resultant bounding-curve (right part).

Lastly, $\beta^{l}_{\{r_i,\text{reality}\}}(\Delta)$ in (3) is an unknown lower bounding-curve of the SUT for the stochastic component being considered.

Indeed, note that as (3) may occasionally hold, the analytically computed results are invalid. For this reason, in [16], statistical methods are used in order to demonstrate that the values of the L and R parameters of $\beta^{l}_{r_i}$ have an adequate level of predictability, and, hence, results are valid up to certain level of confidence.

5. Discussion

In this work, we are interested in the capabilities of each analytical approach for modeling the following aspects of cloud computing: multi-tier cloud web applications, response time guarantees (hard and/or soft), workload models, task processing models, VM provisioning, VMs performance interference, autonomic resource management, server consolidation, and cloud scaling strategies (horizontal and/or vertical).

Table 1 summarizes all these issues. Moreover, to support our comparison, references to analytical studies based on queuing theory (QT) and control theory (CT) are given.

Multi-tier cloud Web application. Several authors have addressed the issue of modeling multi-tier cloud Web application by analytical approach such as QT and CT with varying degree of success (see the review in [4]).

Table 1. Comparison of analytical approaches

Modeling capabilities	MPA-RTC	Queuing Theory (QT)	Control Theory (CT)
Multi-tier cloud Web application	Yes	Yes	Yes
Hard/Soft response time guarantees	Both	No	Soft guarantees

Workload models	Real and/or synthetic	Synthetic	Real or synthetic
Task processing models	Real and/or synthetic	Synthetic	Real or synthetic
VM provisioning	Yes	Yes	Yes
VMs performance interference effect	Yes	Yes	Yes
Autonomic resource management	Yes	Yes	Yes
Server consolidation	Yes	Yes	Yes
Horizontal/Vertical scaling	Both	Both	Both

Based on the ideas exposed in Section 4, we consider that MPA-RTC is also a suitable approach for modeling multi-tier cloud Web applications. Nevertheless, it should be noted that there are differences in the scope of each approach.

RTC belongs to the class of so-called deterministic queuing theories. It is deterministic in the sense that hard upper and lower bounds of the performance metrics (such as latency) can be always found.

This distinguishes it from the class of non-deterministic analysis techniques such as QT and CT for which this guarantee cannot be provided (in general).

Deterministic queuing theories such as MPA-RTC are well-suited for studying hard performance bounds since they ensure that all requirements are met by the system during all the time.

In contrast, RTC does not allow us to model the average response time of web applications. For this purpose, stochastic approaches such as QT are better suited.

Specifically, the RTC-based probabilistic analysis described in Section 4.2 might be useful for obtaining soft real-time guarantees in the context of cloud computing environments.

Response time guarantees. In principle, RTC models allow performance analysts to derive hard and soft response time guarantees in the context of cloud computing systems.

In particular, the end-to-end latency quantity in RTC allows us to evaluate worst case scenario, i.e., the maximum delay experienced by an event stream at a given individual software server (or at a tandem of them).

On the contrary to RTC, the mean delay quantity used in QT-based analysis does not allow to obtain QoS guarantees such as response time.

Regarding CT, this methodology provides only soft performance guarantees. It is to be noted that due to inherent sources of instability in control systems (e.g., latency to get the stationary values of observable variables after applying a control action) under unpredictable disturbances, the deadlines of some tasks could be violated; hence, hard real-time guarantees cannot be obtained at all.

Nevertheless, we consider that an RTC-based stochastic analysis (Section 4.2) would be more suitable from the perspective of performance evaluation of cloud computing environments due to the dynamic nature of incoming requests and server-side processing (Fig. 2). Below we consider our workload and task processing models.

Workload models. The workload model can be analytically evaluated by using any of the following four alternatives:

(1) Real workload traces (data gathered from a production platform);
(2) Naive synthetic workload models that use probability distributions to generate workload data (based on little or no knowledge of real trace characteristics);
(3) Realistic synthetic workload models in which the model and its parameters have been abstracted through careful analysis of real workloads data from production servers;
(4) Combinations of the previous alternatives (in particular, MPA-RTC allows this approach).

Both real and realistic synthetic workloads have been considered in studies based on CT (see [5]). On the other hand, most of QT-based studies use synthetic workload models based on Poisson process [5].

In [22], the authors show that one can reasonably accept that this assumption is valid.

With respect to RTC, it supports a flexible workload model. For example, workload can be expressed by any type of service units per unit time arriving at processing resources (e.g., instructions/s, requests/s, transactions/s, etc.). It has a highly flexible workload granularity level. Besides, we can construct arrival curves from realistic event arrival traces or synthetic traffic models (constant, bursty, Poisson, etc). Also, different workload sizes (fixed or variable) can be modeled.

Task processing models. In [5], a variety of experimental platforms for modeling the processing of tasks in CT-based studies (e.g., real testbeds, simulators) are reviewed.

On the contrary, most QT-based studies only consider synthetic task processing models (e.g., processing times which follow exponential distribution [23]).

Using MPA-RTC, software servers can be modeled by means of RTC components (LR servers). To calibrate these components in isolation, the processing characteristics of software servers in terms of computational work performed by them (e.g., measured in requests/second) can be used.

VM provisioning. The process of provisioning VMs in IaaS clouds includes partial delays caused by queuing, provisioning decision, VMs instantiation and deployment.

In MPA-RTC, these delays can be modeled as non-processing intervals (variable latency periods) in a server trace in terms of processing availability (Fig. 12).

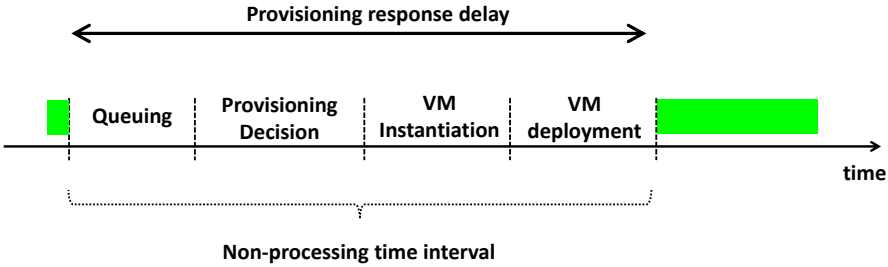


Fig 12. Modeling provisioning response delay: Non-processing intervals in a trace of execution time of software servers.

VM provisioning has been modeled analytically either by using QT [24] or CT [25].

VMs performance interference effect. In a virtualized system, performance interference is caused by sharing physical resources (mainly, I/O [26]) among VMs and virtual machine monitor scheduling (Fig. 13).

VM performance interference has been analytically modeled by using QT [27] and CT [28]. To model the performance degradation due to resource contention by using MPA-RTC, an extra logical performance component (i.e., a non-deterministic RTC component) can be added to the RTC model of the SUT.

Particularly, the service curve of this RTC component would allow us to model the non-deterministic access to shared resources in virtualized environments.

For performance analysis, this abstract component should be properly calibrated in order to achieve realistic results (Section 4.3).

Autonomic resource management. We consider that RTC can be a proper alternative for this purpose. Instead of an offline trace-based calibration approach, online methods could be employed.

To this end, all the desired parameters values of analytical model could be collected through physical infrastructure monitoring [19]. Then, collected data could be incorporated into an RTC-based autonomic control loop, aiming at achieving business objectives.

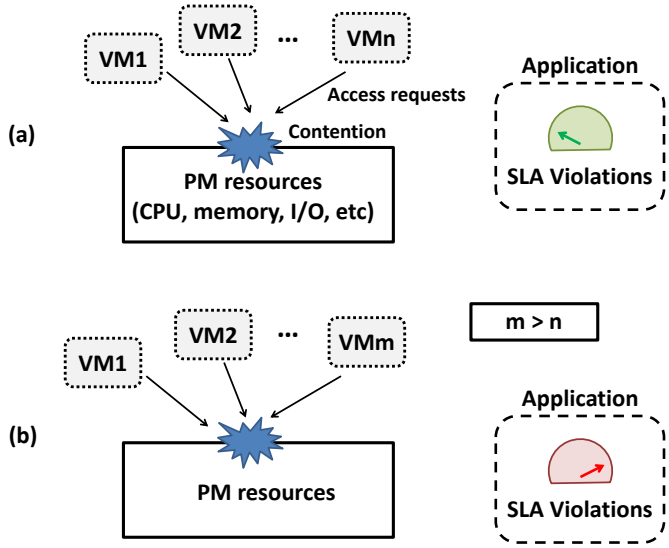


Fig 13. Imaginary examples of VMs performance interference effect due to resource contention on virtualized environments: (a) For a total of n virtual machines deployed, application performance in terms of SLA violations is acceptable. (b) For m > n, performance degrades ostensibly.

This way, cloud systems could dynamically adapt themselves to the changing environment, and, based on management strategies, control actions (e.g., live VMs migration) could be triggered (Fig. 14).

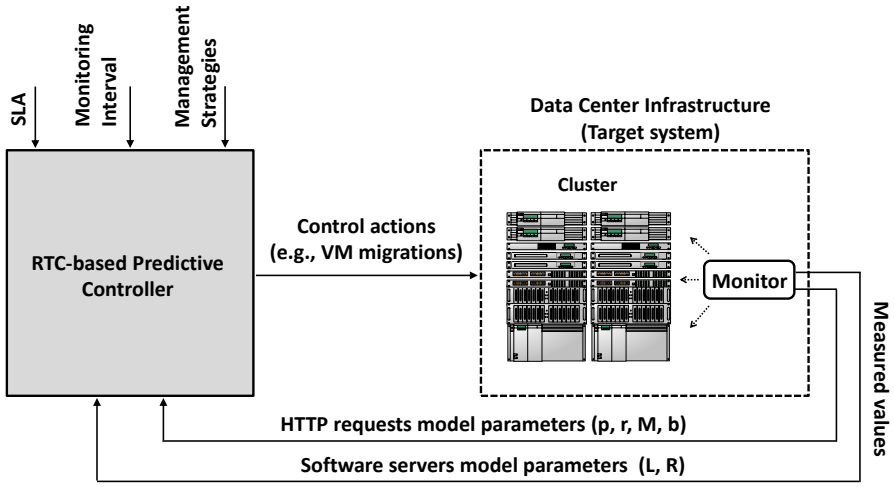


Fig 14. Cloud monitoring through online instrumentation for RTC-based autonomic resource management.

Various typical papers covering autonomic resource management by using QT and CT are surveyed in [3].

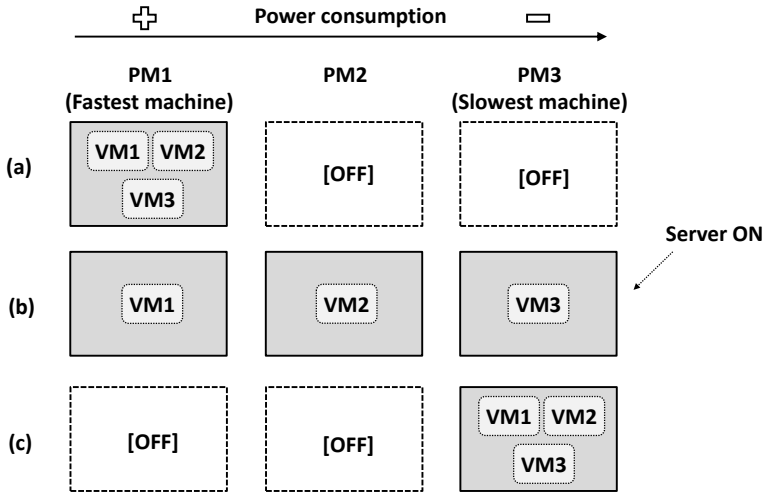


Fig 15. Imaginary examples of VMs deployment scenarios for our SUT (Fig. 2): (a) Speed-oriented server consolidation. (b) Non-consolidated scenario. (c) Energy-efficient server consolidation.

Server consolidation. The consolidation of servers is an energy-aware resource allocation technique for cloud computing systems.

In real scenarios, IaaS providers need to evaluate many VM combinations to find the optimal consolidation of VMs on the physical servers taking into account QoS (Fig. 15). We consider that the RTC-based interference model as well as autonomic resource management issues described above could be precisely incorporated into VM consolidation performance analysis.

In [29], CT is used to deal with the problem of achieving the best consolidation level that can be attained without violating application SLAs.

In [30], server consolidation is analyzed by using QT.

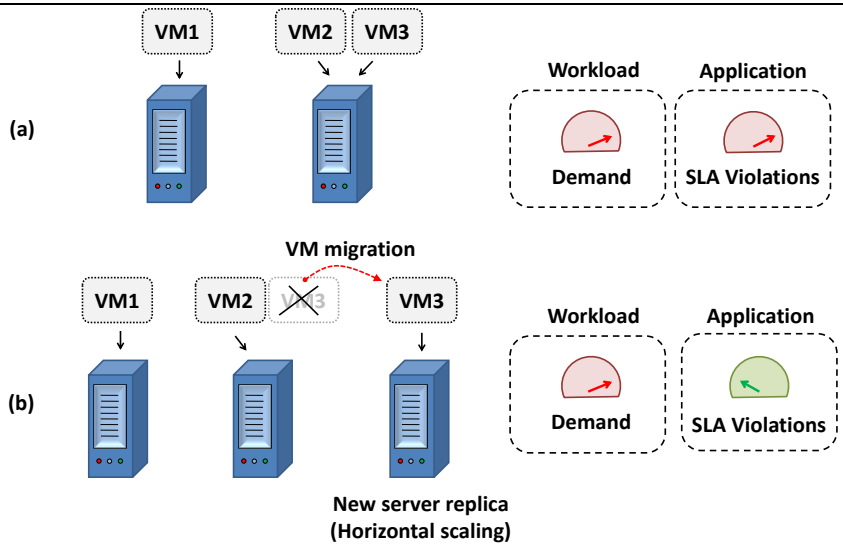


Fig 16. Effect of horizontal scaling on application performance for our SUT (Fig. 2): (a) In this imaginary example, for the baseline VMs deployment scenario considered, a high workload demand leads to a high number of SLA violations. (b) For the same baseline scenario, after adding a new server replica for migration purposes, the number of SLA violations is low.

Horizontal/Vertical scaling. Approaches to scaling cloud infrastructure to meet client workload requirements can be classified as vertical scaling type, e.g., adding larger and more powerful physical machines to accommodate the demand, and horizontal scaling type, e.g., adding new server replicas (i.e., PMs) and load balancers to distribute load among all available replicas (Fig. 16).

We would expect that using a higher speed server (vertical scaling) or adding a new server replica for VMs migration purposes (horizontal scaling) have to be reflected in the shape of the service curves (LR function) characterizing the task processing of the software servers deployed on the VMs being migrated.

For this reason, we consider that MPA-RTC allows us to model both vertical and horizontal scaling strategies. In [5], various examples are reviewed in which vertical scaling strategies are evaluated using QT. Several examples of application of control theory for the performance evaluation of both vertical and horizontal scaling can be found in [5]. In [31], horizontal scaling by using QT is evaluated.

6. Conclusion

In this paper, we discuss different approaches for modeling cloud-based systems. Based on the results of their comparison, we conclude that RTC is suitable framework for estimating statistical response time guarantees, which is an important

quality attribute for Web applications from the user point of view. In addition, other contemporary issues in cloud computing research could be analyzed by using MPA-RTC.

References

[1]. X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin, "What does control theory bring to systems research?," ACM SIGOPS Operating Systems Review, vol. 43, pp. 62-69, 2009.

[2]. K. RahimiZadeh, M. AnaLoui, P. Kabiri, and B. Javadi, "Performance modeling and analysis of virtualized multi-tier applications under dynamic workloads," Journal of Network and Computer Applications, 2015.

[3]. X.-Y. Wang, L.-H. Fan, X.-H. Jia, and W.-T. Huang, "A Survey of Virtualization-based Resource Management in Cloud Computing Environments," Journal of Convergence Information Technology, vol. 8, 2013.

[4]. D. Huang, B. He, and C. Miao, "A survey of resource management in multi-tier web applications," IEEE Communications Surveys & Tutorials, vol. 16, pp. 1574--1590, 2014.

[5]. T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," Journal of Grid Computing, vol. 12, pp. 559-592, 2014.

[6]. N. Grozev and R. Buyya, "Performance modelling and simulation of three-tier applications in cloud and multi-cloud environments," The Computer Journal, vol. 58, pp. 1-22, 2015.

[7]. L. Kleinrock, Theory, Volume 1, Queueing Systems: Wiley-Interscience, 1975.

[8]. . Willig, "A short introduction to queueing theory," Technical University Berlin, Telecommunication Networks Group, vol. 21, 1999.

[9]. T. Abdelzaher, Y. Diao, J. L. Hellerstein, C. Lu, and X. Zhu, "Introduction to control theory and its application to computing systems," in Performance Modeling and Engineering: Springer, 2008, pp. 185-215.

[10]. T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A systematic survey on the design of self-adaptive software systems using control engineering approaches," in Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2012, pp. 33-42.

[11]. E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse, "System architecture evaluation using modular performance analysis: a case study," Int. J. Softw. Tools Technol. Transf., vol. 8, pp. 649-667, 2006.

[12]. S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister, "Performance evaluation of network processor architectures: combining simulation with analytical estimation," Comput. Netw., vol. 41, pp. 641-665, 2003.

[13]. L. Thiele, S. Chakraborty, M. Gries, and S. Künzli, "A framework for evaluating design tradeoffs in packet processing architectures," in Proceedings of the 39th conference on Design automation New Orleans, Louisiana, USA: ACM, 2002.

[14]. S. Chakraborty, S. Künzli, and L. Thiele, "A General Framework for Analysing System Properties in Platform-Based Embedded System Designs," in Proceedings of the conference on Design, Automation and Test in Europe - Volume 1: IEEE Computer Society, 2003.

[15]. G. R. Garay, J. Ortega, and V. Alarcón-Aquino, "Comparing Real-Time Calculus with the Existing Analytical Approaches for the Performance Evaluation of Network Interfaces," in Proceedings of the 21st IEEE International Conference on Electronics, Communications and Computers (CONIELECOMP 2011) Cholula, Puebla, México: IEEE, 2011, pp. 119-124.

[16]. G. R. Garay, J. Ortega, A. F. Díaz, L. Corrales, and V. Alarcón-Aquino, "System performance evaluation by combining RTC and VHDL simulation: A case study on NICs," Journal of Systems Architecture, vol. 59, pp. 1277-1298, 2013.

[17]. S. Shenker and J. Wroclawski, "General characterization parameters for integrated service network elements. RFC 2215.," IETF, 1997.

[18]. J.-Y. L. Boudec and P. Thiran, Network calculus: a theory of deterministic queuing systems for the internet: Springer-Verlag New York, Inc., 2001.

[19]. G. Aceto, A. Botta, W. De Donato, and A. Pescapè, "Cloud monitoring: A survey," Computer Networks, vol. 57, pp. 2093-2115, 2013.

[20]. A. Ahmed and A. S. Sabyasachi, "Cloud computing simulators: A detailed survey and future direction," in Advance Computing Conference (IACC), 2014 IEEE International, 2014, pp. 866-872.

[21]. A. Bahga and V. K. Madisetti, "Performance evaluation approach for multi-tier cloud applications," Journal of Software Engineering and Applications, vol. 6, p. 74, 2013.

[22]. D. Villela, P. Pradhan, and D. Rubenstein, "Provisioning servers in the application tier for e-commerce systems," ACM Transactions on Internet Technology (TOIT), vol. 7, p. 7, 2007.

[23]. Y. C. Lee and A. Y. Zomaya, "Energy efficient utilization of resources in cloud computing systems," The Journal of Supercomputing, vol. 60, pp. 268-280, 2012.

[24]. H. Khazaei, J. Misic, and V. B. Misic, "A fine-grained performance model of cloud computing centers," Parallel and Distributed Systems, IEEE Transactions on, vol. 24, pp. 2138-2147, 2013.

[25]. P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive control of virtualized resources in utility computing environments," in ACM SIGOPS Operating Systems Review, 2007, pp. 289-302.

[26]. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "A view of cloud computing," Communications of the ACM, vol. 53, pp. 50-58, 2010.

[27]. F. Xu, F. Liu, H. Jin, and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," Proceedings of the IEEE, vol. 102, pp. 11-31, 2014.

[28]. R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in Proceedings of the 5th European conference on Computer systems, 2010, pp. 237-250.

[29]. C. Anglano, M. Canonico, and M. Guazzone, "FC2Q: exploiting fuzzy control in server consolidation for cloud applications with SLA constraints," Concurrency and Computation: Practice and Experience, pp. n/a-n/a, 2014.

[30]. Z. Luo and Z. Qian, "Burstiness-aware Server Consolidation via Queuing Theory Approach in a Computing Cloud," in Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, 2013, pp. 332-341.

[31]. A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Modeling the Impact of Workload on Cloud Resource Scaling," in Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on, 2014, pp. 310-317.

[32]. Godofredo R. Garay, Andrei Tchernykh, Alexander Yu. Drozdov. An Approach for the Performance Evaluation of Multi-Tier Cloud Applications. II International Conference «Engineering & Telecommunication –En&T 2015», November 18–19, IEEE Computer Society, 2015

Сравнительный анализ методов оценки производительности многоуровневых облачных приложений[2]

[1]Г.Р. Гарай <godofredo.garay@reduc.edu.cu>
[2]А. Черных <chernykh@cicese.mx>
[3]А.Ю. Дроздов < alexander.y.drozdov@gmail.com>
[1]Университет Камагуэй, Куба
[2]Исследовательский центр CICESE, Мексика
[3]Московский физико-технический институт (государственный университет), 141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9

Аннотация. Аналитическое моделирование используется на ранних этапах проектирования аппаратуры, когда достаточно быстро должны быть рассмотрены многочисленные варианты. Оно позволяет провести оценку производительности предлагаемых систем без сложного и затратного моделирования. Наиболее популярные аналитические подходы к оценке производительности облачных вычислений включают модели теории очередей и теории контроля.Исчисление реального времени (Real-TimeCalculus- RTC) это аналитическая техника высокого уровня, первоначально разработанная для систем обработки потоков в режиме жесткого реального времени и часто используемая для нахождения баланса параметров в архитектурах обработки потока пакетов. Центральная идея модулярного анализа производительности с RTC (MPA-RTC) заключается в построении абстрактной модели производительности, которая связывает всю информацию, необходимую для анализа с исчислением реального времени.В этой статье мы рассматриваем оценку эффективности многоуровневых приложений для облачных вычислений, и сравниваем RTC с двумя классическими аналитическими подходами, такими как модели теории очередей и теории контроля.Мы сосредотачиваемся на возможностях этих альтернатив для оценки ключевого параметра качества обслуживания - времени ответа приложений.Кроме того, мы обсуждаем возможности каждого аналитического подхода для моделирования других аспектов среды облачных вычислений, таких как модели рабочей нагрузки, модели обработки задач, выделение ресурсов для виртуальных машин (VM), помех производительности виртуальных машин, автономное управление ресурсами, консолидацию серверов, а также стратегии масштабирования облачных вычислений (по горизонтали и / или по вертикали).

[2] Работы выполнены при финансовой поддержке Минобрнауки России (Соглашение № 02.G25.31.0061 12/02/2013).

Ключевые слова: исчисление реального времени, теория очередей, теория управления, QoS, облачные вычисления

DOI: 10.15514/ISPRAS-2015-27(6)-14

Для цитирования: Гарай Г.Р., Черных А., Дроздов А.Ю. Сравнительный анализ методов оценки производительности многоуровневых облачных приложений. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 199-224. DOI: 10.15514/ISPRAS-2015-27(6)-14.

Список литературы

[1]. X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin, "What does control theory bring to systems research?," ACM SIGOPS Operating Systems Review, vol. 43, pp. 62-69, 2009.

[2]. K. RahimiZadeh, M. AnaLoui, P. Kabiri, and B. Javadi, "Performance modeling and analysis of virtualized multi-tier applications under dynamic workloads," Journal of Network and Computer Applications, 2015.

[3]. X.-Y. Wang, L.-H. Fan, X.-H. Jia, and W.-T. Huang, "A Survey of Virtualization-based Resource Management in Cloud Computing Environments," Journal of Convergence Information Technology, vol. 8, 2013.

[4]. D. Huang, B. He, and C. Miao, "A survey of resource management in multi-tier web applications," IEEE Communications Surveys & Tutorials, vol. 16, pp. 1574--1590, 2014.

[5]. T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," Journal of Grid Computing, vol. 12, pp. 559-592, 2014.

[6]. N. Grozev and R. Buyya, "Performance modelling and simulation of three-tier applications in cloud and multi-cloud environments," The Computer Journal, vol. 58, pp. 1-22, 2015.

[7]. L. Kleinrock, Theory, Volume 1, Queueing Systems: Wiley-Interscience, 1975.

[8]. A. Willig, "A short introduction to queueing theory," Technical University Berlin, Telecommunication Networks Group, vol. 21, 1999.

[9]. T. Abdelzaher, Y. Diao, J. L. Hellerstein, C. Lu, and X. Zhu, "Introduction to control theory and its application to computing systems," in Performance Modeling and Engineering: Springer, 2008, pp. 185-215.

[10]. T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A systematic survey on the design of self-adaptive software systems using control engineering approaches," in Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2012, pp. 33-42.

[11]. E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse, "System architecture evaluation using modular performance analysis: a case study," Int. J. Softw. Tools Technol. Transf., vol. 8, pp. 649-667, 2006.

[12]. S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister, "Performance evaluation of network processor architectures: combining simulation with analytical estimation," Comput. Netw., vol. 41, pp. 641-665, 2003.

[13]. L. Thiele, S. Chakraborty, M. Gries, and S. Künzli, "A framework for evaluating design tradeoffs in packet processing architectures," in Proceedings of the 39th conference on Design automation New Orleans, Louisiana, USA: ACM, 2002.

[14]. S. Chakraborty, S. Künzli, and L. Thiele, "A General Framework for Analysing System Properties in Platform-Based Embedded System Designs," in Proceedings of the conference on Design, Automation and Test in Europe - Volume 1: IEEE Computer Society, 2003.

[15]. G. R. Garay, J. Ortega, and V. Alarcón-Aquino, "Comparing Real-Time Calculus with the Existing Analytical Approaches for the Performance Evaluation of Network Interfaces," in Proceedings of the 21st IEEE International Conference on Electronics, Communications and Computers (CONIELECOMP 2011) Cholula, Puebla, México: IEEE, 2011, pp. 119-124.

[16]. G. R. Garay, J. Ortega, A. F. Díaz, L. Corrales, and V. Alarcón-Aquino, "System performance evaluation by combining RTC and VHDL simulation: A case study on NICs," Journal of Systems Architecture, vol. 59, pp. 1277-1298, 2013.

[17]. S. Shenker and J. Wroclawski, "General characterization parameters for integrated service network elements. RFC 2215.," IETF, 1997.

[18]. J.-Y. L. Boudec and P. Thiran, Network calculus: a theory of deterministic queuing systems for the internet: Springer-Verlag New York, Inc., 2001.

[19]. G. Aceto, A. Botta, W. De Donato, and A. Pescapè, "Cloud monitoring: A survey," Computer Networks, vol. 57, pp. 2093-2115, 2013.

[20]. A. Ahmed and A. S. Sabyasachi, "Cloud computing simulators: A detailed survey and future direction," in Advance Computing Conference (IACC), 2014 IEEE International, 2014, pp. 866-872.

[21]. A. Bahga and V. K. Madisetti, "Performance evaluation approach for multi-tier cloud applications," Journal of Software Engineering and Applications, vol. 6, p. 74, 2013.

[22]. D. Villela, P. Pradhan, and D. Rubenstein, "Provisioning servers in the application tier for e-commerce systems," ACM Transactions on Internet Technology (TOIT), vol. 7, p. 7, 2007.

[23]. Y. C. Lee and A. Y. Zomaya, "Energy efficient utilization of resources in cloud computing systems," The Journal of Supercomputing, vol. 60, pp. 268-280, 2012.

[24]. H. Khazaei, J. Misic, and V. B. Misic, "A fine-grained performance model of cloud computing centers," Parallel and Distributed Systems, IEEE Transactions on, vol. 24, pp. 2138-2147, 2013.

[25]. P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive control of virtualized resources in utility computing environments," in ACM SIGOPS Operating Systems Review, 2007, pp. 289-302.

[26]. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "A view of cloud computing," Communications of the ACM, vol. 53, pp. 50-58, 2010.

[27]. F. Xu, F. Liu, H. Jin, and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," Proceedings of the IEEE, vol. 102, pp. 11-31, 2014.

[28]. R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in Proceedings of the 5th European conference on Computer systems, 2010, pp. 237-250.

[29]. C. Anglano, M. Canonico, and M. Guazzone, "FC2Q: exploiting fuzzy control in server consolidation for cloud applications with SLA constraints," Concurrency and Computation: Practice and Experience, pp. n/a-n/a, 2014.

[30]. Z. Luo and Z. Qian, "Burstiness-aware Server Consolidation via Queuing Theory Approach in a Computing Cloud," in Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, 2013, pp. 332-341.

[31]. A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Modeling the Impact of Workload on Cloud Resource Scaling," in Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on, 2014, pp. 310-317.

[32]. Godofredo R. Garay, Andrei Tchernykh, Alexander Yu. Drozdov. An Approach for the Performance Evaluation of Multi-Tier Cloud Applications. II International Conference «Engineering & Telecommunication –En&T 2015», November 18–19, IEEE Computer Society, 2015

Distributed Data Storage Systems: Analysis, Classification and Choice

Alexander Tormasov <tor@innopolis.ru>
Anatoly Lysov <a.lysov@innopolis.ru>
Emil Mazur <e.mazur@innopolis.ru>
Innopolis University, 1, str. Universitetskaya, Innopolis
Republic of Tatarstan, Russian Federation, 420500

Abstract. There are a large number of distributed data storage systems, and the vendors have different definitions of what is their solution: cloud storage, distributed file system, or a cluster file system, etc. This imposes difficulty in the selection of the distributed storage system, because it is not clear what indicators you should pay attention in the first place.
This paper proposes an analysis of various distributed data storage systems and possible solutions to basic problems of the subject area, in particular, the issue of system scaling, data consistency, availability and partition tolerance.
In this work we have ranked distributed storage systems based on various characteristics and have chosen the top of them for a further analysis. As the result of the analysis key system development patterns and trends were identified. These trends were further studied for correlations with systems functional and non-functional attributes.
Based on the performed analysis we have classified the systems by different criteria, including presence or absence of particular functions or attributes. In the course of a comparative study we have investigated basic system functionality (archive storage, deduplication, geo-replication etc.) and system performance (system scalability limits, architecture, operating environment etc.). In addition, we analyzed safety mechanisms and system self-management tools.
Based on the analysis data and the classification of the systems we have proposed methods for distributed data storage systems selection. The results of this work may be used by researchers and practitioners to make a justified choice of a storage systems for their specific needs.

Keywords: storage systems, distributed storage systems

DOI: 10.15514/ISPRAS-2015-27(6)-15

For citation: Tormasov Alexander, Lysov Anatoly, Mazur Emil. Distributed Data Storage Systems: Analysis, Classification and Choice. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 225-252 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-15

1. Introduction

Digital data volumes keep growing at a great pace. According to IBM statistics 2014, the daily amount of new information generated worldwide is about 15 petabyte. Meanwhile, the overall number of digital data doubles approximately every two year (Fig. 1).

Considering the fact that companies do not rush to expand the budget for data storage and support, the gap between the data volume growth and associated costs of database maintenance keeps rising [1].

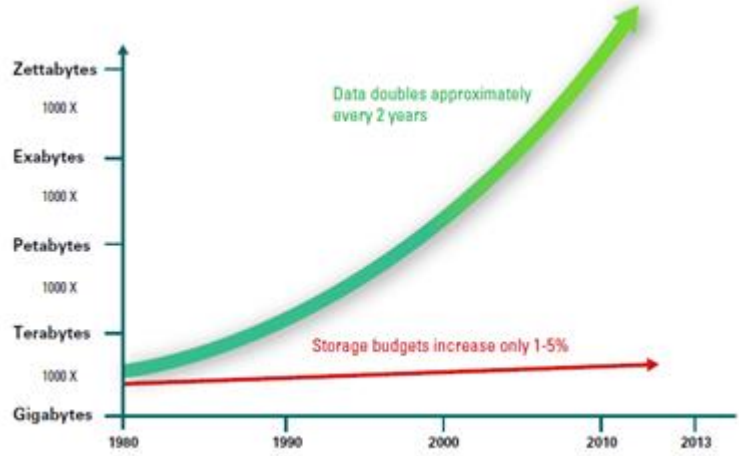


Figure 1: The growth of information

The majority of companies solve the problem of the explosive data growth via the purchase of hard drive arrays and network components thus expanding its data storage networks. This solution will eventually result in a complicated system administration (data backup, archiving etc.) and, accordingly, an increase in expenses on the system support. Thus, one can conclude that horizontal scaling is not reduce capital and operating costs associated with the database storage and maintenance.

Amid these problems, the network-wide dynamic boost of information regarding new solutions in the sphere of distributed data storage or software defined storage (SDS). SDS provide automated and policy-oriented storage services that consider the client and application-specific issues. SDS use a basic storage infrastructure and support of the software-defined environment in general. Correct deployment of SDS is a reliable solution for clients overwhelmed with large data volumes stored at complex hardware of different vendors [2]. SDS is a trend, so it's getting apparent that manufacturers have diverse opinions on the choice of the appropriate solution: cloud storage, distributed file system or cluster file system etc.

For the purpose of analysis, we combine all the solutions, including distributed file systems, cluster file systems and cloud platforms, in the group of the “Distributed Data Storage Systems”. Our objective is to study a large number of mentioned solutions and develop approaches to the choice of distributed data storage systems.

In order to achieve this goal we have set the following tasks:

1. Make a list of systems to be analyzed;
2. Study each system and carry out a comparative analysis of selected systems;
3. Classify systems according to the identified common factors;
4. Design approaches to the choice of distributed data storage systems based on data analysis and classification.

The paper will be useful for those who are interested in effective data storage and is looking for options to justify a certain solution including those who are concerned about the issue of distributed data storage in general.

2. Selection of Systems to be Analyzed

Let us run a bit ahead and say that our list comprises approximately one hundred elements. Compiling the list, we used Internet articles, references found in forums, comments to discussions and a Wikipedia article that turned out to be quite useful [3].

The list turned out to be quite long, so we decided to shorten the list it. It was suggested a formula to calculate the system rating based on two indexes: the number of relevant Google links and the average number of requests in Google Trends for 2014 [4]. The formula calculates the average weighted value and the result is shown in percentage.

Scaling the list according to the rating identified the market leaders:

1. Amazon Simple Storage Service - 29.49%
2. Google File System (in particular, GFS2 - Colossus) - 27.11%
3. Microsoft Azure - 12.00%

Systems with 1 - 10% rating can be referred to the second group:

4. Global File System 2 - 8.43%
5. Ceph - 5.22%
6. Hadoop FS - 3.75%
7. Windows DFS - 2.62%
8. Quantcast File System - 2.02%
9. Gluster - 1.76%

All other systems were referred to the third group: Self-certifying File System, Server Message Block FS, General Parallel File System, VMware Virtual SAN, Openstack SWIFT, ViPR, Microsoft SharePoint Workspace, Data ONTAP,

Rackspace Cloud Files, AcroStorage, Chord File System, OdinStorage, dCache, GridFS, Elliptic Network, Cassandra File System, ExaFS, Moose File System, Coda, Coherent Remote File System, MogileFS, Apple Filing Protocol, Starfish, Lustre, CloudStore, GLORY-FS, StarFS, SmartCloud Virtual, Farsite, NetWare Core Protocol FS, Chiron FS, Parallel NFS, Oceanstore, OpenAFS, Kyoto Tycoon, Arla, InterMezzo, Panasas ActiveScale File System, HAMMER/ANVIL, Sheepdog, MapR-FS, OneFS, Cleversafe, OS4000, Gfarm, Tahoe-LAFS, OrangeFS, zFS, XtreemFS, LeoFS, IBRIX Fusion, OriFS, IFS (EMC Isilon), TerraGrid, Unilium, BeeGFS, PlasmaFS, TorFS, WebDFS, PeerFS, NimbusFS.

Considering that nine systems are not enough for a profound analysis, it was decided to take several systems from the third group that seemed interesting for analysis.

In total, the final sampling included 30 systems listed below:

Amazon S3, Google File System, MS Azure, Global File System 2, Ceph, Hadoop FS, Windows DFS, Quantcast File System, Gluster, AcroStorage, OdinStorage, TorFS, VMware Virtual SAN, OpenStack SWIFT, ViPR, Rackspace, dCache, GridFS, Elliptics Network, MooseFS, CODA, Lustre, OceanStore, OpenAFS, Kyoto Tycoon, Arla, Tahoe, zFS, Leo FS, Andrew File System.

3. System Analysis

In the course of a comparative study we have investigated basic system functionality (archive storage, deduplication, geo-replication etc.) and system performance (system scalability limits, architecture, operating environment etc.). In addition, we analyzed safety mechanisms and system self-management tools.

3.1. Analysis of the Main System Functionalities

Based on the analysis of the main system functionalities and classification provided in “A Taxonomy of Distributed Storage System” we have identified the below mentioned functions and mechanism [5]:

- Archive storage function;
- Data compression function;
- Data deduplication function;
- Controlled redundancy mechanism based on (n,k)-scheme;
- User interface support mechanism for the file system;
- Shared data access mechanism;
- Geo-replication mechanism;
- Object storage mechanism;

3.1.1. Archive Storage Function

Archive storage enables data backup storage and retrieval with the main usage scenario named “cold storage” i.e. a single data recording aimed for the long-term storage. As a rule, such data is appealed to in emergency cases [6]. The chart below shows the percentage ratio of systems under analysis in terms of the archive storage function (Fig. 2).

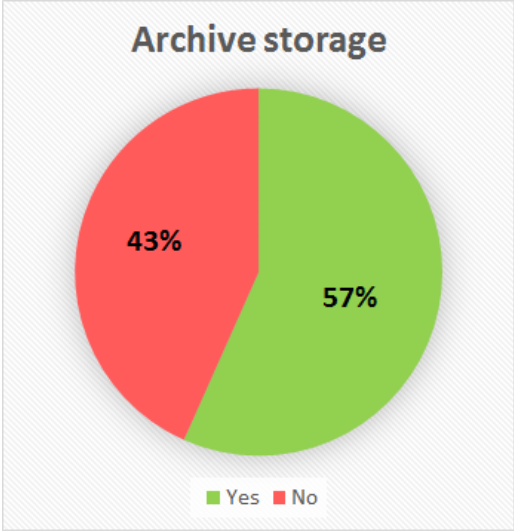


Figure 2: Archive storage

It should be noted, that the majority of systems under analysis are closed. In this and in other cases, if the functionality is not stated clearly, it is interpreted as missing.

According to the chart, the archive storage function is stated in more than a half of systems under analysis. Besides, this function is used by the rating leaders: Amazon S3, Google FS and MS Azure.

3.1.2. Data Compression Function

Data compression is algorithm-based data conversion for reducing the amount of data storage place [7]. According to suggested classification, systems can be divided into two groups: systems that use/ do not use this function (Fig. 3).

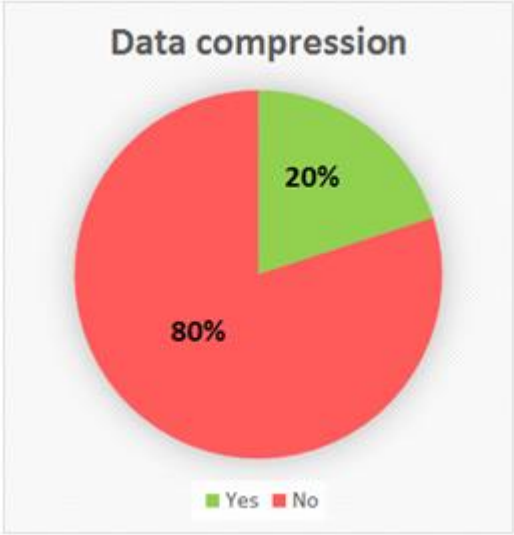


Figure 3: Data compression

Following the analysis, data compression function is used by less than a quarter of systems. It should be mentioned, that there is no "leading system" among them. Conversely, systems that identified the option of using data compression function are as follows: Ceph, HadoopFS, Gluster, RackSpace and dCashe.

3.1.3. Data Deduplication Function

Data deduplication is a technology for identification and eliminating duplicate copies of repeating data at the disk storage [8]. There are two types of deduplication: the file-level deduplication and the block-level deduplication.

In the file-level deduplication, for a deduplication unit is taken a single file serves. In this case, duplicating files are eliminated from the storage system.

In the block-level deduplication, for a deduplication unit is taken a variable length block, which is iterated in different logical objects of the data storage system [9].

Accordingly, the suggested classification includes 3 groups of systems (Fig. 4):

- Systems using the file-level deduplication function;
- Systems using the block-level deduplication function;
- Systems not using the deduplication function.

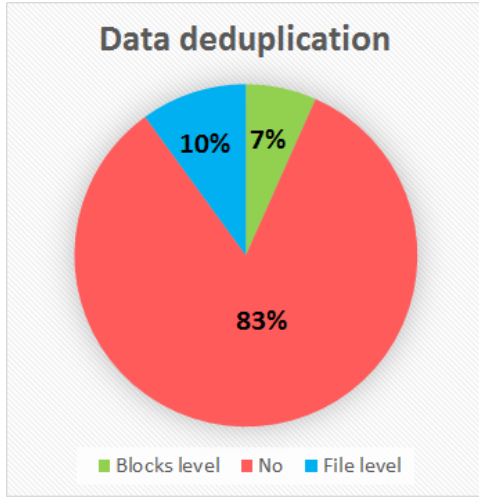


Figure 4: Data deduplication

Both data deduplication and data compression functions are supported by at least a quarter of the systems under analysis. Systems supporting the file-level deduplication are as follows: Ceph, RackSpace. Meanwhile, Amazon S3, HadoopFS and Windows DFS Systems belong to systems supporting the block-level deduplication.

Rating leaders: Google FS and MS Azure do not support deduplication.

3.1.4. Controlled Redundancy Mechanism based on (n,k)-Scheme

Here we are going to speak about a support of the so-called “erasure codes”. The nature of such codes is as follows: after coding of a certain number of files we get the “n” chunks of data. Each of them is stored at a single cloud storage. In order to restore the initial file, it is necessary to collect and decode any “k” chunks of data. It should be noted that n > k. The rest (n-k) chunks can be deleted, damaged, unavailable etc (Fig. 5). Thus, any system using the “erasure codes” can solve the problem of errors emerging in (n - k) chunks [10, 11].

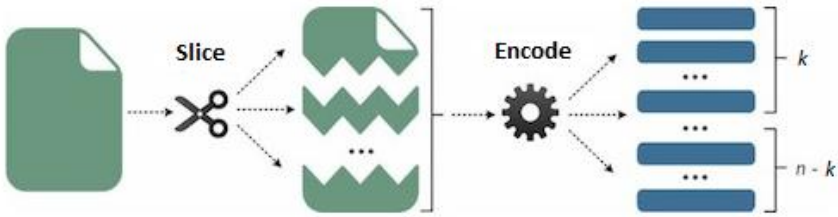


Figure 5: (n,k)-scheme

Accordingly, the suggested classification, systems can be divided into 2 groups: systems that support/ do not support such mechanism (Fig. 6).

Figure 6: Controlled redundancy

Following the chart, the controlled redundancy mechanism based on (n,k)-scheme is supported by at least a half of systems under study. As for the leading systems, MS Azure and Google FS are the only to support the function. Amazon S3 is likely to support the function in one form or another. However, the functionality was not clearly stated.

3.1.5. Shared Data Access Mechanism

Based on the analysis, the shared data access mechanism is not supported by every system. Besides, among systems supporting this mechanism there are systems requiring / not requiring user authorization (not to be confused with user authentication) [5].

Accordingly, the suggested classification includes 3 groups of systems (Fig. 7):

- Systems providing shared data access with required authorization (Sharing-systems);
- Systems providing shared data access without required authorization (Anonymity-systems);
- Systems not providing shared data access.

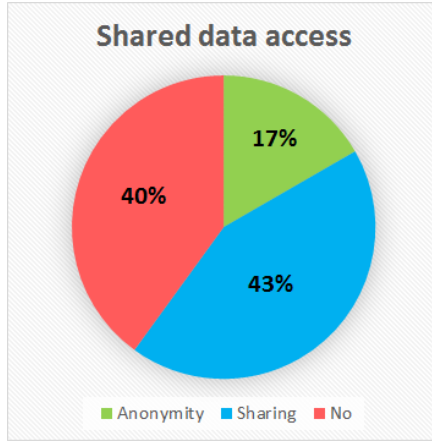


Figure 7: Shared data access

According to the chart, more than a half of systems use the shared access mechanism in one form or another. All the rating leaders support authorization-free access. HadoopFS and Rackspace refer to this group as well.

Systems that support the shared access to files and require authorization are as follows: Global File System 2, Ceph, QFS, Gluster, CODA, dCache, Elliptics Network, Leo FS, Lustre, SWIFT, ViPR, zFS, AFS.

3.1.6. Geo-Replication Mechanism

Geo-replication is a mechanism of synchronizing several object copies between geographically separated data centers [12]. Accordingly, systems can be divided into 2 groups: systems that support / do not support such mechanism (Fig. 8).

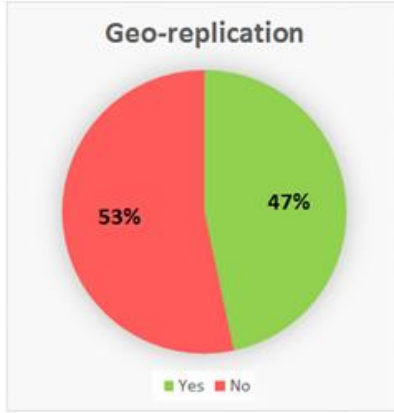


Figure 8: Geo-replication

Following the chart, over a half of systems under study have the geo-replication mechanism, including, in particular, the rating leaders: Amazon S3, Google FS and MS Azure.

3.1.7. Object Storage Mechanism

Object storage is a data storage architecture that unlike file system performs data management on the object level instead of blocks and sectors. [13].

Abstraction from multiple low-level storage tasks is one of the basic object storage principles. System administrators shall not be responsible for the logic volumes management, the disk space usage control and the RAID arrays configuration, which makes it easier to manage the storage system and reduces maintenance costs [13].

As opposed to classic arrays, object storage isolates the technological part of data storage, thus it enables easy system scaling, eliminates the hardware-dependence [14].

Thus, object storage provides relatively inexpensive scalable tool, which perfectly suits for effective storage of large non-structured data volumes. However, these advantages are achieved through lower requirements to data consistency. See the Section 2.6.6. for detailed information.

Coming back to the analysis, object storage is supported by almost a half of systems under study, including the rating leaders (Fig. 9).

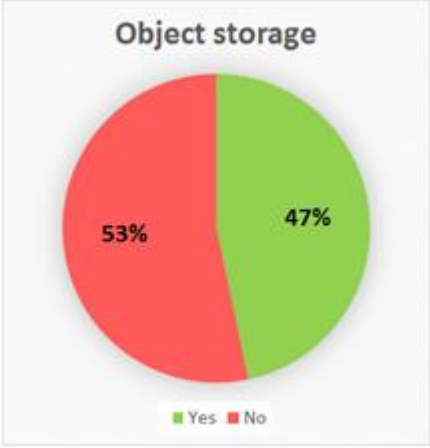


Figure 9: Object storage

3.2. Analysis of main system features

Besides the key system functionalities, comparative analysis included the investigation of non-functional system features. The following features were identified in the course of the analysis:

- System architecture;
- Scalability (based on data volume, number of servers, number of users);
- Cross-platformity (of server and client part);
- Access interface support NFS, SMB, https, WebDAV, S3 or proprietary protocol;
- Runtime environment;
- CAP theorem solution.

3.2.1. Architecture

Based on the analysis of system architectures, the latter were divided into two large classes: systems with client-server architecture and systems with peer-to-peer architecture. In the first case, a node acts as either a client or a server. Conversely, in the peer-to-peer architecture each node can be a client and a server at the same time.

In the client-server architecture, all control functions are concentrated in a single spot which provides the optimal safety and security level along with good performance. However, such concentration is a bottleneck, in particular, it affects system scalability and fault-tolerance because in case of a server failure it damages the whole system.

There are two types of the client-server infrastructure: globally-centralized and locally-centralized. In the first case, only one server is responsible for service provision and client servicing. Besides, all the problems related to the client-server architecture are apparent. In the second case, enhanced scalability and fault-tolerance can be achieved through the distribution of the control functions between groups of servers communicating with each other via data replication. Nevertheless, systems with the client-server architecture still have scalability limits.

As opposed to the client-server architecture, the peer-to-peer architecture is more suitable for untrusted environments where control and storage functions are distributed among all the nodes. Advantages of the peer-to-peer architecture refer to high scalability, self-management and fault-tolerance. However, such systems have their own shortcomings, including, low safety level, complex control if compared to the client-server architecture.

There are three types of the peer-to-peer architecture: globally-centralized, locally-centralized and pure peer-to-peer. In globally-centralized architecture, one of the system nodes serves as a central server which stores the information regarding other system nodes. Here the weak points are related with system scalability and fault-tolerance just like in the client-server architecture. In case of locally-centralized architecture, functions of the central server are distributed among several nodes with high performance. These nodes are called the super-nodes or the super-servers and their key function is to provide clients with information related to data location and accessibility.

Finally, the pure peer-to-peer architecture contains no specific nodes that identify the location of other nodes. Each node can be a server or a client which makes the peer-to-peer system well-adjustable to dynamic environment where the nodes can connect or disconnect from the network any time. As for the shortcomings, this type of infrastructure suffers from the low scalability and load asymmetry, requiring special mechanisms to remedy such defects. Besides, the system needs safety mechanisms to reduce the number of common threats [15].

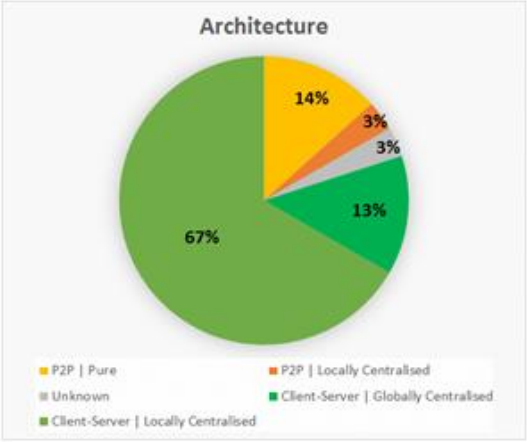


Figure 10: Architecture

Following the chart (Fig. 10), more than ¾ of systems under analysis have the client-server architecture. Meanwhile, the architecture of the majority of such systems refers to the locally-centralized one. The market leaders are among them.

Less than 20% of systems have the peer-to-peer architecture. We haven't found examples of globally-centralized architecture among them.

3.2.2. Scalability

We have analyzed suggested thresholds of system scalability according to the maximum volume of stored data, the number of servers and users who can operate with the system simultaneously.

When investigating the systems, we tried to avoid quality estimation and identified digital scalability indexes instead. It worked with just a part of systems. Some of them showed the abstract index "a lot". Another part showed unlimited capabilities. In addition, we could not find any information about the scalability thresholds of certain systems. In total, we have agreed that the "unlimited" is the maximum score, the "a lot" is 1 order lower than the "unlimited", while the "unknown" is a minimum score.

As a result, we have identified the following scalability range:

- Data Volume:
  Unknown < Petabytes (10^{15}) < Exabytes (10^{18}) < Brontobytes (10^{27}) < Not limited
- Number of servers:
  Unknown < x10 < $x10^2$ < $x10^3$< A lot < Not limited
- Number of users:
  Unknown < $x10^2$ < $x10^3$< $x10^4$ < $x10^6$< Not limited

The chart below integrates all three features and shows the whole picture (Fig. 11):

Figure 11: Scalability

According to the chart, one can make the following conclusions:

- On average, 7 systems claim to have unlimited capabilities or, in particular, their scalability threshold is limited solely with the company budget. Also, it should be noted that none of the rating leaders is among such systems;
- More than the third part of systems measure their data volume threshold in petabytes with only a few of them measuring the data volume threshold in exabytes and brontobytes;
- Among the sixth part of systems under analysis, none of them showed any information about scalability thresholds;
- In conclusion, the following trend is observed: a number of systems determine their limits as follows: “multiple servers”, “millions of users” and “petabytes of data”.

3.2.3. Cross-Platformity

Analysis of the platforms' ability to support the server and the client part showed the following results (Fig. 12):

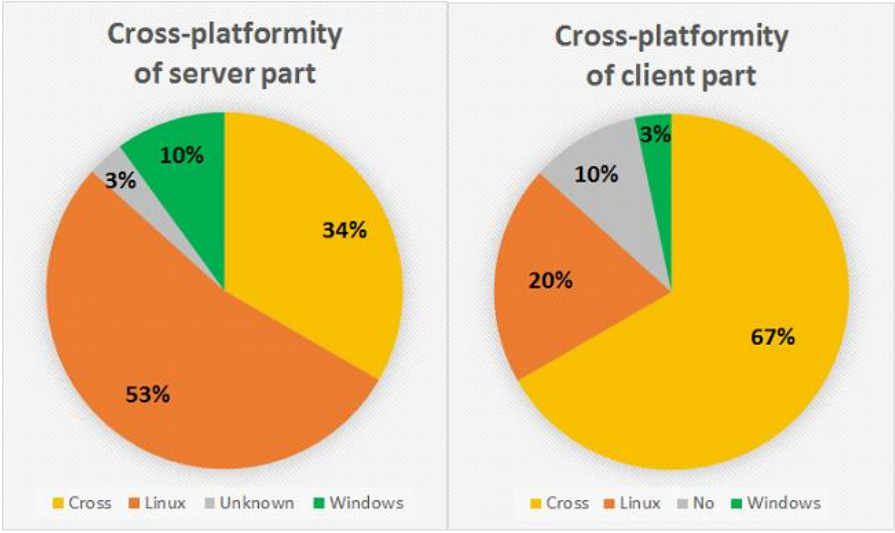


Figure 12: Cross-platformity

If we are talking about the server part, more than a half of systems prefers Unix-platforms. The rating leaders, in particular, Amazon S3 and Google FS, are among them. The leader, MS Azure refers to 10%, with its server part based on Windows.

The two third part of system under analysis have the cross-platform client part, including all three rating leaders.

3.2.4. Support of Access Interfaces

Support of access interfaces was one of the criteria of system analysis, in particular, the possibility to connect (mount) the NFS and SMB file systems as well as REST API (S3) and WebDAV access protocols.

See the chart below to find the information on distribution of systems in terms of the number of supported access interfaces mentioned above (Fig. 13).

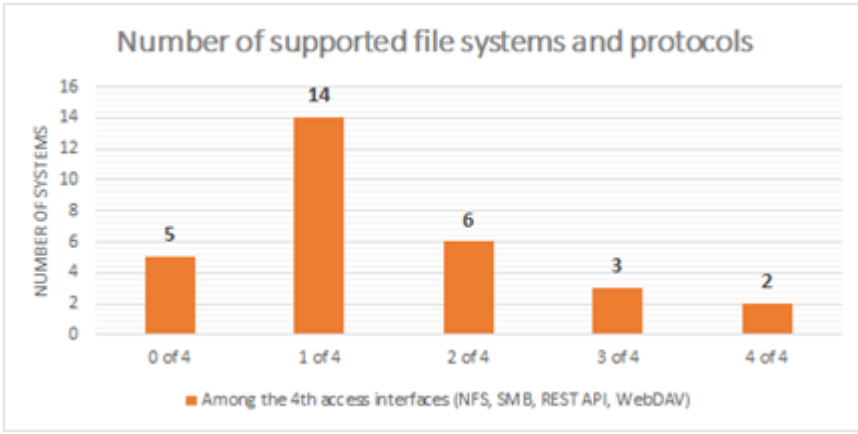


Figure 13: Number of supported file systems and protocols among the 4th kind of access interfaces (NFS, SMB, REST API, WebDAV)

However, the chart above does not take into account the proprietary access interfaces available in the majority of systems. If we add this information, the chart will look as follows (Fig. 14):

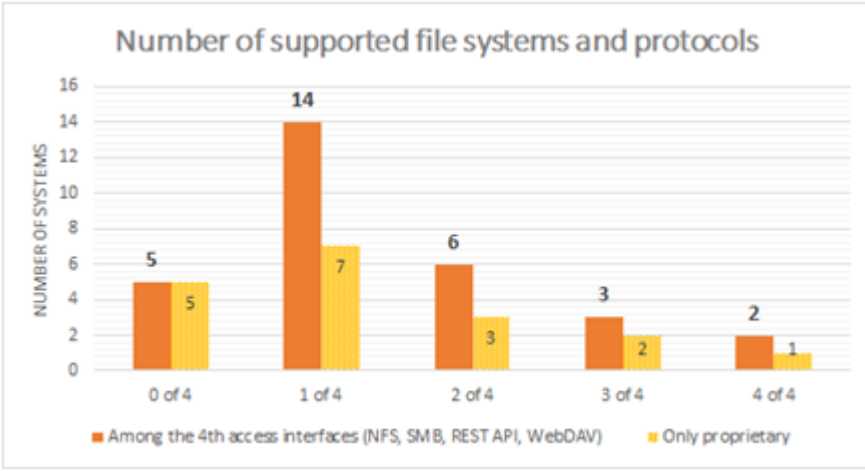


Figure 14: Number of supported file systems and protocols among the 4th kind of access interfaces (NFS, SMB, REST API, WebDAV) and proprietary interfaces

According to the second chart, most of the systems have their own access interfaces. It should be noted that only two systems support all 4 interfaces. These systems are Ceph and HadoopFS. Besides, Ceph has its own proprietary interface. Also, there

are 5 systems, supporting only proprietary interfaces. The leader Google FS along with zFS, TorFS, QFS, OceanStore are among them.

The rating leaders Amazon S3 and MS Azure refer to the group "1 of 4" supporting REST API (S3) protocols.

The chart below shows that the most "popular" interfaces are NFS and REST API. Also, more than a half of systems have their own proprietary access interfaces (Fig. 15).

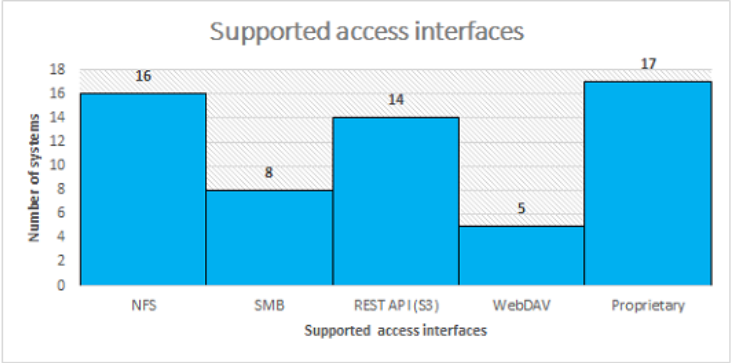


Figure 15: Supported access interfaces

3.2.5. Operating Environment

The operating environment is an important functional aspect of any distributed system. Trusted infrastructure is usually isolated from alien networks, which makes it predictable and easy-to-administrate. Controlled environment provides a high-quality servicing and trust. However, it adds to a substantial scalability limitation. Conversely, the untrusted environment implies a close interaction with the open access networks. In the open environment, it is hard or practically impossible to keep any records of users or control them. Systems located in the untrusted environment are subject to multiple attacks. Thus, such systems shall be equipped with additional safety mechanisms.

Based on our analysis, we have singled out 4 main types of the operating environments: Trusted, Partially trusted, Untrusted and Alien Infrastructure.

If a system is deployed on its own infrastructure, i.e. in the local corporate network on its own hardware, including the cases when any other software is not installed on the hardware platform, such environment shall be considered ***trusted***.

Likewise, if the system is deployed on its own infrastructure, i.e. in the local corporate network and installed on its own hardware, given that at least one different-type software is installed on the hardware platform, such environment shall be considered ***partially-trusted.***

If such system is deployed on its own hardware but is connected through the alien network, in particular, the Internet, such environment shall be considered ***untrusted.***

If the system is deployed on the alien hardware and connected through the alien network, in particular, the Internet, such environment shall be considered the ***alien infrastructure***.

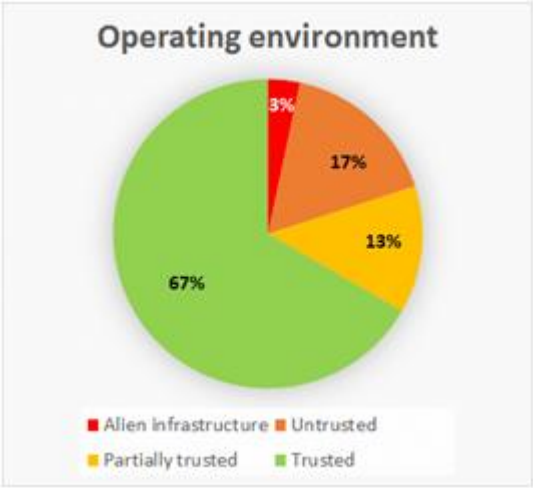


Figure 16: Operating environment

According to the chart (Fig. 16), only the third part of systems is able to operate in untrusted environments. Tahoe, however, can operate in the environment called the "alien infrastructure". In Tahoe, when the access to resources is opened, it is followed by data encryption, encoding and caching based on the (n-k) scheme or the principle of "minimal privileges" [16].

Amazon S3 and MS Azure, the rating leaders, refer to 67% of systems, that can operate only in the trusted environment.

Google FS, the rating leader, belongs to 13% systems, that can operate in partially-trusted environments.

3.2.6. Solutions to the CAP-theorem

According to Wikipedia, the CAP theorem (also known as Brewer's theorem) is a heuristic statement of the fact that in any distributed computing can provide only two out of three features as follows:

- Data consistency i.e. all computing nodes see the same data;
- Availability i.e. every request to the distributed system receives a correct response;

- Partition tolerance i.e. partition of the distributed system into several separate sections does not result into incorrect response received from each section [17].

In the second half of 2000-s, Brewer introduced the acronym BASE (Basically Available, Soft-State, Eventually Consistent) that implied that the requirements of integrity and availability are only partially fulfilled.

Consistency models are numerous. However, the systems under analysis can be referred to the following three models:

- Strong consistency;
- Eventual consistency;
- Weak consistency.

Strong consistency model guarantees that after the update any further data access will restore the updated values [18].

Eventual consistency model guarantees that in case of no data changes all the inquires will be eventually returned to the latest updated value.

Weak consistency model guarantees that further data inquires will restore the updated value. Before the updated value is restored it is necessary to fulfill a certain requirements. The inconsistency window is the time between the update and the moment when each user is sure to see the updated value.

To see the full picture and understand which of the "two out of three" parameters of CAP-theorem are fulfilled by the systems under analysis, please refer to the chart below that integrates all three parameters (Fig. 17):

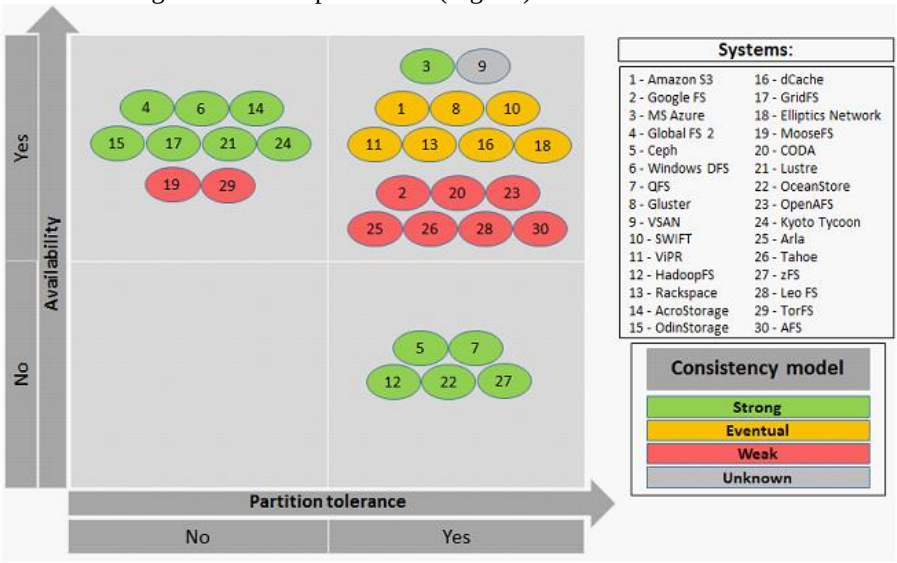


Figure 17: CAP-theorem

According to the chart, more than a half of systems are “PA” i.e. cannot guarantee consistency, but are CAP-available and partition tolerant. Besides, one half of such systems supports the eventual consistency while the other half supports the weak consistency.

The rating leaders also belong to such systems: Amazon S3 supports the eventual consistency, Google FS – the weak consistency. It is worth mentioning MS Azure. Azure developers claim to have found solutions for all three problems, providing strong consistency, availability and resilience. Meanwhile, this solution is valid for only certain types of system failures, in particular, “node failures” and “top-of-rack” failures. To provide this solution, strict consistency and availability are divided into two levels: Stream Layer and Partition Layer accordingly. In case of the node / workstation failure, the stream layer switches to the properly functioning node/workstation and keeps operating (reading/recording). Meanwhile, the partition layer identifies lost data and duplicates them to the properly functioning nodes/ workstations [19].

The second rating position belongs to the “CA”-systems, that cannot guarantee the partition tolerance but support strict consistency and availability. Here, 2 systems are worth mentioning. Technically, they cannot be referred to the “CA”-systems, because they support weak consistency.

The third rating position is taken by the “PC”-systems, that cannot guarantee availability but support strict consistency and partition tolerance.

3.3. Analysis of Safety Functions and System Self-Management Tools

3.3.1. System Safety Functions

The main system safety functions refer to the following:

- User authentication;
- Data access management;
- Data privacy;

User authentication can rely either on the local base, located in the system, or on the network authorization protocols (Kerberos, Radius, LDAP etc.).

Access control list (ACL) is a tool that is often used to solve the problem of the data access management.

Data privacy is provided through the encryption mechanism.

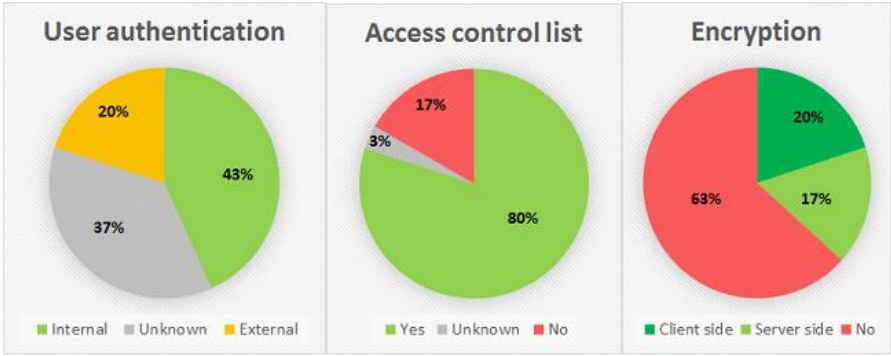


Figure 18: System safety functions

According to the charts (Fig. 18):

- Most of systems, including the rating leaders, use the local user base for authentication. However, it should be taken into account that no information regarding authentication mechanisms was found in respect of the majority of systems;
- More than a three quarter of systems under study, including the rating leaders, use access control lists;
- Almost two-thirds of systems do not use encryption. Among all the rating leaders, encryption is used only by Amazon S3. Encryption is based on the client part and employs symmetric encryption algorithm AES in GCM mode with 256-bit encryption keys [20].

3.3.2. Analysis of system self-management mechanisms

System self-management is a process when computer systems manage their own operation without human interaction. Modern distributed computer systems are heterogeneous and represent a combination of various information technologies combining network, mobile and wireless technologies. Manual control of such systems is complicated and labor-consuming which is the main aspect decelerating the development of such systems [21].

IBM is the major contributor to the development of self-managing systems. In 2001, the company created a special initiative group. IBM singles out 4 main features of self-managing systems [22]:

- **Self-Configuration** is the ability of a system to coordinate the values of low-level parameters (for instance, components set-up) with the high-level rules determined by business goals, and apply such parameters;
- **Self-Optimization / Self-Adaptation** is the ability of a system to provide continuous control and management of its resources in order to achieve the most efficient operation depending on certain requirements;

- **Self-Healing** is the ability of a system to identify and fix the occurred errors and heal in case of the single components failure using all possible means;
- **Self-Protection** is the ability of a system to protect itself from any harmful effects or successive failures. Systems operating on the Internet are exposed to a wide range of attacks. Thus, self-protection is of special importance.

The chart below (Fig. 19): shows that self-healing and self-optimization functions are dominating among the systems under study. Self-configuration and self-protection functions are less frequent.

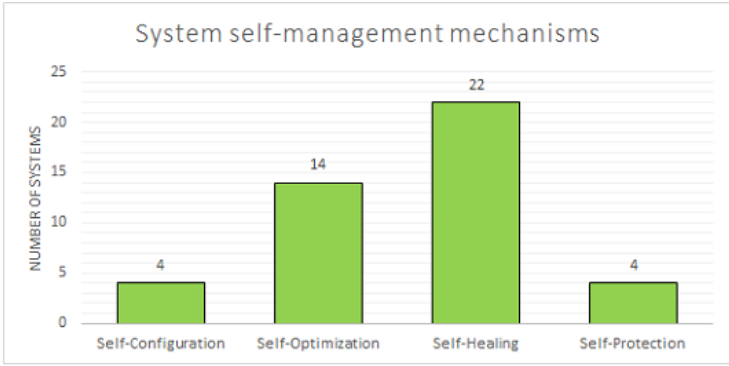


Figure 19: System self-management mechanisms

Self-healing function can be as follows:

- Healing after the disk failure;
- Healing after the node failure;
- Healing after the datacenter failure.

The chart below shows (Fig. 20) the results of the analysis in terms of available self-healing functions. More than two-thirds of systems support healing functions after the failure of disks and servers. Less than the third part of systems support the healing function after the collapse of data-centers.

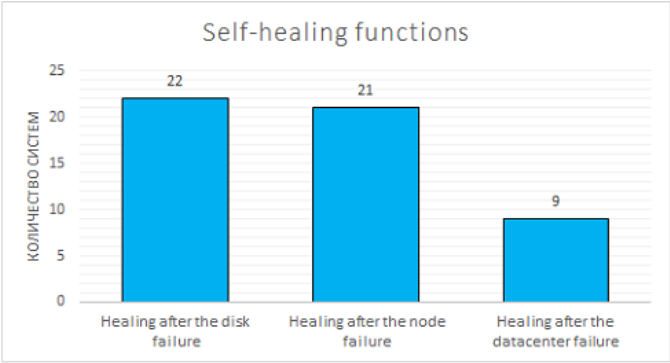


Figure 20: System self-healing functions

The **self-optimization** functions can be referred to the following:

- Optimizing consistency level is the selection of efficient consistency level of transferred data based on the analysis of users activity;
- Caching mechanism. The name speaks for itself. Self-adaptation is understood as the analysis of the application load and self-configuration of caching parameters;
- Adaptation of operation with SSD i.e. identification of speed-critical elements (caches, journals etc.), based on the analysis of the data inquiry frequency and their reading/ recording time including storage migration to SSD drives;
- Predicting disk failure i.e. use of the hard drive condition technologies like the "SMART", and prediction of the disk failure period;
- Load balancing i.e. keeping record of the network functioning and acceptance of distributed data transmission to other nodes;
- Control of electricity i.e. power-off in order to save on disks that store redundant information;
- Control of content popularity i.e. calculation of data inquires and approval of the popular content replica increase or, conversely, deleting data which was not requested for a long time.

Results of the analysis in terms of self-optimization functions are illustrated on the chart below (Fig. 21). Apparently, caching is the most popular function supported by almost half of systems. Over one third of systems support the self-adaptation function with SSD. None of the systems demonstrated control of electricity.

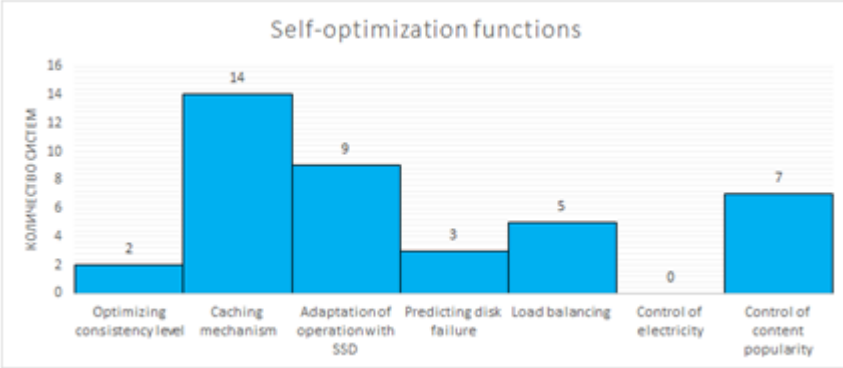


Figure 21: System self-optimization functions

4. Findings

According to the analysis, the broad market of distributed data storage systems lacks a single approach to design of such systems. In most cases, manufacturers rely on business tasks and try to reach the balance between high performance, scalability, safety and easy management, which makes quite a challenging task.

Below is the list of our findings that illustrate why certain functions and mechanisms are necessary for the distributed systems in terms of business tasks.

1. If your company stores data that does not require the every-day access but need to be stored for a long time, for instance, in accordance with the legal requirements, it is reasonable to use the archive storage mechanism. In most cases, the so-called “cold” data storage is more cost-efficient as opposed to the “hot” data storage.
2. Distributed storage is de facto more partition tolerant, so according to the CAP-theorem, the question of choice remains between the data consistency and data availability. If you do not store data that is always in-demand, it would be more rational to sacrifice the data consistency and use the “PA-model” with the weak data consistency.
3. The mechanisms like (n,k)-encoding, data compression and deduplication will sufficiently reduce the amount of stored information. Meanwhile, such system will be more expensive.
4. No one is secured from natural and man-induced disasters [23]. So, if you want your information to be protected from such factors, it would be wise to store data in the remote places geographically separated from each other using the geo-replication mechanism.
5. If the storage access is supported via the untrusted data channel, it’s apparent that encryption mechanisms shall be used to provide data privacy. However, if data processing and retrieval occurs in the controlled area, encryption can be ignored in order to enhance the performance.
6. It makes no sense trying to affect the architecture, cross-platformity, operating environment and object storage due to the conceptual and system-forming nature of

these features that determine all the rest functions and mechanisms. As for the scalability, it depends largely on the company's budget.

A small number of systems have the full range of self-awareness and self-management functions. Nevertheless, we believe that the development of such technologies is quite promising. Self-awareness tools make the system more flexible, which is important in terms of current business realities.

5. Conclusion

In this research paper we have analyzed over 30 distributed data storage systems. In the course of work it became apparent that the majority of systems under study are closed. For this reason, it was decided that if the functionality was not clearly specified, it is missing. Thus, it is most likely that statistic data described in this paper do not show 100% real picture.

Despite this fact, in this paper we have managed to classify systems in accordance with various parameters, including basic functional capabilities, features, safety mechanisms and self-management which served as the basis for approaches to distributed data storage systems. Thus, one can say that the objectives of these research work were completed in full and the goal was achieved.

Acknowledgements. This work has been supported by the Russian Ministry of education and science with the project "Development of new generation of cloudy technologies of storage and data control with the integrated security system and the guaranteed level of access and fault tolerance". (agreement: 14.612.21.0001, ID: RFMEFI61214X0001).

References

[1]. IBM Platform Computing Edition, Software Defined Storage For Dummies,New Jersey, 2014, pp.4-5 .

[2]. Software-Defined Storage (SDS) - perspektivy rosta [growth prospects] - ChannelForIT. Available at: http://channel4it.com/blogs/Programmno-opredelyaemye-hranilishcha-vsyo-bolee-vostrebovany-6787.html (accessed 23 July 2015). (In Russian)

[3]. List of file systems - Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/wiki/List_of_file_systems (accessed 23 July 2015).

[4]. Google Trends. Available at: https://www.google.com/trends (accessed 23 July 2015).

[5]. M.Placek, R. Buyya, A taxonomy of distributed storage systems, p. 53.

[6]. Kak Facebook sekonomil 75% energii, kotoraya trebuetsya dlya khraneniya dannykh pol'zovatelei [Fasebook saved 75% of the energy required to store users' data] / King Servers company's blog / Habrahabr. Available at: http://habrahabr.ru/company/kingservers/blog/257699/ (accessed 23 July 2015). (In Russian)

[7]. Data compression - Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/wiki/Data_compression (accessed 23 July 2015).

[8]. Deduplikatsiya dannykh — podkhod NetApp [Deduplication - NetApp's approach] / NetApp Company's Blog / Habrahabr. Available at: http://habrahabr.ru/company/netapp/blog/110482/ (accessed 23 July 2015). (In Russian)

[9]. Vvedenie v deduplikatsiyu dannykh [Introducition to data duplication] / Veeam Software company's blog. Available at: http://habrahabr.ru/company/veeam/blog/203614/ (accessed 23 July 2015). (In Russian)
[10]. Erasure code - Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/wiki/Erasure_code (accessed 23 July 2015).
[11]. O chem stoit zadumat'sya, sokhranyaya svoi dannye v oblake. Chast' 2 [What you should thinking about, when you saving data in cloud. Part 2] / Habrahabr. Available at: http://habrahabr.ru/post/141487/ (accessed 23 July 2015). (In Russian)
[12]. Vysokaya dostupnost' web-saita: georeplikatsia failov saita s "lsyncd" [Web site's high availability: file geo-replication of site with "lsyncd"] / Habrahabr. Available at: http://habrahabr.ru/company/infobox/blog/252751/ (accessed 23 July 2015). (In Russian)
[13]. Object storage - Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/wiki/Object_storage (accessed 23 July 2015).
[14]. Ob''ektnaya sistema khraneniya dannykh - konkurent zheleznykh SKHD [Object system of data storage - the competitor of hardware SAN]. Available at: http://www.jetinfo.ru/stati/konkurenty-zheleznykh-skhd (accessed 23 July 2015). (In Russian)
[15]. Y. R'kaina, Reliable and persistent storage for CoDeS a distributed collaborative system // 2013, p. 40
[16]. An Overview of Tahoe-LAFS. Secure and fault tolerant distributed storage system. Available at: https://code.google.com/p/nilestore/wiki/TahoeLAFSBasics (accessed 23 July 2015).
[17]. CAP theorem - Wikipedia, the free encyclopedia. Available at: https://ru.wikipedia.org/wiki/Теорема_CAP (accessed 23 July 2015). (In Russian)
[18]. Soglassovannye v konechnom schete [Eventually Consistent]. Available at: http://habrahabr.ru/post/100891 (accessed 23 July 2015). (In Russian)
[19]. B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. Fahim ul Haq, M. Ikram ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, L. Rigas Windows Azure Storage: a highly available cloud storage service with strong consistency // SOSP '11 Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, 143-157.
[20]. M. Campagna, AWS Key Management Service Cryptographic Details, 2015, p. 28
[21]. Autonomic computing - Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/wiki/Autonomic_computing (accessed 23 July 2015).
[22]. Self-management (computer science) - Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/wiki/Self-management_(computer_science) (accessed 23 July 2015).
[23]. Google poteryal chast' dannykh pol'zovatelei iz-za udara molnii - BBC Russkaya sluzhba [Google lost a part of users' data because of a lightning strike - BBC Russian Service]. Available at: http://www.bbc.com/russian/international/2015/08/150819_google_lightning_data (accessed 25 August 2015). (In Russian)

Распределенные системы хранения данных: анализ, классификация и варианты выбора

Александр Тормасов <tor@innopolis.ru>
Анатолий Лысов <a.lysov@innopolis.ru>
Эмиль Мазур <e.mazur@innopolis.ru>
Университет Иннополис, 4200500, Россия, Республика Татарстан, г. Иннополис, ул. Университетская, д.1

Аннотация. В статье содержится анализ различных распределенных систем хранения данных и возможных решений основных проблем этой области, в частности, проблем масштабирования систем, согласованности данных, доступности и устойчивости к разделению. Проведенный анализ позволил выявить ряд закономерностей и попытаться классифицировать системы на основе разных параметров, в частности, при наличии или отсутствии специфических функций и механизмов. Варианты выбора распределенных систем хранения данных основываются на анализе и классификации.

Ключевые слова: системы хранения, распределенные системы хранения

DOI: 10.15514/ISPRAS-2015-27(6)-15

Для цитирования: Тормасов Александр, Лысов Анатолий, Мазур Эмиль. Распределенные системы хранения данных: анализ, классификация и варианты выбора. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 225-252. DOI: 10.15514/ISPRAS-2015-27(6)-15.

Литература

[1]. IBM Platform Computing Edition, Software Defined Storage For Dummies,New Jersey, 2014, pp.4-5 .
[2]. Программно-определяемые хранилища всё более востребованы – ChannelForIT. http://channel4it.com/blogs/Programmno-opredelyaemye-hranilishcha-vsyo-bolee-vostrebovany-6787.html (дата обращения 23 July 2015).
[3]. List of file systems - Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/List_of_file_systems (дата обращения 23 July 2015).
[4]. Google Trends. https://www.google.com/trends (обращение 23 July 2015).
[5]. M.Placek, R. Buyya, A taxonomy of distributed storage systems, p. 53.
[6]. Как Facebook сэкономил 75% энергии, которая требуется для хранения фоточек котиков и селфи пользователей / Блог компании King Servers / Habrahabr. http://habrahabr.ru/company/kingservers/blog/257699/ (дата обращения 23 July 2015).
[7]. Data compression – Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Data_compression (дата обращения 23 July 2015).
[8]. Дедупликация данных – подход NetApp / Блог компании NetAppg / Habrahabr. http://habrahabr.ru/company/netapp/blog/110482/ (дата обращения 23 July 2015).

[9]. Введение в дедупликацию данных / Блог компании Veeam Software. http://habrahabr.ru/company/veeam/blog/203614/ (дата обращения 23 July 2015).
[10]. Erasure code - Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Erasure_code (обращение 23 July 2015).
[11]. О Чем Стоит Задуматься, Сохраняя Свои Данные в Облаке. Часть 2 / Habrahabr. http://habrahabr.ru/post/141487/ (дата обращения 23 July 2015).
[12]. Высокая доступность веб-сайта: георепликация файлов сайта с lsyncd / Habrahabr. http://habrahabr.ru/company/infobox/blog/252751/ (дата обращения 23 July 2015).
[13]. Object storage - Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Object_storage (дата обращения 23 July 2015).
[14]. Объектная система хранения данных - конкурент железных СХД. http://www.jetinfo.ru/stati/konkurenty-zheleznykh-skhd (дата обращения 23 July 2015).
[15]. Y. R'kaina, Reliable and persistent storage for CoDeS a distributed collaborative system // 2013, p. 40
[16]. An Overview of Tahoe-LAFS. Secure and fault tolerant distributed storage system. https://code.google.com/p/nilestore/wiki/TahoeLAFSBasics (дата обращения 23 July 2015).
[17]. Теорема CAP – Wikipedia, the free encyclopedia. https://ru.wikipedia.org/wiki/Теорема_CAP (дата обращения 23 July 2015).
[18]. Согласованные в конечном счете (Eventually Consistent) . http://habrahabr.ru/post/100891 (дата обращения 23 July 2015).
[19]. B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. Fahim ul Haq, M. Ikram ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, L. Rigas Windows Azure Storage: a highly available cloud storage service with strong consistency // SOSP '11 Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, 143-157.
[20]. M. Campagna, AWS Key Management Service Cryptographic Details, 2015, p. 28
[21]. Autonomic computing - Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Autonomic_computing (дата обращения 23 July 2015).
[22]. Self-management (computer science) - Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Self-management_(computer_science) (дата обращения 23 July 2015).
[23]. Google потерял часть данных пользователей из-за удара молнии – Русская служба BBC. http://www.bbc.com/russian/international/2015/08/150819_google_lightning_data (дата обращения 23 July 2015).

Модель надежности распределенной системы хранения данных в условиях явных и скрытых дисковых сбоев

[1]Л. В. Иваничкина <livanichkina@odin.com>
[2]А.П. Непорада <Andrew.Neporada@acronis.com>
[1]ООО Проект ИКС, 127566, Россия, г. Москва, Алтуфьевское ш, дом 44
[2]ООО Акронис, 127566, Россия, г. Москва, Алтуфьевское ш, дом 44

Аннотация. В настоящей работе рассматривается подход к расчёту надёжности хранилища данных, учитывающий как явные дисковые сбои, так и скрытые битовые ошибки, а также процедуры их выявления. Для расчёта надёжности предлагается новая математическая модель аналитического описания распределенного хранилища, описывающая динамику потерь и восстановления данных на основе Марковских цепей, соответствующих разным схемам избыточного кодирования. Рассматриваются преимущества разработанной модели по сравнению с классическими моделями для традиционных RAID-массивов. Исследуется влияние скрытых ошибок жестких дисков без учёта других битовых сбоев, возникающих в остальных аппаратных компонентах вычислительной машины. Оценка надёжности осуществляется согласно новым аналитическим формулам на основе критерия среднего времени до отказа, при котором утрата данных превышает порог восстанавливаемости, определяемый параметрами избыточного кодирования. Приводятся новые аналитические зависимости среднего времени жизни хранилища от среднего времени полной проверки данных в нём.

Ключевые слова: Среднее время до отказа (MTTF); цепи Маркова; избыточное кодирование; задача Гюйгенса о разорении игрока; распределенное хранилище данных; процедура скраббинга; контрольные суммы; MTTDL распределенного хранилища данных; дисковые сбои; скрытые битовые ошибки; секторные ошибки.

DOI: 10.15514/ISPRAS-2015-27(6)-16

Для цитирования: Иваничкина Л.В., Непорада А.П. Модель надежности распределенной системы хранения данных в условиях явных и скрытых дисковых сбоев. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 253-274. DOI: 10.15514/ISPRAS-2015-27(6)-16.

1. Введение

Хранилища данных петабайтного объема состоят из большого количества накопителей на жёстких магнитных дисках. Сохранность данных в распределенной системе в значительной степени зависит от надежности и рабочих характеристик используемых жестких дисков. Обеспечение надёжности хранилища в целом является важной прикладной задачей, сложность решения которой возрастает по мере увеличения количества задействованных накопителей.

Будучи сложным техническим устройством, жесткий диск подвержен сбоям, которые вызываются комплексом разнообразных случайных факторов и носят стохастический характер. С некоторой степенью приближения вероятность отказа жесткого диска можно описать статистическим законом, а затем обобщить процесс возникновения отказов на совокупность всех дисков хранилища.

Модель надёжности хранилища, описывающая серии отказов и замен жестких дисков до выполнения определенного условия, соответствует некоторой задаче Гюйгенса о разорении игрока. В классе задач Гюйгенса рассматривается игра с ограниченной начальной суммой при фиксированной ставке и известным математическим ожиданием выигрыша на каждом раунде, проводимая либо до приумножения начального капитала, либо до его полной потери. Широко используются два эквивалентных подхода к решению этой задачи, а именно: испытания Бернулли и случайное блуждание. Так, в методе случайного блуждания рассматривается стохастическое перемещение игрока по дискретным состояниям в зависимости от исхода очередного раунда.

В модели надёжности хранилища при допущении равнозначности и независимости всех дисков дискретные состояния соответствует количеству исправных дисков. Переход из состояния в состояние определяется явным сбоем, который обнаруживается сразу после возникновения. К таким сбоям относят, в частности, отказ программно-аппаратной части жесткого диска.

Поскольку в реальных приложениях применяется резервирование, сохранность данных обеспечивается при выходе из строя некоторой части накопителей. Максимальный порог отказавших дисков, выше которого происходит безвозвратная потеря данных, определяется схемой и параметрами избыточного хранения.

Еще одной технической особенностью жестких дисков является возникновение скрытых битовых ошибок, не выявляемых непосредственно на момент их возникновения. Битовые ошибки искажают хранимые данные, но не влияют на физическое функционирование устройства. Поскольку процедура восстановления запускается только после обнаружения ошибок, скрытые битовые ошибки отрицательно влияют на надёжность хранилища. Этот класс ошибок относится к невосстановимым ошибкам чтения (irrecoverable read errors) и должен приниматься во внимание при проектировании хранилища данных.

Основным способом борьбы с такими ошибками является вычисление контрольных сумм для фрагментов каждого блока и их дальнейшая проверка. При этом возможны несколько подходов к проверке данных.

В первом методе проверка контрольных сумм выполняется только при непосредственном запросе данных клиентом. По результатам проверки в случае несовпадения контрольных сумм инициируется процесс восстановления. В таком случае важным параметром, влияющим на надежность хранилища, будет средняя интенсивность доступа к данным. На практике интенсивность доступа к данным разного типа может сильно отличаться, поэтому риск потери данных для большого архивного файла, к которому почти нет обращений, может быть значительным. Исходя из этого, первый вариант не обеспечивает в достаточной мере требуемый уровень надежности хранения для всех файлов.

Второй подход к проверке контрольных сумм предполагает наличие в хранилище данных централизованного сервиса, который управляет непрерывным процессом проверки контрольных сумм фрагментов, этот подход также известен как скраббинг (scrubbing). При этом централизованному сервису необязательно самостоятельно заниматься пересчетом контрольных сумм, достаточно, чтобы он организовывал хранение информации, необходимой для проверки данных, а также посылал команды проверки данных сервисам хранения. Таким образом, нагрузка, связанная с проверкой данных, будет распределена по всему хранилищу. Интенсивность работы такого процесса принято оценивать через ожидаемое время проверки всего объема данных, которые содержатся в хранилище, и может составлять от нескольких суток до нескольких недель.

В настоящей работе предлагается новая математическая модель надежности хранения данных с оригинальным аналитическим описанием переходов между состояниями, учитывающая возникновение скрытых дисковых ошибок и непрерывный процесс скраббинга.

2. Обзор литературы

Классические модели надёжности хранения данных, построенные на основе Марковских цепей в непрерывном времени, дают ориентировочное представление о среднем времени до потери данных (MTTDL) хранилища, но не учитывают возникающие на дисках битовые ошибки. Эти модели рассмотрены в ряде работ (например, [1,2,3,4]) по исследованию надежности RAID-массивов в условиях явных дисковых сбоев. За последние годы всё больше научных работ посвящается построению расширенных Марковских моделей надёжности, включающих математические описания скрытых дисковых ошибок и процессов их обнаружения. Так, например, недостатки классических моделей с одномерными Марковскими цепями без учёта памяти подробно описаны в широко известной работе [5]. Контраргументы, в которых показана пригодность Марковских цепей для моделирования надёжности,

приводятся в работе [6], где эффект памяти воспроизводится в рамках более точного детального описания системы увеличенным количеством состояний и переходов между ними. Оригинальная модель для RAID-массива из SSD-накопителей с ростом интенсивности ошибок по мере износа предложена в работе [7]. Например, в работах [8,9,10,11] рассмотрено последовательное обобщение модели надёжности MTTDL для RAID-массивов на случаи возникновения скрытых невосстановимых секторных ошибок чтения, и применения процедуры проверки контрольных сумм. В опубликованных работах исследование свойств двумерной Марковской модели в силу её сложности осуществляется имитационными численными методами, включающими методы статистического моделирования. Несмотря на сложность описания двумерных цепей, Марковские модели остаются привлекательными для использования, поскольку позволяют оценивать надежность хранения данных для произвольной схемы избыточного кодирования, учитывать различные типы дисковых сбоев, различные состояния компонентов хранилища, а также различные политики замены сбойных компонентов и восстановления данных, такие как параллельное и последовательное восстановление, разные интенсивности восстановления для разного количества потерянных фрагментов. В отличие от имитационного моделирования, основанного на статистических испытаниях Монте-Карло, данный метод позволяет получить точное аналитическое выражение для надежности хранилища в рамках данной модели, а аналитическое выражение может быть использовано для выявления некоторых функциональных зависимостей между параметрами модели. Кроме того, Марковские модели сохраняют значительное преимущество в производительности и скорости расчётов (до 150 раз, см. [6]) по сравнению с полномасштабным имитационным моделированием.

Представленная модель развивает идеи описания традиционных аппаратных средств локального хранения данных в RAID-массивах из отдельных дисков и рассматривает перспективные распределенные системы хранения данных с избыточным кодированием [12], где отдельные фрагменты блоков данных не привязаны к дискам и перемещаются при репликации или балансировке нагрузки. В разработанной математической модели переопределяется семантика классических моделей для RAID-систем — состояния соответствуют фрагментам блока с избыточным кодированием, а не дискам. Аналогичным образом переходы из состояния в состояние описывают потерю или восстановление фрагмента данных, а не выход из строя индивидуального накопителя.

3. Базовая математическая модель

Пусть имеется блок данных, составленный из n закодированных фрагментов, в которых k фрагментов соответствуют исходным данным, а остальные $(n-k)$ фрагментов — контрольным суммам. При этом пусть схема

кодирования такова, что для восстановления данных достаточно любых k фрагментов. Блок данных характеризуется следующим набором состояний S_i: состояние S_0 – все n фрагментов доступны и ни одного фрагмента не повреждено, состояние S_1 – поврежден один фрагмент, состояние S_2 – повреждено 2 фрагмента, …, состояние S_{n-k} – повреждено $(n-k)$ фрагментов, состояние S_{n-k+1} – повреждено больше, чем $(n-k)$ фрагментов, что означает невозможность восстановить блок, т.е. потерю данных блока. Состояние $(n-k+1)$ в дальнейшем удобно обозначить названием DL (“data loss”) для выделения среди всех остальных состояний. Состояние DL является абсорбирующим, поскольку для него отсутствуют возвратные переходы в другие состояния системы. Согласно модели система за некоторое время вне зависимости от начального состояния с вероятностью 1 перейдет в абсорбирующее состояние. Для таких систем важной практической характеристикой является среднее время работы до перехода в абсорбирующее состояние.

Переход между состояниями системы описывает потерю фрагментов данных в блоке из-за сбоев дисковых накопителей. В качестве первого упрощающего допущения можно принять, что дисковые сбои подчиняются Пуассоновскому распределению, согласно которому вероятность выхода диска из строя в течение какого-либо периода времени не зависит от возраста диска. Выход из строя диска означает полную потерю хранящихся на нем данных, таким образом, потерянные фрагменты могут быть восстановлены только с использованием фрагментов, которые располагаются на работающих дисках.

Пусть интенсивность дисковых отказов равна λ, это означает, что ожидаемое время функционирования диска до отказа составляет $1/\lambda$. Для преобладающего большинства моделей дисковых накопителей интенсивность отказов составляет порядка 10^{-9} секунды. Вероятность сбоя диска в течение интервала времени $[0,t)$ определяется интенсивностью дисковых отказов и равна $\Pr(t_F < t) = 1 - e^{-\lambda t}$.

Вторым общепринятым допущением является предположение о независимости в совокупности сбоев индивидуальных дисков. В этом случае для n работающих дисков суммарная интенсивность отказов равна $n\lambda$, а ожидаемое время до сбоя (MTTF) хотя бы одного диска из n равно $1/(n\lambda)$.

Отказ диска определяет переход модели из состояния S_l с l неработающими дисками в состояние S_{l+1} с $(l+1)$ неработающими дисками. Поскольку в моделях с непрерывным временем совпадение моментов выхода из строя двух любых дисков считается случайным событием нулевой меры, можно считать, что непосредственные переходы между состояниями, различающимися более чем на один работающий диск, имеют нулевую вероятность, несмотря на априорную возможность самого такого события. Таким образом в модели определяющими событиями являются единичные дисковые сбои.

Следующим важным фактором модели является среднее время до восстановления фрагмента данных $1/\mu$. В отличие от интенсивности

дисковых отказов этот параметр определяется не только физическими свойствами используемых носителей данных, но и архитектурой конкретного распределенного хранилища, и может зависеть от многих факторов. Среднее время восстановления фрагмента данных складывается из времени, прошедшего от дискового сбоя до его обнаружения, и времени, прошедшего от обнаружения сбоя до момента, когда восстановленный фрагмент записан на один из дисков системы. Как правило, архитектура распределенного хранилища данных предполагает наличие сервиса мониторинга, который следит за состоянием всех блоков в хранилище данных и запускает процедуру восстановления данных при обнаружении сбоев, поэтому время до обнаружения сбоя даже при большом объеме данных, которые хранятся в системе, можно считать достаточно малым – порядка нескольких минут. Далее рассматривается обобщение классической модели на сценарии возникающих скрытых дисковых сбоев и процедур их обнаружения.

4. Расширенная модель

В этом разделе раскрыты характеристики стилей, используемых в данном документе.

4.1 Частота невосстановимых ошибок чтения

Как правило, частота невосстановимых ошибок чтения указывается производителем в спецификации диска и составляет порядка одной ошибки на 10^{14} битов или порядка одной ошибки на 11 Тб прочитанных с диска данных. С целью учета этого параметра в предложенной модели необходимо оценить, насколько часто такие ошибки будут происходить в течение жизни хранилища, а для этого требуется оценка среднего количества данных, которое в среднем считывается с одного диска в датацентре на протяжении некоторого характерного интервала времени, например, в течение года.

Среднюю интенсивность доступа к диску можно оценить, исходя из средней загруженности диска, указанную в спецификации. Пусть ожидаемая загруженность диска составляет порядка 20%, т.е. в течение 20% времени происходит доступ к диску, а 80% времени диск простаивает. Тогда установившаяся скорость последовательного чтения с диска равна 140 Мб/с, тогда в течение года с диска будет прочитано $(140 \cdot 0.2 \cdot 3600 \cdot 24 \cdot 365)/1024 = 862312$Гб или около 842 Тб. Исходя из этого, частоту невосстановимых битовых ошибок для одного диска по порядку величины можно оценить как 77 ошибок в год. Пусть размер диска составляет 2 Тб и он заполнен наполовину, а характерный размер одного фрагмента данных в хранилище равен 50 Мб, тогда средняя частота ошибок при чтении выбранного фрагмента блока данных будет равна приблизительно 0.00367 ошибок в год или около 1 ошибки в течение 272 лет.

В разработанную модель добавлен параметр c — частота невосстановимых ошибок чтения для одного фрагмента блока данных, в частном случае равная

$272лет^{-1}$. Этот параметр сравним по величине с интенсивностью дисковых сбоев, которую для дисков с MTTF=200000 часов можно оценить как $23лет^{-1}$, и должен быть учтен при оценке MTTDL хранилища данных.

Необходимо отметить, что этот параметр был рассчитан для сценария сравнительно высокой степени использования дисков. В общем случае частота невосстановимых ошибок чтения существенно зависит от интенсивности использования диска. Средняя ожидаемая загруженность диска, указанная в спецификации, является ориентировочным значением для оценок, но для конкретного хранилища данных лучше брать средние оценки загруженности дисков для данного хранилища. Например, архивное хранилище данных может иметь более низкую загруженность дисков. и в общем случае будет зависеть от интенсивности и паттернов доступа к данным для реального датацентра. Тем не менее, рассчитанное значение параметра позволит оценить снизу MTTDL хранилища с учетом наличия битовых ошибок, а также характер их влияния на надежность хранения данных.

4.2 Ожидаемое время полной проверки данных в датацентре

Среднее время выполнения полной проверки данных в датацентре, определяется средней интенсивностью проверки данных, т.е. средним объемом данных, проверяемым за единицу времени. Физически реалистичные значения для средней интенсивности проверки данных будут определяться как степенью загруженности хранилища и гарантиями производительности, которые оно предоставляет клиентам, так и экономическими затратами на выполнение проверки контрольных сумм.

В зависимости от загруженности хранилища пользовательские задачи доступа к данным и сервисные процессы работы с данными могут претендовать на одни и те же ресурсы и провоцировать конфликты доступа. В зависимости от конкретной архитектуры хранилища и ее задач такими ресурсами могут быть процессорное время, дисковые операции ввода-вывода, а также пропускная способность сети. В таком случае, как правило, ресурсы, используемые сервисными процессами, ограничиваются так, чтобы обеспечить клиентам гарантированный уровень производительности (гарантированное время отклика, гарантированное полное количество операций ввода-вывода).

В случае, когда реальная загруженность хранилища далека от предельных значений, а пользовательские задачи и системные процессы не требуют эксклюзивного доступа к одним ресурсам, интенсивность процесса проверки данных полностью определяется балансом между требованиями обеспечения надежности хранения данных и предельными затратами в процессе проверки контрольных сумм. Эти затраты определяются, во-первых, возможной необходимостью вывода диска, который содержит проверяемый фрагмент, из режима ожидания, в котором снижено его энергопотребление. Во-вторых, затраты связаны с необходимостью расходования процессорного времени на вычислительно требовательный алгоритм подсчета контрольных сумм и также

возможным выводом одного из ядер процессора из режима пониженного энергопотребления.

В качестве ориентировочного значения времени полной проверки данных можно выбрать характерный для в датацентра период времени от одной недели до одного месяца. Среднее время полной проверки также должно быть меньше, чем среднее время между запросами одних и тех же данных пользователя хранилища. значение интенсивности проверки данных как α.

В классической модели состояние распределенной системы хранения полностью определяется количеством вышедших из строя дисковых накопителей. В новой модели состояние системы зависит не только от явных поломок, но и от скрытых поломок, определяющих количество поврежденных фрагментов. Таким образом, расширенная модель, построена на двумерном, а не одномерном пространстве состояний Марковской цепи, в отличие от классической модели.

Пусть состояние системы (l, m) характеризуется явными сбоями l дисков, а также m скрытыми повреждениями фрагментов. При этом скрытые повреждения фрагментов учитываются только для работоспособных дисков, т.к. данные на вышедших из строя дисках считаются недоступными для чтения.

Можно отметить, что значение суммы $(l + m)$ для состояния, в котором данные блока все еще можно восстановить, ограничено сверху количеством фрагментов, которые можно утратить в данной схеме помехоустойчивого кодирования без потери данных, т.е. для такого состояния выполняется неравенство $0 \le l + m \le n - k$. Для состояний с условием $l + m > n - k$, данные блока восстановлению не подлежат, поэтому вне зависимости от соотношения скрытых и явных повреждений все эти состояния можно объединить в одно состояние DL («data loss»).

Таким образом общее количество состояний в системе, не считая состояния DL, будет равно

$$\sum_{i=1}^{n-k+1} i = (n - k + 1)\frac{n - k + 2}{2}.$$

Первый тип переходов между состояниями описывает дисковые сбои. Для состояния (l, m) возможны два варианта переходов. Во-первых, может выйти из строя диск, соответствующий одному из поврежденных фрагментов и система с интенсивностью $m\lambda$ перейдет в состояние $(l + 1, m - 1)$. Во-вторых, может отказать диск, соответствующий одному из неповрежденных фрагментов, и система с интенсивностью $(n - l)\lambda$ перейдет в состояние $(l + 1, m)$.

Второй тип переходов — это переходы из-за битовых ошибок в состояния со скрытыми повреждениями. Из-за битовых ошибок возможен переход из состояния (l, m) в состояние $(l, m + 1)$ с интенсивностью $(n - l - m)c$. При таком переходе не учитывается возникновение скрытых битовых ошибок в

уже вышедших из строя дисках, а также не рассматривается возможность появления новых битовых ошибок в уже поврежденных фрагментах, т.к. эти события не меняют состояния системы.

Переходы между событиями, связанные с восстановлением данных, происходят как из-за процесса восстановления данных при обнаружении явных сбоев, так и из-за процесса детектирования скрытых ошибок в данных. Пусть при восстановлении после явных сбоев происходит проверка контрольных сумм для фрагментов на работоспособных дисках, и наоборот при обнаружении скрытых ошибок происходит не только перезапись поврежденных фрагментов, но и восстановление фрагментов, потерянных в результате явных дисковых сбоев. Пусть также процесс восстановления данных всегда переводит систему в состояние (0, 0), в котором отсутствуют явные и скрытые повреждения. Переход в состояние (0, 0) обоснован тем, что значения MTTDL для процессов последовательного и параллельного восстановления данных не отличаются в случае, когда интенсивности процессов восстановления намного выше интенсивности дисковых сбоев. В зависимости от состояния системы интенсивность перехода в состояние $(0,0)$ меняется следующим образом – для состояний вида $(l,0), l > 0$ интенсивность равна μ, для состояний вида $(0,m), m > 0$ интенсивность равна α, а для состояний вида $(l,m), l > 0, m > 0$ интенсивность равна $(\alpha + \mu)$.

Переходы в состояние DL в новой модели отличаются от переходов в другие состояния, т.к. состояние DL объединяет в себе множество состояний. В состояние DL система может перейти из состояний вида $(i, n-k-i)$, где $0 \le i \le n-k$. Интенсивность перехода в состояние DL для любого из этих состояний равна $k(\lambda + c)$.

4.3 Расчет MTTDL с учетом невосстановимых битовых ошибок и процесса проверки контрольных сумм

До перехода к описанию модели с произвольными параметрами n и k для наглядност целесообразно рассмотреть практически важные частные случаи расчета MTTDL со значениями $n-k=1$ и $n-k=2$. Прикладная эффективность этих наборов параметров для использования в локально восстановимых кодах (Locally Repairable Codes, LRC) показана в работе [12]. Схемы состояний системы и переходов между ними для этих случаев приведены на рис. 1 и рис. 2. Алгоритм расчета в целом аналогичен расчетам для классической модели.

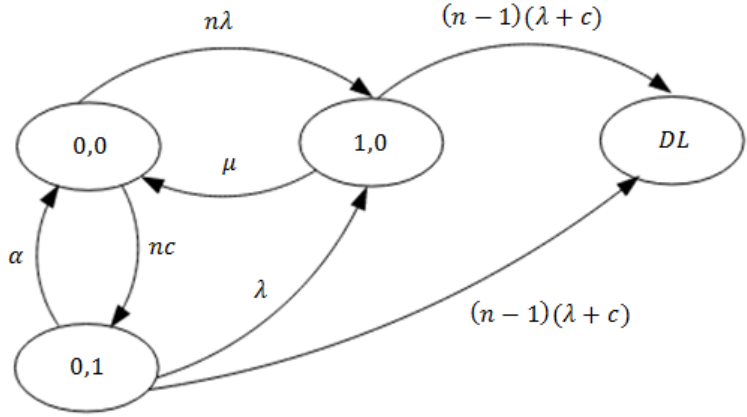


Рис. 1. Интенсивности переходов в Марковской цепи для случая n - k = 1

Пусть $Q_{l,m}$ — вероятность того, что система перейдет из начального состояния (l,m) в состояние DL до того, как окажется в состоянии $(0,0)$, где отсутствуют скрытые или явные сбои. Из определения $Q_{l,m}$ следует, что $Q_{0,0}=0$, а $Q_{DL}=1$.

Значение $Q_{1,0}$ выражается через вероятности $Q_{0,0}$ и Q_{DL} для состояний, в которые можно перейти из состояния $(1,0)$:

$$Q_{1,0}=\frac{1}{\mu+(n-1)(\lambda+c)}\left(\mu Q_{0,0}+(n-1)(\lambda+c)Q_{DL}\right).$$

В этом пункте и далее при проведении расчетов предполагается, что значения параметров λ и c пренебрежимо малы по сравнению со значениями параметров μ и α. Учитывая, что $Q_{0,0}=0$, $Q_{DL}=1$ и отбрасывая пренебрежимо малые члены, можно получить:

$$Q_{1,0}\approx\frac{(n-1)(\lambda+c)}{\mu}.$$

Значение $Q_{0,1}$ выражается следующим образом:

$$Q_{0,1}=\frac{1}{\alpha+\lambda+(n-1)(\lambda+c)}\left(\alpha Q_{0,0}+\lambda Q_{1,0}+(n-1)(\lambda+c)Q_{DL}\right).$$

Можно заметить, что член $\lambda Q_{1,0}\sim\lambda(\lambda+c)$, а член $(n-1)(\lambda+c)Q_{DL}\sim(\lambda+c)$, следовательно, $\lambda Q_{1,0}$ является пренебрежимо малым по отношению $(n-1)(\lambda+c)Q_{DL}\sim(\lambda+c)$:

$$Q_{0,1}\approx\frac{(n-1)(\lambda+c)}{\alpha}.$$

В данной модели возможны две варианта начала цикла – цикл может начаться либо с дискового сбоя, т.е. из состояния $(1,0)$, или с невосстановимой ошибки чтения, т.е. из состояния $(0,1)$.

Ожидаемое количество циклов до потери данных в случае начала из состояния $(1,0)$ будет равно

$$n_{e,(1,0)} = \frac{1}{Q_{1,0}} \approx \frac{\mu}{(n-1)(\lambda + c)}.$$

Ожидаемое количество циклов до потери данных в случае начала из состояния $(0,1)$ будет равно

$$n_{e,(0,1)} = \frac{1}{Q_{0,1}} \approx \frac{\alpha}{(n-1)(\lambda + c)}.$$

Оценка среднего времени цикла выполняется следующим образом. Пусть $T_{l,m}$ ожидаемое время, которое пройдет прежде, чем система перейдет из состояния (l, m) в состояние $(0,0)$ или в состояние DL. По определению $T_{0,0} = 0, T_{DL} = 0$.

Значения $T_{l,m}$ для состояний системы вычисляются как

$$T_{1,0} = \frac{1}{\mu + (n-1)(\lambda + c)}\left(\mu T_{0,0} + (n-1)(\lambda + c)T_{DL}\right) + \frac{1}{\mu + (n-1)(\lambda + c)};$$

$$T_{1,0} = \frac{1}{\mu + (n-1)(\lambda + c)};$$

$$T_{1,0} \approx \frac{1}{\mu};$$

$$T_{0,1} = \frac{1}{\alpha + \lambda + (n-1)(\lambda + c)}\left(\alpha T_{0,0} + \lambda T_{1,0} + (n-1)(\lambda + c)T_{DL}\right) + \frac{1}{\alpha + \lambda + (n-1)(\lambda + c)};$$

$$T_{0,1} = \frac{1}{\alpha + \lambda + (n-1)(\lambda + c)}\frac{\lambda}{\mu} + \frac{1}{\alpha + \lambda + (n-1)(\lambda + c)};$$

$$T_{0,1} \approx \frac{1}{\alpha + \lambda + (n-1)(\lambda + c)} \approx \frac{1}{\alpha}.$$

Пусть $t_{0,0\to1,0}$ и $t_{0,0\to0,1}$ — средние времена перехода из состояния $(0,0)$ в состояния $(1,0)$ и $(0,1)$ соответственно.

Для средних времен цикла $t_{e,(1,0)}$ и $t_{e,(0,1)}$ при начале из состояний $(1,0)$ и $(0,1)$ выполняются соотношения:

$$t_{e,(1,0)} = t_{0,0\to1,0} + T_{1,0};$$

$$t_{e,(0,1)} = t_{0,0\to0,1} + T_{0,1};$$

$$t_{e,(1,0)} \approx \frac{1}{n\lambda} + \frac{1}{\mu};$$

$$t_{e,(0,1)} \approx \frac{1}{n\mathrm{c}} + \frac{1}{\alpha}.$$

Учитывая, что значения $\frac{1}{\mu}$ и $\frac{1}{\alpha}$ пренебрежимо малы по сравнению с $\frac{1}{(\lambda+c)}$, можно получить

$$t_{e,(1,0)} \approx \frac{1}{n\lambda}$$
$$t_{e,(0,1)} \approx \frac{1}{nc}.$$

Пусть $p_{e,(1,0)} = \frac{\lambda}{\lambda+c}$ и $p_{e,(0,1)} = \frac{c}{\lambda+c}$ вероятности начать цикл из состояний (1,0) и (0,1) соответственно.

Из вышеприведенных соотношений видно, что оценка MTTDL хранилища для новой модели равна наименьшему значению из ожидаемых времен до потери данных в случае начала из одного из состояний $(1,0)$ или $(0,1)$:

$$MTTDL_{n-k=1} = p_{e,(1,0)} n_{e,(1,0)} t_{e,(1,0)} + p_{e,(0,1)} n_{e,(0,1)} t_{e,(0,1)} \approx \frac{\mu}{n(n-1)(\lambda+c)^2}, \frac{\alpha}{n(n-1)(\lambda+c)^2}.$$

Далее рассматривается случай $n-k=2$.

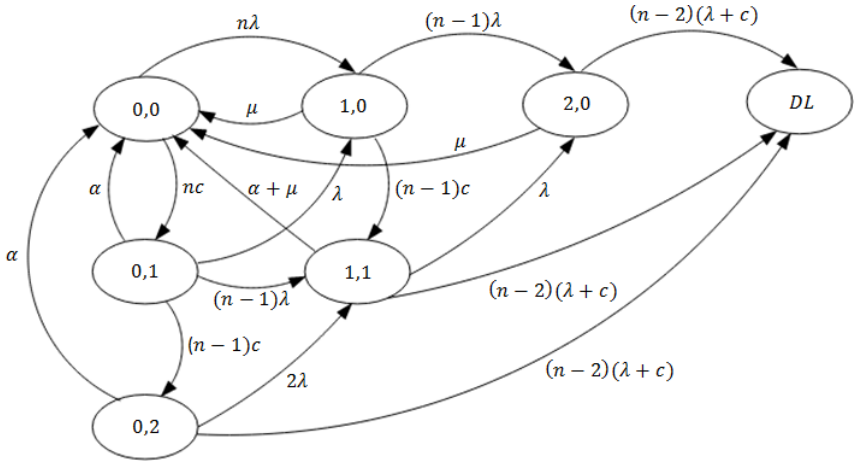


Рис.2. Интенсивности переходов в Марковской цепи для случая $n-k=2$

Значения $Q_{l,m}$ вычисляются, начиная с состояний на диагонали $l+m=n-k$.

$$Q_{2,0} = \frac{1}{\mu + (n-2)(\lambda+c)}\left(\mu Q_{0,0} + (n-2)(\lambda+c)Q_{DL}\right);$$

$$Q_{2,0} \approx \frac{(n-2)(\lambda+c)}{\mu};$$

$$Q_{1,1} = \frac{1}{\alpha+\mu+\lambda+(n-2)(\lambda+c)}\left((\alpha+\mu)Q_{0,0} + \lambda Q_{2,0} + (n-2)(\lambda+c)Q_{DL}\right);$$

$$Q_{1,1} \approx \frac{(n-2)(\lambda+c)}{\alpha+\mu};$$

$$Q_{0,2} = \frac{1}{\alpha+2\lambda+(n-2)(\lambda+c)}\left(\alpha Q_{0,0} + 2\lambda Q_{1,1} + (n-2)(\lambda+c)Q_{DL}\right);$$

$$Q_{0,2} \approx \frac{(n-2)(\lambda+c)}{\alpha}.$$

Далее рассматриваются состояния на диагонали $l+m=n-k-1$:

$$Q_{1,0} = \frac{1}{\mu+(n-1)\lambda+(n-1)c}\left((n-1)\lambda Q_{2,0} + (n-1)cQ_{1,1} + \mu Q_{0,0}\right);$$

$$Q_{1,0} \approx \frac{n-1}{\mu}\left(\lambda\frac{(n-2)(\lambda+c)}{\mu} + c\frac{(n-2)(\lambda+c)}{\alpha+\mu}\right);$$

$$Q_{1,0} \approx \frac{(n-1)(n-2)(\lambda+c)}{\mu}\left(\frac{\lambda}{\mu} + \frac{c}{\alpha+\mu}\right);$$

$$Q_{0,1} = \frac{1}{\alpha+n\lambda+(n-1)c}\left(\lambda Q_{1,0} + (n-1)\lambda Q_{1,1} + (n-1)cQ_{0,2} + \alpha Q_{0,0}\right).$$

Учитывается пренебрежимая малость $\lambda Q_{1,0}$ по сравнению с $\lambda Q_{1,1}$ и $cQ_{0,2}$:

$$Q_{0,1} \approx \frac{1}{\alpha}\left((n-1)\lambda Q_{1,1} + (n-1)cQ_{0,2}\right);$$

$$Q_{0,1} \approx \frac{1}{\alpha}\left((n-1)\lambda\frac{(n-2)(\lambda+c)}{\alpha+\mu} + (n-1)c\frac{(n-2)(\lambda+c)}{\alpha}\right);$$

$$Q_{0,1} \approx \frac{(n-1)(n-2)(\lambda+c)}{\alpha}\left(\frac{\lambda}{\alpha+\mu} + \frac{c}{\alpha}\right).$$

Вычисляется ожидаемое количество циклов до потери данных в случае начальных состояний $(1,0)$ и $(0,1)$:

$$n_{e,(1,0)} = \frac{1}{Q_{1,0}} \approx \frac{\mu}{(n-1)(n-2)(\lambda+c)}\left(\frac{\lambda}{\mu} + \frac{c}{\alpha+\mu}\right)^{-1};$$

$$n_{e,(0,1)} = \frac{1}{Q_{0,1}} \approx \frac{\alpha}{(n-1)(n-2)(\lambda+c)}\left(\frac{\lambda}{\alpha+\mu} + \frac{c}{\alpha}\right)^{-1}.$$

Оцениваются средние времена цикла для начальных состояний $(1,0)$ и $(0,1)$. Значения $T_{l,m}$ вычисляются в той же последовательности, что и соответствующие значения $Q_{l,m}$:

$$T_{2,0} = \frac{1}{\mu+(n-2)(\lambda+c)};$$

$$T_{2,0} \approx \frac{1}{\mu};$$

$$T_{1,1} = \frac{1}{\alpha+\mu+\lambda+(n-2)(\lambda+c)}\lambda T_{2,0} + \frac{1}{\alpha+\mu+\lambda+(n-2)(\lambda+c)};$$
$$T_{1,1} \approx \frac{1}{\alpha+\mu};$$
$$T_{0,2} = \frac{1}{\alpha+2\lambda+(n-2)(\lambda+c)} \cdot 2\lambda T_{1,1} + \frac{1}{\alpha+2\lambda+(n-2)(\lambda+c)};$$
$$T_{0,2} \approx \frac{1}{\alpha};$$
$$T_{1,0} = \frac{1}{\mu+(n-1)\lambda+(n-1)c}\left((n-1)\lambda T_{2,0} + (n-1)cT_{1,1}\right) + \frac{1}{\mu+(n-1)\lambda+(n-1)c};$$
$$T_{1,0} \approx \frac{1}{\mu};$$
$$T_{0,1} = \frac{1}{\alpha+n\lambda+(n-1)c}\left(\lambda T_{1,0} + (n-1)\lambda T_{1,1} + (n-1)cT_{0,2}\right) + \frac{1}{\alpha+n\lambda+(n-1)c};$$
$$T_{0,1} \approx \frac{1}{\alpha}.$$

Для средних времен цикла $t_{e,(1,0)}$ и $t_{e,(0,1)}$ для начальных состояний $(1,0)$ и $(0,1)$ соотношения принимают вид:

$$t_{e,(1,0)} = t_{0,0\to1,0} + T_{1,0};$$
$$t_{e,(0,1)} = t_{0,0\to0,1} + T_{0,1};$$
$$t_{e,(1,0)} \approx \frac{1}{n\lambda};$$
$$t_{e,(0,1)} \approx \frac{1}{nc}.$$

MTTDL хранилища для данной модели равен

$$MTTDL_{n-k=2} = p_{e,(1,0)}n_{e,(1,0)}t_{e,(1,0)} + p_{e,(0,1)}n_{e,(0,1)}t_{e,(0,1)} \approx \frac{\mu\left(\frac{\lambda}{\mu}+\frac{c}{\alpha+\mu}\right)^{-1}}{n(n-1)(n-2)(\lambda+c)^2} + \frac{\alpha\left(\frac{\lambda}{\alpha+\mu}+\frac{c}{\alpha}\right)^{-1}}{n(n-1)(n-2)(\lambda+c)^2}.$$

4.4 Обобщение расширенной модели на произвольные параметры $(\mathbf{n}, \mathbf{k})$

Соотношения, полученные для частных случаев $n-k=1$ и $n-k=2$ можно обобщить для произвольных параметров (n,k).

Схема вычисления значений $Q_{l,m}$ и $T_{l,m}$, изображенная на рис. 3, предполагает последовательное вычисление значений вдоль диагоналей $l+m=const$,

начиная с диагонали, соответствующей состояниям (l, m), для которых $l + m = n - k$, при движении сверху вниз вдоль диагоналей.

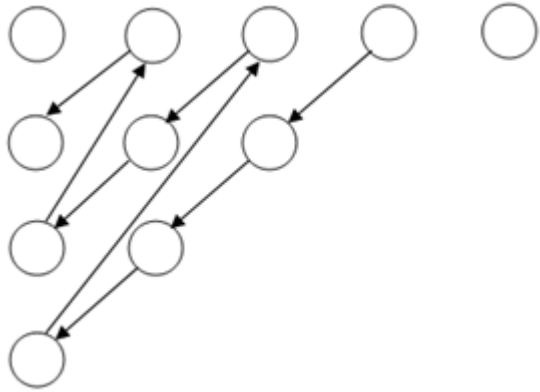


Рис. 3. Схема вычисления значений $\boldsymbol{Q_{l,m}}$ *и* $\boldsymbol{T_{l,m}}$

Средние времена цикла $t_{e,(1,0)}$ и $t_{e,(0,1)}$ для модели с произвольными параметрами (n, k) вычисляются следующим образом.

$$T_{l,m} = \begin{cases} \dfrac{(n-l-m)\lambda T_{l+1,m} + (n-l-m)cT_{l,m+1} + 1}{\mu + m\lambda + (n-l-m)(\lambda + c)}, l \neq 0, m = 0 \\ \dfrac{m\lambda T_{l+1,m-1} + (n-l-m)\lambda T_{l+1,m} + (n-l-m)cT_{l,m+1} + 1}{\alpha + \mu + m\lambda + (n-l-m)(\lambda + c)}, l \neq 0, m \neq 0 \\ \dfrac{m\lambda T_{l+1,m-1} + (n-l-m)\lambda T_{l+1,m} + 1}{\alpha + m\lambda + (n-l-m)(\lambda + c)}, l = 0, m \neq 0 \end{cases}$$

Отбрасывая пренебрежимо малые члены, можно получить:

$$T_{l,m} \approx \begin{cases} \dfrac{1}{\mu}, l \neq 0, m = 0 \\ \dfrac{1}{\alpha + \mu}, l \neq 0, m \neq 0 \\ \dfrac{1}{\alpha}, l = 0, m \neq 0 \end{cases}$$

Из выведенного соотношения видно, что значение $T_{l,m}$ зависит только от положения состояния (l, m) внутри диагонали $l + m = const$, но не зависит от диагонали.

Таким образом, средние времена цикла равны:

$$t_{e,(1,0)} \approx \frac{1}{n\lambda}, \qquad t_{e,(0,1)} \approx \frac{1}{nc}.$$

Вычисление $Q_{l,m}$ в произвольном случае производится несколько сложнее.

$$Q_{l,m} = \begin{cases} \dfrac{(n-l-m)\lambda Q_{l+1,m} + (n-l-m)cQ_{l,m+1}}{\mu + m\lambda + (n-l-m)(\lambda + c)}, l \neq 0, m = 0, \\ \dfrac{m\lambda Q_{l+1,m-1} + (n-l-m)\lambda Q_{l+1,m} + (n-l-m)cQ_{l,m+1}}{\alpha + \mu + m\lambda + (n-l-m)(\lambda + c)}, l \neq 0, m \neq 0, \\ \dfrac{m\lambda Q_{l+1,m-1} + (n-l-m)\lambda Q_{l+1,m} + (n-l-m)cQ_{l,m+1}}{\alpha + m\lambda + (n-l-m)(\lambda + c)}, l = 0, m \neq 0. \end{cases}$$

Если учесть, что $\lambda Q_{l+1,m-1}$ пренебрежимо мало по сравнению с $\lambda Q_{l+1,m}$, а $cQ_{l,m+1}$ — значения $Q_{l,m}$ в рамках одной диагонали $l + m = const$ сравнимы между собой по величине, в то время как значения $Q_{l,m}$ на внутренних диагоналях пренебрежимо малы по сравнению со значениями $Q_{l,m}$ на диагоналях, внешних по отношению к этим диагоналям, то выполняются условия:

$$Q_{l_1,m_1} \ll Q_{l_2,m_2}, \text{где } l_1 + m_1 < l_2 + m_2;$$

$$Q_{l,m} \approx \begin{cases} \dfrac{(n-l-m)\lambda Q_{l+1,m} + (n-l-m)cQ_{l,m+1}}{\mu}, l \neq 0, m = 0, \\ \dfrac{(n-l-m)\lambda Q_{l+1,m} + (n-l-m)cQ_{l,m+1}}{\alpha + \mu}, l \neq 0, m \neq 0, \\ \dfrac{(n-l-m)\lambda Q_{l+1,m} + (n-l-m)cQ_{l,m+1}}{\alpha}, l = 0, m \neq 0. \end{cases}$$

Пусть «простым» путем из состояния Q_{l_1,m_1} в состояние Q_{l_2,m_2} называется такая последовательность переходов между состояниями системы, в которой нет событий восстановления данных, а также нет переходов между состояниями, располагающимися на одной диагонали. Пути, в которых есть переходы между состояниями на одной диагонали, можно не учитывать в силу их маловероятности по отношению к «простым» путям.

Можно заметить, что значение $Q_{l,m}$ в состоянии (l, m) определяется как сумма слагаемых S_i, вычисляемых вдоль всех «простых» путей из состояния $Q_{l,m}$ в состояние Q_{DL}. Число таких «простых» путей для состояния $Q_{l,m}$ равно $2^{n-k-(l+m)}$, т.к. длина всех «простых» путей из состояния $Q_{l,m}$ в состояние Q_{DL} фиксирована и равна $n - k - (l + m)$, и в каждом промежуточном состоянии есть два варианта выбора дальнейшего направления пути – вверх или вниз.

Из рекуррентных соотношений для $Q_{l,m}$ следует, что значение S_i, соответствующее некоторому фиксированному «простому» пути, определяется как произведение интенсивностей переходов между состояниями вдоль данного пути, поделенное на произведение интенсивностей восстановления данных в каждом из состояний пути, кроме состояния DL.

Определим вспомогательные функции $I(l, m)$ и $P(l, m)$:

$$I(l,m) = \begin{cases} \mu, l \neq 0, m = 0 \\ \alpha + \mu, l \neq 0, m \neq 0 \\ \alpha, l = 0, m \neq 0 \end{cases}$$

$$P(l,m) = \prod_{i=0}^{(n-k)-(l+m)} (n - (l+m) - i)$$

Таким образом, значение $Q_{l,m}$ для произвольного состояния (l,m) определяется следующим выражением:

$$Q_{l,m} \approx P(l,m)\left(\sum_{\beta_1,\dots,\beta_{(n-k)-(l+m)}=0}^{1} \frac{(\lambda + \mathrm{c})\prod_{i=1}^{(n-k)-(l+m)} \lambda^{\beta_i} c^{1-\beta_i}}{I(l,m)\prod_{i=1}^{(n-k)-(l+m)} I(l+\sum_{j=1}^{i}\beta_i, m+\sum_{j=1}^{i}(1-\beta_i))}\right)$$

$$Q_{l,m} \approx \frac{P(l,m)}{I(l,m)}\left(\sum_{\beta_1,\dots,\beta_{(n-k)-(l+m)}=0}^{1} \prod_{i=1}^{(n-k)-(l+m)} \frac{(\lambda + \mathrm{c})\lambda^{\beta_i} c^{1-\beta_i}}{I(l+\sum_{j=1}^{i}\beta_i, m+\sum_{j=1}^{i}(1-\beta_i))}\right)$$

Ожидаемое количество циклов $n_{e,(1,0)}$ и $n_{e,(1,0)}$ определяется следующими выражениями

$$Q_{1,0} \approx \frac{P(1,0)}{I(1,0)}\left(\sum_{\beta_1,\dots,\beta_{(n-k)-1}=0}^{1} \prod_{i=1}^{(n-k)-1} \frac{(\lambda + \mathrm{c})\lambda^{\beta_i} c^{1-\beta_i}}{I(1+\sum_{j=1}^{i}\beta_i, \sum_{j=1}^{i}(1-\beta_i))}\right)$$

$$Q_{0,1} \approx \frac{P(0,1)}{I(0,1)}\left(\sum_{\beta_1,\dots,\beta_{(n-k)-1}=0}^{1} \prod_{i=1}^{(n-k)-1} \frac{(\lambda + \mathrm{c})\lambda^{\beta_i} c^{1-\beta_i}}{I(\sum_{j=1}^{i}\beta_i, 1+\sum_{j=1}^{i}(1-\beta_i))}\right)$$

$$n_{e,(1,0)} = \frac{1}{Q_{1,0}} \approx \left(\sum_{\beta_1,\dots,\beta_{(n-k)-1}=0}^{1} \frac{P(1,0)}{I(1,0)} \prod_{i=1}^{(n-k)-1} \frac{(\lambda + \mathrm{c})\lambda^{\beta_i} c^{1-\beta_i}}{I(1+\sum_{j=1}^{i}\beta_i, \sum_{j=1}^{i}(1-\beta_i))}\right)^{-1}$$

$$n_{e,(0,1)} = \frac{1}{Q_{0,1}} \approx \left(\sum_{\beta_1,\dots,\beta_{(n-k)-1}=0}^{1} \frac{P(0,1)}{I(0,1)} \prod_{i=1}^{(n-k)-1} \frac{(\lambda + \mathrm{c})\lambda^{\beta_i} c^{1-\beta_i}}{I(\sum_{j=1}^{i}\beta_i, 1+\sum_{j=1}^{i}(1-\beta_i))}\right)^{-1}$$

Таким образом, аналитическое выражение для MTTDL системы для произвольных параметров (n,k):

$$MTTDL_{n,k} \approx p_{e,(1,0)} n_{e,(1,0)} t_{e,(1,0)} + p_{e,(0,1)} n_{e,(0,1)} t_{e,(0,1)}$$

$$= \frac{1}{n(\lambda + c)^2\left(\sum_{\beta_1,\dots,\beta_{(n-k)-1}=0}^{1} \frac{P(1,0)}{I(1,0)} \prod_{i=1}^{(n-k)-1} \frac{(\lambda + \mathrm{c})\lambda^{\beta_i} c^{1-\beta_i}}{I(1+\sum_{j=1}^{i}\beta_i, \sum_{j=1}^{i}(1-\beta_i))}\right)}$$

$$+ \frac{1}{n(\lambda + c)^2\left(\sum_{\beta_1,\dots,\beta_{(n-k)-1}=0}^{1} \frac{P(0,1)}{I(0,1)} \prod_{i=1}^{(n-k)-1} \frac{(\lambda + \mathrm{c})\lambda^{\beta_i} c^{1-\beta_i}}{I(\sum_{j=1}^{i}\beta_i, 1+\sum_{j=1}^{i}(1-\beta_i))}\right)}$$

5. Результаты расчётов по разработанной модели

В этом разделе раскрыты характеристики стилей, используемых в данном документе.

5.1 Сравнительный анализ классической и расширенной моделей

Сравнение результатов расчётов по новой разработанной модели с итогами вычислений по классической модели приведено на рис. 4. В новой модели интенсивность битовых ошибок определялась средней интенсивностью доступа к дискам в хранилище. По графику видно, что результаты, полученные в классической модели, учитывающей только явные дисковые сбои, отличаются от уточненных результатов, полученных в рамках предложенной модели. Наблюдаемая разница является значительной и подтверждает практическую значимость предложенной модели. Полученные результаты соответствуют приведенным в работе [11] выводам о необходимости учёта скрытых ошибок и проведении проверок контрольных сумм.

Важно отметить, что MTTDL хранилища в значительной степени зависит от средней интенсивности использования дисков, увеличиваясь с уменьшением интенсивности использования и приближаясь к более простой модели, не учитывающей битовые ошибки. Этот результат подтверждает существенную зависимость частоты невосстановимых ошибок чтения от интенсивности использования диска. Таким образом интенсивность использования диска может служить эмпирическим критерием оценки частоты возникновения скрытых дисковых сбоев.

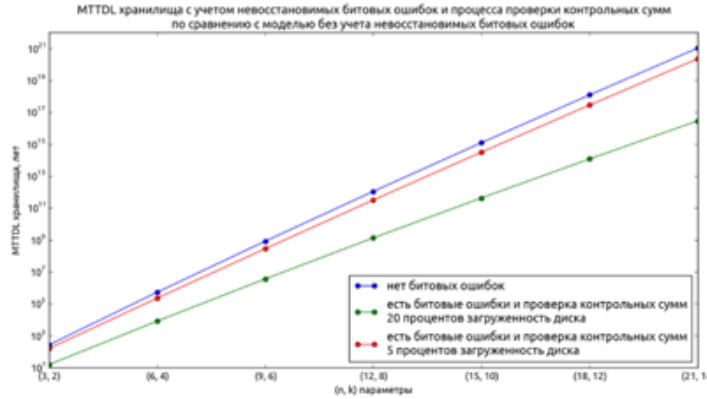


Рис. 4. Сравнение результатов, полученных в модели, учитывающей невосстановимые битовые ошибки и процесс проверки контрольных сумм, и в более простой модели

Из графика зависимости MTTDL хранилища данных от среднего времени полной проверки данных в хранилище видно, что MTTDL сначала резко убывает на начальном участке, а потом становится практически постоянной и не зависит от интенсивности проверки данных, приближаясь к значению MTTDL хранилища, в котором присутствуют скрытые битовые ошибки, но отсутствуют процессы проверки контрольных сумм.

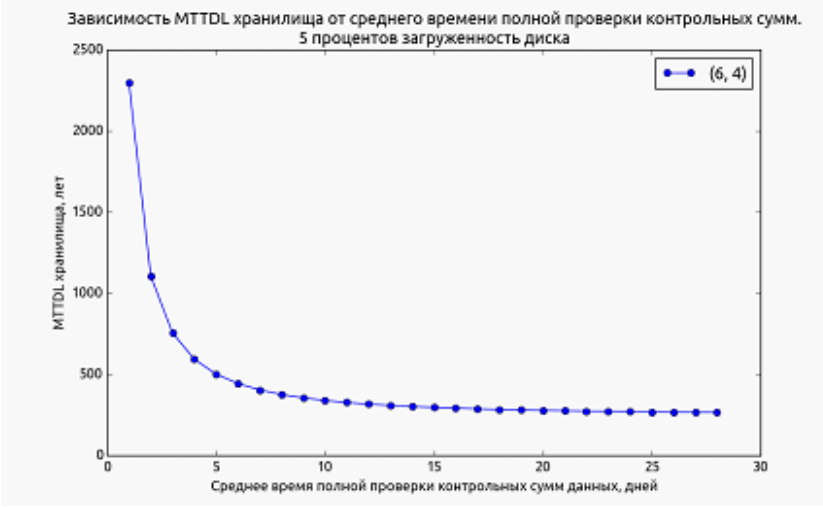


Рис. 5. Зависимость MTTDL хранилища данных от среднего периода полной проверки данных в хранилище

Зависимость между MTTDL хранилища данных и интенсивностью проверки контрольных сумм позволяет находить баланс между надежностью хранения данных и экономическими затратами на непрерывный процесс проверки данных.

5. Выводы

В настоящей работе была представлена новая математическая модель надёжности распределенных хранилищ с новым аналитическим описанием с основе двумерных Марковских цепей. Предложен новый метод расчёта MTTDL хранилища с помощью приближенно аналитических выражений для произвольных параметров (n, k). Наглядно показано, что невозможность учёта битовых ошибок чтения в классических моделях приводит к завышенным оценкам MTTDL. Из этого следует, что скрытые сбои существенно влияют на надежность хранения данных, и этим фактором не следует пренебрегать.

6. Примечания

Статья опубликована в рамках выполнения прикладных научных исследований при финансовой поддержке Министерства образования и науки Российской Федерации. Соглашения о предоставлении субсидий № 14.579.21.0010. Уникальный идентификатор Соглашения RFMEFI57914X0010.

Литература

[1]. Patterson D. A., Gibson G., and Katz R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID), Proc. of ACM SIGMOD, 1988.

[2]. Reibman A. and Trivedi K. S. A. Transient Analysis of Cumulative Measures of Markov Model Behavior. Communications in Statistics-Stochastic Models, 1989, vol. 5, pp. 683–710.

[3]. Schultz M., Gibson G., Katz R., and Patterson D. How Reliable is a RAID? Proceedings of CompCon, 1989, pp. 118–123.

[4]. Malhotra M. and Trivedi K. S. Reliability Analysis of Redundant Arrays of Inexpensive Disks. Journal of Parallel and Distributed Computing — Special issue on parallel I/O systems, 1993, vol.17, – no. 1-2., pp. 146-151.

[5]. Greenan K. M., Plank J. S. and Wylie J. J. Mean Time To Meaningless: MTTDL, Markov models, and Storage System Reliability, Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems, 2010, pp. 1–5.

[6]. Karmakar P. and Gopinath K. Are Markov Models Effective for Storage Reliability Modelling? arXiv:1503.07931v1, 2015.

[7]. Li Y., Lee P. P. and Lui J. Stochastic analysis on raid reliability for solid-state drives, IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS). IEEE, 2013, pp. 71–80. (http://arxiv.org/pdf/1304.1863.pdf)

[8]. Mann S. E., Anderson M. and Rychlik M. On the Reliability of RAID Systems: An Argument for More Check Drives, arXiv:1202.4423v1, 2012.

[9]. Pâris J.-F., Schwarz T., Amer A. and Long D. D. E. Improving Disk Array Reliability Through Expedited Scrubbing, Proceedings of the 5th IEEE International Conference on Networking, Architecture, and Storage, 2010, pp. 119–125.

[10]. Xin Q., Miller E. L., Schwarz T. J., Long D. D. E., Brandt S. A. and Litwin W. .Reliability mechanisms for very large storage systems, Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003, pp. 146–156.

[11]. Elerath J.G. and Pecht M. Enhanced Reliability Modeling of RAID Storage Systems, 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007, pp. 175–184.

[12]. Ivanichkina L. and Neporada A. Mathematical methods and models of improving data storage reliability including those based on finite field theory, Contemporary Engineering Sciences, 2014, vol. 7, no. 28, 1589-1602 http://dx.doi.org/10.12988/ces.2014.411236.

The Reliability Model of a Distributed Data Storage in Case of Explicit and Latent Disk Faults

[1]L. Ivanichkina <ivanov@ispras.ru>
[2]A. Neporada <Andrew.Neporada@acronis.com>
[1]OOO Proekt IKS, Altufievskoe sh, 44, Moscow, 127566, Russian Federation
[2]OOO Acronis, Altufievskoe sh, 44, Moscow, 127566, Russian Federation

Abstract. This work examines the approach to the estimation of the data storage reliability that accounts for both explicit disk faults and latent bit errors as well as procedures to detect them. A new analytical math model of the failure and recovery events in the distributed data storage is proposed to calculate reliability. The model describes dynamics of the data loss and recovery based on Markov chains corresponding to the different schemes of redundant encoding. Advantages of the developed model as compared to classical models for traditional RAIDs are covered. Influence of latent HDD errors is considered, while other bit faults occurring in the other hardware components of the machine are omitted. Reliability is estimated according to new analytical formulas for calculation of the mean time to failure, at which data loss exceeds the recoverability threshold defined by the redundant encoding parameters. New analytical dependencies between the storage average lifetime until the data loss and the mean time for complete verification of the storage data are given.

Keywords: Mean time to failure (MTTF), Markov chains, redundant encoding, Huygens' gambler's ruin problem, distributed data storage, scrubbing procedure, checksums, MTTDL of distributed data storage, disk faults, irrecoverable bit errors, latent sector errors.

DOI: 10.15514/ISPRAS-2015-27(6)-16

For citation: Ivanichkina L., Neporada A. The Reliability Model of a Distributed Data Storage in Case of Explicit and Latent Disk Faults. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 253-274 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-16

References

[1]. Patterson D. A., Gibson G., and Katz R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID), Proc. of ACM SIGMOD, 1988.
[2]. Reibman A. and Trivedi K. S. Transient Analysis of Cumulative Measures of Markov Model Behavior. Communications in Statistics-Stochastic Models, 1989, vol. 5, pp. 683–710.
[3]. Schultz M., Gibson G., Katz R., and Patterson D. How Reliable is a RAID? Proceedings of CompCon, 1989, pp. 118–123.

[4]. Malhotra M. and Trivedi K. S. Reliability Analysis of Redundant Arrays of Inexpensive Disks. Journal of Parallel and Distributed Computing — Special issue on parallel I/O systems, 1993, vol.17, – no. 1-2., pp. 146-151.

[5]. Greenan K. M., Plank J. S. and Wylie J. J. Mean Time To Meaningless: MTTDL, Markov models, and Storage System Reliability, Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems, 2010, pp. 1–5.

[6]. Karmakar P. and Gopinath K. Are Markov Models Effective for Storage Reliability Modelling? arXiv:1503.07931v1, 2015.

[7]. Li Y., Lee P. P. and Lui J. Stochastic analysis on raid reliability for solid-state drives, IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS). IEEE, 2013, pp. 71–80. (http://arxiv.org/pdf/1304.1863.pdf)

[8]. Mann S. E., Anderson M. and Rychlik M. On the Reliability of RAID Systems:An Argument for More Check Drives, arXiv:1202.4423v1, 2012.

[9]. Pâris J.-F., Schwarz T., Amer A. and Long D. D. E. Improving Disk Array Reliability Through Expedited Scrubbing, Proceedings of the 5th IEEE International Conference on Networking, Architecture, and Storage, 2010, pp. 119–125.

[10]. Xin Q., Miller E. L., Schwarz T. J., Long D. D. E., Brandt S. A. and Litwin W. .Reliability mechanisms for very large storage systems, Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003, pp. 146–156.

[11]. Elerath J.G. and Pecht M. Enhanced Reliability Modeling of RAID Storage Systems, 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007, pp. 175–184.

[12]. Ivanichkina L. and Neporada A. Mathematical methods and models of improving data storage reliability including those based on finite field theory, Contemporary Engineering Sciences, 2014, vol. 7, no. 28, 1589-1602 http://dx.doi.org/10.12988/ces.2014.411236.

Модель проблемно-ориентированной облачной вычислительной среды[1]

Г.И. Радченко <gleb.radchenko@susu.ru>
Южно-Уральский государственный университет,
454080, Россия, г. Челябинск, проспект Ленина, д. 76.

Аннотация. Для эффективного использования ресурсов высокопроизводительных вычислительных систем при реализации методов численного исследования физических, биологических, социальных и др. явлений могут быть использованы подходы предоставления распределенных проблемно-ориентированных вычислительных сред. Они обеспечивают пользователям прозрачный доступ к решению конкретных классов прикладных задач на базе доступных вычислительных ресурсов. Для повышения эффективности таких сред необходимо применение проблемно-ориентированных методов планирования вычислительных задач, использующих информацию о предметной области для прогнозирования вычислительных характеристик задач при планировании и распределении заданий. В статье представлены модели предметной области и проблемно-ориентированной облачной вычислительной среды, ориентированные на поддержку разработки новых проблемно-ориентированных алгоритмов планирования при выполнении расчетов в конкретных предметных областях на базе облачных вычислительных систем.

Ключевые слова: облачные вычисления; модель вычислительной среды; распределенные вычислительные системы; проблемно-ориентированная вычислительная среда

Ключевые слова: модели данных, объектная модель, ODMG 3.0, SQL, разыменование объектных ссылок, размещение объектов.

DOI: 10.15514/ISPRAS-2015-27(6)-17

Для цитирования: Радченко Г.И. Модель проблемно-ориентированной облачной вычислительной среды. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 275-284. DOI: 10.15514/ISPRAS-2015-27(6)-17.

[1] Работа выполнена при поддержке Российского фонда фундаментальных исследований, грант № 15-29-07959 офи-м.

1. Введение

Использование методов суперкомпьютерного моделирования и интеллектуального анализа данных обеспечивает получение качественно новых результатов во всех отраслях знаний, позволяя проводить численные исследования физических, биологических, социальных и др. явлений, предоставляя реальную альтернативу дорогостоящим (или невозможным) экспериментам [1].

Этот тренд привел к появлению особой научной дисциплины, названной «Computational Science» («Вычислительная наука»). Вычислительная наука – это быстро развивающаяся мульти-дисциплинарная научная дисциплина, использующая передовые вычислительные методы для решения сложных задач, объединяющая в себе методы, алгоритмы и программное обеспечение для компьютерного моделирования, разработанные для решения сложных научных и инженерных задач; а также аспекты фундаментальной информатики и информационных технологий, обеспечивающие развитие аппаратных, программных, сетевых компонентов и СУБД, необходимых для решения таких вычислительно-сложных задач [2]. Примерами разделов вычислительной науки являются такие дисциплины как биоинформатика, вычислительная химия, вычислительная гидродинамика, вычислительная инженерия и др.

Процесс решения задач такого типа с использованием суперкомпьютерных ресурсов для рядового пользователя может быть сопряжен с определенными трудностями в связи с необходимостью специфических знаний, умений и навыков в области высокопроизводительных вычислений. Для эффективного использования ресурсов высокопроизводительных вычислительных систем и организации прозрачного доступа к распределенным вычислительным ресурсам могут быть использованы подходы предоставления распределенных проблемно-ориентированных вычислительных сред (ПВС). В таком случае, вместо прямого доступа к аппаратным интерфейсам удаленных вычислительных устройств, пользователю предоставляется прозрачный веб-интерфейс распределенной проблемно-ориентированной вычислительной среды, которая берет на себя задачи по решению конкретных классов прикладных задач на базе доступных вычислительных ресурсов, включая декомпозицию задания в иерархию вычислительных задач; поиск и выделение необходимых вычислительных ресурсов; мониторинг хода решения задач; передачу результатов решения задач пользователю [3–5].

Современным задачам вычислительной науки характерны высокие требования к предоставляемым вычислительным ресурсам, а также сложная вычислительная структура заданий, которую можно описать в виде потока работ [6]. Также, для задач такого рода характерны многовариантные расчеты, когда вычислительное задание запускается многократно с различными вариациями входных параметров [7]. Приложения такого рода составляют большой процент загрузки современных суперкомпьютерных и

распределенных вычислительных систем, что влечет необходимость в создании методов и алгоритмов эффективного распределения ресурсов таких систем для оптимизации решения таких задач. Одним из подходов, ориентированных на повышение эффективности таких сред, является применение проблемно-ориентированных методов планирования вычислительных задач [8].

В рамках данной статьи будет представлена модель проблемно-ориентированной облачной вычислительной среды, которая бы обеспечила поддержку разработки новых алгоритмов для планирования потоковых приложений при выполнении расчетов в конкретных предметных областях на базе облачных вычислительных систем.

Статья организована следующим образом. В разделе 1 даются основные определения и вводится понятие предметной области. Раздел 2 посвящен обзору модели проблемно-ориентированной облачной вычислительной системы. В заключении изложены выводы и направления дальнейшего развития работы.

2. Модель предметной области

Особенностью распределенных проблемно-ориентированных вычислительных сред является то, что они обеспечивают решение заданий в рамках конкретной предметной области. Такие вычислительные задания часто могут быть представлены в виде ориентированного ациклического графа, узлами которого являются взаимосвязанные вычислительные задачи, а дуги соответствуют потокам данных, передаваемых между отдельными задачами. При этом, в рамках предметной области, набор задач, из которых строятся задания, является предопределенным. Задачи могут быть сгруппированы в конечное множество классов. Класс задач представляет собой множество задач, имеющих одну и ту же семантику, а также одинаковые наборы входных параметров и выходных данных.

С одной стороны, это накладывает ограничения на классы задач, которые могут быть решены в рамках ПВС. С другой стороны, такое ограничение позволяет использовать информацию о предметной области для прогнозирования вычислительных характеристик задач при планировании и распределении заданий, увеличивая эффективность использования доступных вычислительных ресурсов.

Дадим определение предметной области посредством на основе понятия информационного объекта.

Пусть *множество базовых информационных объектов* $\mathcal{B}$ – это счетное множество объектов, которые в рамках соответствующей предметной области рассматриваются как атомарные (т.е. не имеющие составных частей), и разбиваются на k множеств *непересекающихся базовых классов* B_i:

$$\mathcal{B} = \bigcup_{i=1}^{k} B_i, (\forall i \neq j) B_i \cap B_j = \emptyset.$$

Будем считать, что мы можем определить размер любого базового информационного объекта множества $\mathcal{B}$ (в байтах). Обозначим размер базового информационного объекта b как $|b|$:

$$|b|: b \to \mathbb{N}.$$

Определим *класс информационных объектов* C следующим образом:

1) базовый класс информационных объектов будет являться классом информационных объектов:
$$C = (B), \forall B \in \mathcal{B};$$
2) упорядоченный набор базовых классов информационных объектов будет являться классом информационных объектов:
$$C = (B_1, \dots, B_{\mathrm{m}}),\ (\forall l \in \{1, \dots, m\}) B_j \in \mathcal{B};$$
3) упорядоченный набор ранее определенных классов информационных объектов так же будет являться классом информационных объектов:
$$C = (C_1, \dots, C_u).$$

Таким образом, информационный объект I класса $C = (C_1, \dots, C_u)$ представляет собой упорядоченный набор конечного числа информационных объектов

$$I = (I_1, \dots, I_u),$$

где I_i – это:

1) (b), где $b \in B$, если $C_i = (B)$;
2) $(b_1, \dots, b_m)$, где $(\forall l \in \{1, \dots, m\}) b_l \in B_l$, если $C_i = (B_1, \dots, B_{\mathrm{m}})$;
3) $\left(I_{i_1}, \dots, I_{i_y}\right)$, где $(\forall l \in \{1, \dots, y\}) I_l$ – это информационный объект класса C_{i_l}, если $C_i = \left(C_{i_1}, \dots, C_{i_y}\right)$.

Будем говорить, что базовый информационный объект b *входит в информационный объект* I ($b \in I$), если выполняется одно из следующих условий:

1) $I = (b)$;
2) $I = (b_1, \dots, b_m)$ и $b \in (b_1, \dots, b_2)$;
3) $I = (I_1, \dots, I_n)$ и $\exists I_i | b \in I_i$.

Определим *множество классов информационных объектов* $\mathcal{C}$ как конечное множество всех определенных в рамках предметной области классов информационных объектов:

$$\mathcal{C} = \bigcup_{i=1}^{|\mathcal{C}|} C_i.$$

Определим *набор функций* $\mathcal{F}$ над множеством $\mathcal{C}$:

$$\mathcal{F} = \bigcup_{i=1}^{|\mathcal{F}|} f_i \text{, где } (\forall i \in \{1, \dots, |\mathcal{F}|\}) f_i \colon \mathcal{C}_i^{in} \rightarrow \mathcal{C}_i^{out}$$

Каждая функция $f \in \mathcal{F}$ на вход получает n информационных объектов $\mathcal{I}^{in} = \left(I_1^{in}, \dots, I_n^{in}\right)$ соответствующих классов $\mathcal{C}^{in} = (C_1, \dots, C_n)$. Результатом работы функции являются m новых информационных объектов $\mathcal{I}^{out} = (I_1^{out}, \dots, I_m^{out})$ соответствующих классов $\mathcal{C}^{out} = (C_1', \dots, C_m')$.

Таким образом, определим *предметную область* $\mathfrak{P}$ как упорядоченную тройку:

$$\mathfrak{P} = \langle \mathcal{B}, \mathcal{C}, \mathcal{F} \rangle,$$

где $\mathcal{B}$ - множество базовых информационных объектов, $\mathcal{C}$ - множество классов информационных объектов, $\mathcal{F}$ – множество функций, определенных над $\mathcal{C}$.

3. Модель проблемно-ориентированной облачной вычислительной среды

Определим *проблемно-ориентированную облачную вычислительную среду* как четверку:

$$\mathfrak{C}(\mathfrak{N}, \mathfrak{E}, \mathfrak{M}, \mathfrak{P}),$$

где $\mathfrak{N}$ – множество узлов вычислительной системы; $\mathfrak{E}$ – множество сетевых соединений, связывающих вычислительные узлы; $\mathfrak{M}$ – множество образов виртуальных вычислительных машин, доступных для развертывания на узлы из множества $\mathfrak{N}$, $\mathfrak{P}$ – базовая предметная область.

Вычислительным узлом $\mathfrak{n} \in \mathfrak{N}$ назовем вычислительную систему с общей памятью, представленную тройкой:

$$(P_{\mathfrak{n}}, m_{\mathfrak{n}}, \Pi_{\mathfrak{n}}),$$

где $P_{\mathfrak{n}}$ – это упорядоченное множество вычислительных ядер узла; $m_{\mathfrak{n}}$ – это объем оперативной памяти, доступной на вычислительном узле; $\Pi_{\mathfrak{n}}$ – это вектор характеристик производительности вычислительного узла.

Образом виртуальной машины $\mathfrak{m} \in \mathfrak{M}$ назовем тройку:

$$(P_{\mathfrak{m}}, m_{\mathfrak{m}}, \Pi_{\mathfrak{m}}),$$

где $P_{\mathfrak{n}}$ – это упорядоченное множество вычислительных ядер, выделенных виртуальной машине; $m_{\mathfrak{m}}$ – это объем оперативной памяти, выделенный виртуальной машине; $\Pi_{\mathfrak{m}}$ – это вектор характеристик производительности виртуальной машины.

Определим характеристику производительности вычислительной машины как отображение:

$$\pi \colon \mathfrak{m} \rightarrow \mathbb{Z}_{>0},$$

где $\mathfrak{m}$ – образ это виртуальной машины (либо вычислительный узел) существующий в вычислительной системе $\mathfrak{C}$.

Примерами характеристик производительности могут служить числовые характеристики машины, результаты синтетических тестов (Dhrystone [9], SuperPI [10], LINPACK [11], LAPACK [12] и др.) или результаты тестового выполнения конкретных классов функций с заранее определенными наборами входных данных.

Очевидно, что для качественного прогнозирования параметров выполнения задач на заданных машинах, нам необходимо учитывать максимально возможное количество характеристик производительности, включая такие характеристики как: количество доступных процессоров; частота процессора; скорость обмена данными с жестким диском; характеристика машины по LINPACK и др. Таким образом, определим вектор характеристик производительности виртуальных машин, развернутых в облачной вычислительной системе $\mathfrak{C}$:

$$\Pi = [\pi_1, \pi_2 \dots \pi_r].$$

Каждой машине $\mathfrak{m} \in \mathfrak{M}$ облачной вычислительной системы $\mathfrak{C}$ сопоставим вектор характеристик производительности, отражающий значения производительности вычислительной машины:

$$\Pi: \mathfrak{M} \to \mathbb{Z}^r_{>0}.$$

В дальнейшем будем считать, что в рамках предоставления вычислительных ресурсов, каждой задаче выделяется одна либо несколько виртуальных машин. Прямого доступа к узлам вычислительной системы не обеспечивается.

Особенностью проблемно-ориентированной облачной вычислительной среды является то, что она использует информацию об особенностях классов решаемых задач при планировании и распределении вычислительных ресурсов. Будем требовать, чтобы в рамках проблемно-ориентированной вычислительной среды, для каждого класса задач были определены следующие функции для прогноза процесса выполнения задачи в зависимости от значений входных параметров:

1) функция оценки объема выходных данных при определенных входных параметрах;

2) функция оценки времени выполнения задачи при определенных входных параметрах на машине с указанным вектором характеристик производительности.

Таким образом, для каждой функции $f \in F$ из предметной области $\mathfrak{P}$, выполняющейся в проблемно-ориентированной среде $\mathfrak{C}$, определим следующий набор операторов:

1) *оператор ожидаемого выхода* $\nu(f, \mathcal{I}^{in})$ – это оператор, возвращающий ожидаемый общий размер в байтах всех выходных информационных объектов $\mathcal{I}^{out}$:

$$v(f, \mathcal{I}^{in}) = |\mathcal{I}^{out}| = \sum_{\forall I \in \mathcal{I}^{out}} \sum_{\forall b \in I} |b|.$$

2) *оператор ожидаемого времени выполнения функции* $\tau(f, \mathcal{I}^{in}, \Pi)$, возвращающий оценочное время выполнения (в секундах) функции f при заданном множестве входных информационных объектов $\mathcal{I}^{in}$ на машине, с вектором характеристик производительности Π:

$$\tau: (f, \mathcal{I}^{in}, \Pi) \to \mathbb{N}.$$

Время выполнения функции $f: \mathcal{C}^{in} \to \mathcal{C}^{out}$ на конкретной машине с вектором характеристик производительности Π можно представить в виде оператора, зависящего от вектора входных информационных объектов $\mathcal{I}^{in}$. К сожалению, невозможно оценить время выполнения функции с идеальной точностью, т.к. вычислительная работа подготовки набора выходных информационных объектов $\mathcal{I}^{out}$ может косвенно зависеть от множества факторов, которые наша модель учесть не может (возможные фоновые процессы, качество предсказания ветвления конкретной версии процессора, объем занятого кэша и др.). Для компенсации данной ошибки, оценку времени выполнения функции можно смоделировать в виде случайной величины:

$$\chi(f, \Pi, \mathcal{I}^{in}) = \tau(f, \Pi, \mathcal{I}^{in}) + \alpha,$$

где $\tau(f, \Pi, \mathcal{I}^{in})$ – детерминированная функция, представляющая зависимость времени выполнения функции f на машине с вектором характеристик производительности Π от вектора входных информационных объектов $\mathcal{I}^{in}$, α – это стохастическая величина с нулевым математическим ожиданием ($M[\alpha] = 0$), представляющая факторы, не входящие в разрабатываемую модель.

Таким образом, для оценки времени выполнения задач необходимо обеспечить сбор и хранение статистики запусков по всем классам задач. После каждого запуска задачи в базе данных сохраняется следующая информация: значения параметров запуска, вектор характеристик производительности вычислительной машины, включая количество выделенных процессорных ядер и объем выделенной оперативной памяти, время выполнения и объем сгенерированных выходных данных.

4. Заключение

В данной статье предложены модели предметной области и проблемно—ориентированной облачной вычислительной среды. Представленные модели поддерживают возможность прогнозирования характеристик выполнения вычислительных задач в распределенных вычислительных средах (таких как время выполнения и объем выходных данных). Данные оценки будут использоваться для более эффективного планирования ресурсов распределенных вычислительных сред.

В рамках развития данного исследования, планируется выявить соответствующие классы и построить для них соответствующие оценочные функции для прогнозирования времени выполнения задач и объема выходных данных при планировании многовариантных расчетов в облачных вычислительных системах.

Список литературы

[1]. Glotzer S.C. International assessment of research and development in simulation-based engineering and science. Imperial College Press, 2011. 312 p. doi: 10.1142/9781848166981.

[2]. Reed D. et al. Computational Science: Ensuring America's Competitiveness. United States. President's Information Technology Advisory Committee. National Coordination Office for Information Technology Research & Development, 2005. 104 p.

[3]. Folino G. et al. A grid portal for solving geoscience problems using distributed knowledge discovery services. Future Generation Computer Systystems, 26(1), 2010. P. 87–96. doi: 10.1016/j.future.2009.08.002.

[4]. Walker D.W. et al. The software architecture of a distributed problem-solving environment. Concurrency: Practice and Experience, 12(15), 2000. P. 1455–1480.

[5]. Радченко Г.И. Распределенные виртуальные испытательные стенды: использование систем инженерного проектирования и анализа в распределенных вычислительных средах. Вестник ЮУрГУ. Серия "Математическое моделирование и программирование", том 10, № 37(254), 2011 г. стр. 108–121.

[6]. Deelman E. et al. Workflows and e-Science: An overview of workflow system features and capabilities. Future Generation Computer Systystems, 25(5), 2009. P. 528–540.

[7]. Bil C. Concurrent Engineering in the 21st Century. Concurrent Engineering in the 21st Century: Foundations, Developments and Challenges, 2015. P. 421–454. doi: 10.1007/978-3-319-13776-6.

[8]. Шамакина А.В., Соколинский Л.Б. Исследование алгоритма планирования POS для проблемно-ориентированных вычислительных сред. Параллельные вычислительные технологии труды международной научной конференции (31 марта - 2 апреля 2015 г., г. Екатеринбург), 2015. стр. 488–493.

[9]. Weicker R.P. Dhrystone: a synthetic systems programming benchmark. Communications of the ACM, 27(10), 1984. P. 1013–1030. doi: 10.1145/358274.358283.

[10]. WPrime Systems. Super PI. 2013. URL: http://www.superpi.net/ (дата обращения: 14.11.2015).

[11]. Dongarra J.J., Luszczek P., Petite A. The LINPACK benchmark: Past, present and future. Concurrency and Computation: Practice and Experience. 15(9), 2003. P. 803–820. doi: 10.1002/cpe.728.

[12]. Demmel J., Dongarra J., Parlett B. Prospectus for the next LAPACK and ScaLAPACK libraries // PARA'06 Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing, 2006. P. 11–23. doi: 10.1007/978-3-540-75755-9.

Model of Problem-Oriented Cloud Computing Environment

G. Radchenko <gleb.radchenko@susu.ru>
South Ural State University,
454080, Russian Federation, Chelyabinsk, Lenina pr-kt, 76

Abstract. For efficient use of high-performance computing resources while implementing Computational Science methods for the study of physical, biological and social problems one can use problem-oriented distributed computing environments approach. They provide users with transparent access to the solution of specific classes of applications based on the available computing resources. To increase the effectiveness of such environments, one must use problem-oriented planning methods, which use the information about the subject area for predicting computing problems performance for optimal tasks planning and allocation. In the article the models of the subject area and problem-oriented cloud computing environment, focused on supporting the development of new problem-oriented scheduling algorithms are presented. Subject area *P* is defined as an ordered triple, consisting of a set of basic information objects *B*, the set of information object classes *C* and a set of functions defined over *C*. The task-oriented cloud computing environment can be defined as an ordered quadruple consisting of the set of nodes of a computer system *N*; a set of network connections *E*; a set of virtual machines images *M*, the basic subject area *P*. It should be required that within a problem-oriented computing environment the following functions for the prediction of the task execution, depending on the values of the input parameters for each class of problems were identified: the estimation of the amount of output data when a certain set of input parameters is given; the evaluation function of task execution time, given certain input parameters on the machine with the specified performance characteristics vector. Since it is impossible to estimate the time of the task execution with a perfect accuracy, task runtime evaluation should be modeled as a random variable. The provided model allows tasks execution time and output parameters volume evaluation through the collection, storage and analysis of statistics for all problems, executed in the environment.

Keywords: cloud computing; computing environment model; distributed computing system; problem-oriented computing environment.

DOI: 10.15514/ISPRAS-2015-27(6)-17

For citation: Radchenko G. Model of Problem-Oriented Cloud Computing Environment. *Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015,* pp. 275-284 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-17

References

[1]. Glotzer S.C. International assessment of research and development in simulation-based engineering and science. Imperial College Press, 2011. 312 p. doi: 10.1142/9781848166981.

[2]. Reed D. et al. Computational Science: Ensuring America's Competitiveness. United States. President's Information Technology Advisory Committee. National Coordination Office for Information Technology Research & Development, 2005. 104 p.

[3]. Folino G. et al. A grid portal for solving geoscience problems using distributed knowledge discovery services. Future Generation Computer Systystems, 26(1), 2010. P. 87–96. doi: 10.1016/j.future.2009.08.002.

[4]. Walker D.W. et al. The software architecture of a distributed problem-solving environment. Concurrency: Practice and Experience, 12(15), 2000. P. 1455–1480.

[5]. Radchenko G.I. Raspredelennye virtual'nye ispytatel'nye stendy: ispol'zovanie sistem inzhenernogo proektirovanija i analiza v raspredelennykh vychislitel'nykh sredakh. [Distributed virtual test-beds: usage of CAE systems in distributed computing environments] Vestnik JuUrGU. Serija "Matematicheskoe modelirovanie i programmirovanie" [SUSU Bulletin: The "Mathematical Modeling and Programming" series], vol 10, No 37(254), 2011. pp. 108–121. (In Russian).

[6]. Deelman E. et al. Workflows and e-Science: An overview of workflow system features and capabilities. Future Generation Computer Systystems, 25(5), 2009. P. 528–540.

[7]. Bil C. Concurrent Engineering in the 21st Century. Concurrent Engineering in the 21st Century: Foundations, Developments and Challenges, 2015. P. 421–454. doi: 10.1007/978-3-319-13776-6.

[8]. Shamakina A.V., Sokolinsky L.B. Issledovanie algoritma planirovanija POS dlja problemno-orientirovannykh vychislitel'nykh sred [Sudy of the the POS scheduling algorithm for problem-oriented computing environments] Parallel'nye vychislitel'nye tekhnologii trudy mezhdunarodnoj nauchnoj konferencii (31 marta - 2 aprelja 2015 g., g. Ekaterinburg) [Parallel Computing Technologies: proceedings of the International Scientific Conference (31 March - 2 April 2015 Ekaterinburg)], 2015. pp. 488–493. (In Russian).

[9]. Weicker R.P. Dhrystone: a synthetic systems programming benchmark. Communications of the ACM, 27(10), 1984. P. 1013–1030. doi: 10.1145/358274.358283.

[10]. WPrime Systems. Super PI. 2013. URL: http://www.superpi.net/ (дата обращения: 14.11.2015).

[11]. Dongarra J.J., Luszczek P., Petite A. The LINPACK benchmark: Past, present and future. Concurrency and Computation: Practice and Experience. 15(9), 2003. P. 803–820. doi: 10.1002/cpe.728.

[12]. Demmel J., Dongarra J., Parlett B. Prospectus for the next LAPACK and ScaLAPACK libraries // PARA'06 Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing, 2006. P. 11–23. doi: 10.1007/978-3-540-75755-9.

Расширение референтной модели облачной вычислительной среды в концепции крупномасштабных научных исследований[1]

А.В. Скатков <AVSkatkov@sevsu.ru>
В.И. Шевченко <VIShevchenko@sevsu.ru>
Севастопольский государственный университет,
299053, Россия, г. Севастополь, ул. Университетская, дом 33

Аннотация. В условиях развития технологий облачных вычислений актуальной является задача разработки новых фундаментальных подходов и проработки методов прикладной реализации сервис-ориентированных облачных сред, обеспечивающих непрерывное развитие и поддержку крупномасштабных научных исследований. В статье дается краткий обзор облачных информационных технологий, рассмотрены основные модели предоставления услуг облачных вычислений. Исследованы архитектура и особенности реализации моделей облачных вычислений на основе референтной модели NIST. В работе на общесистемном уровне описаны состав, цели и функции акторов облачной вычислительной среды, ориентированных на выполнение и поддержку крупномасштабных научных исследований. Приведены схемы базовых сценариев взаимодействия акторов облачной инфраструктуры.

Ключевые слова: облачная вычислительная среда; адаптация; брокер облачных сервисов; уровень ИТ-сервиса; конвергентная инфраструктура.

DOI: 10.15514/ISPRAS-2015-27(6)-18

Для цитирования: Скатков А.В., Шевченко В.И. Расширение референтной модели облачной вычислительной среды в концепции крупномасштабных научных исследований. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 285-306. DOI: 10.15514/ISPRAS-2015-27(6)-18.

1. Введение

Современное развитие информационных технологий, программного обеспечения и аппаратных средств позволяет организовывать сложные

[1] *Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований по проекту № 15-29-07936*

территориально-распределенные вычислительные системы для поддержки проведения крупномасштабных ресурсоемких фундаментальных и прикладных научных исследований. На сегодняшний день многие российские компании, научные организации и университеты используют облачные вычислительные среды с целью размещения своих научных и бизнес-приложений, что позволяет им избежать расходов на создание и поддержание функционирования собственных центров обработки данных. Владельцы облачных вычислительных систем путем консолидации вычислительных ресурсов и систем хранения данных способны снизить совокупную стоимость владения ИТ-инфраструктурой за счет обслуживания значительного количества пользователей, а также использования эффективных технических средств планирования и балансировки нагрузки, управления пересылкой данных в сети, интеграции нескольких территориально разрозненных сегментов систем хранения данных [1].

Кроме того, все большее распространение получает модель научной кооперации и разделения труда [2], основанная на совместной работе территориально распределенных международных коллективов исследователей в рамках концепции «сервис-ориентированной науки» (Service-Oriented Science) [3]. В соответствии с данной концепцией сервис-ориентированный подход позволяет организовать распределенный доступ к разнородным научным ресурсам, автоматизировать процесс научных исследований и тем самым повысить их эффективность.

В настоящее время отсутствуют проработанные подходы к реализации сервис-ориентированных облачных сред для крупномасштабных научных исследований. Обусловлено это как относительной новизной данного направления, так и рядом технологических проблем, стоящих на пути реализации подобных сред. В частности – не достаточный уровень компетенций исследователей в области установки, конфигурации и дальнейшей поддержки гарантированного уровня ИТ-сервисов в облачной вычислительной среде. Необходимо снижение технологического барьера для потенциальных пользователей-исследователей облачных сервисов, что может быть достигнуто путем привлечения сторонних посреднических служб и реализации проблемно-ориентированных интерфейсных сред, скрывающих техническую сторону функционирования сервисов и сложность поддерживаемой вычислительной инфраструктуры.

В данной статье предложена референтная модель облачной вычислительной среды, базирующаяся на эталонной архитектуре NIST (National Institute of Standards and Technology) [4, 5], имеющая ряд дополнений с учетом специфики решения крупномасштабных научных задач. Авторами предлагается описание на общесистемном уровне процессов взаимодействия акторов облачного сервиса при проведении крупномасштабных научных исследований. Целью исследований является разработка методологических основ для построения системы поддержки принятия решений по управлению

взаимодействием гетерогенных облачных агентов с целью повышения качества ИТ-сервисов при решении крупномасштабных научных задач.

В разделе 2 приведен обзор основных моделей предоставления услуг облачных вычислений и тенденции их развития. В разделе 3 даны необходимые определения, описаны основные акторы референтной модели облачной вычислительной среды, адаптируемой к специфике процессов научных исследований, приведены базовые сценарии взаимодействия акторов. В заключении приведены направления дальнейших исследований в развитии концепций облачных вычислительных сред ориентированных на поведение и поддержку крупномасштабных научных исследований.

2. Обзор основных моделей предоставления услуг облачных вычислений

Согласно определению, предложенному в [6], облачные вычисления – это модель организации удаленного доступа по запросу к разделяемому набору конфигурируемых вычислительных ресурсов, которые могут быть быстро выделены и освобождены с минимальными расходами на управление или взаимодействие с провайдером услуг.

В работах [6,7] описаны три основных модели обслуживания облачных сервисов:

- Программное обеспечение как сервис (Software as a Service, SaaS). Пользователю предоставляется возможность использования прикладного программного обеспечения провайдера, работающего в облачной инфраструктуре и доступного из различных клиентских устройств. Контроль и управление основной физической и виртуальной инфраструктурой облака осуществляется облачным провайдером.
- Платформа как сервис (Platform as a Service, PaaS). Пользователю предоставляется возможность использования облачной инфраструктуры с набором базового программного обеспечения для последующего размещения, разработки и тестирования на нем своих программных приложений. В состав таких платформ входят инструментальные средства создания, тестирования и выполнения прикладного программного обеспечения, предоставляемые облачным провайдером. Провайдер осуществляет контроль и управление физической и виртуальной инфраструктурой облака, за исключением приложений, разработанных и установленных пользователем.
- Инфраструктура как сервис (Infrastructure as a Service, IaaS). Пользователю предоставляется возможность использования облачной инфраструктуры для самостоятельного управления ресурсами обработки, хранения. Потребители развертывают собственное программное обеспечение, включая операционные системы и

приложения на инфраструктуре провайдера. Пользователи могут контролировать операционные системы, виртуальные системы хранения данных и установленные приложения, а так же осуществляют ограниченный контроль набора доступных сервисов.

Схема основных вычислительных моделей приведена на рис. 1. Применение таких облачных технологий как PaaS и IaaS привело к возрастающему переносу данных в облачные хранилища, что в свою очередь, вызвало необходимость решения задач интеграции композитных облачных приложений.

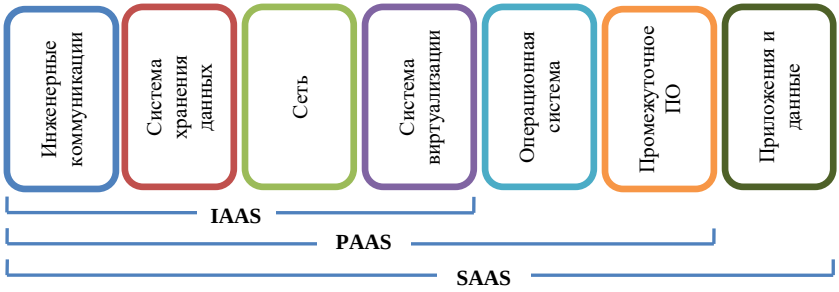


Рис. 1. Схема основных вычислительных моделей облачной инфраструктуры

В исследовании [8] сформулированы ключевые особенности новой модели облачных сервисов – Integration Platform as Service (iPaaS). iPaaS объединяет множественные облачные сервисы с целью интеграции и управления любой комбинацией внутренних и внешних (т.е. облачных) приложений, SOA и облачных сервисов, процессов и данных внутри и вне организации. Решения iPaaS способны обеспечивать многообразие сценариев интеграции и предоставлять безопасные средства доступа к корпоративным платформам [9]. В работе [10] рассмотрен вариант модели предоставления облачных сервисных услуг - аналитика как сервис (Data Mining as a Service, DMaaS) – данные, анализируемые пользователем «трансформируются» в микрокубы на «облаке». Кроме того предлагается трансформация не только данных, введенных в таблицу, но и любых данных предприятия, которое в таком случае оплачивает трансформационные затраты и анализирует данные. На сегодняшний день концепция облачных вычислений предполагает оказание потребителям различных дополнительных видов облачных услуг: Storage-as-a-Service, Database-as-a-Service, Information-as-a-Service, Process-as-a-Service, Integration-as-a-Service, Testing-as-a-Service [11-13]:

- Storage-as-a-Service («хранение как сервис»). Услуга дает возможность сохранять данные во внешнем хранилище, в «облаке». Для пользователя внешнее удаленное хранилище будет выглядеть, как дополнительный логический диск или папка.

- Database-as-a-Service («база данных как сервис»). Модель, основана на модели SaaS, в рамках данной модели пользователю предоставляется база данных требуемой конфигурации.
- Information-as-a-Service («информация как сервис»). Дает возможность пользователю удаленно использовать любые виды информации, в режиме реального времени.
- Process-as-a-Service («управление процессом как сервис»). Представляет собой удаленный ресурс, который может связать воедино несколько ресурсов (таких как услуги или данные, содержащиеся в пределах одного «облака» или других доступных «облаков»), для создания единого бизнес-процесса.
- Security-as-a-Service («безопасность как сервис»). Комплекс услуг по обеспечению безопасности, осуществляемое удаленно на базе системы, находящейся в собственности третьей стороны (одного или нескольких поставщиков услуги). Поставщик обеспечивает функции безопасности, основанные на разделяемых технологических услугах, которые потребляются в рамках модели «один ко многим» с оплатой по факту использования объема потребленных услуг. Данный вид услуги предоставляет возможность пользователям быстро развертывать программные продукты, позволяющие обеспечить безопасное использование Интернет-технологий, электронной переписки, локальной сети и т.п., что позволяет пользователям данного сервиса экономить на развертывании и поддержании своей собственной системы безопасности.
- Management/Governace-as-a-Service («администрирование и управление как сервис»). Дает возможность пользователям управлять и задавать параметры работы одного или многих «облачных» сервисов. В основном это такие параметры, как топология сети, использование ресурсов, виртуализация.
- Testing-as-a-Service («тестирование как сервис»). Дает пользователям возможность тестирования локальных или «облачных» вычислительных систем с использованием тестового программного обеспечения из «облака» (при этом никакого специального оборудования или обеспечения пользователям не требуется).

Соответствие дополнительных моделей облачных сервисов базовой трехзвенной архитектуре приведено на рис.2.

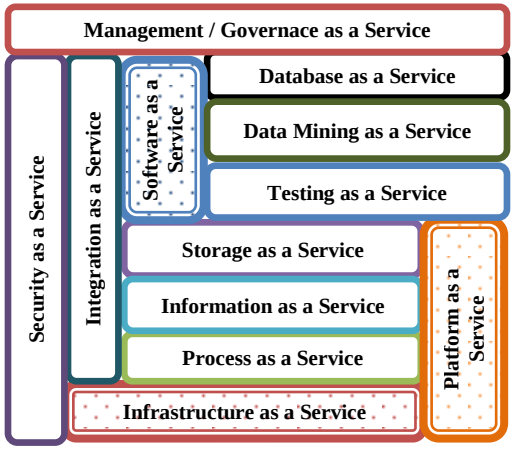


Рис. 2. Схема соответствия дополнительных моделей облачных сервисов базовой трехзвенной архитектуре по типам услуг

Тенденция развития облачных инфраструктур такова, что все вышеперечисленные типы моделей образуют новые виды сервисов, объединенные в обобщенную модель «Все как сервис» (Everything as a Service, EaaS) [14]. Идеологически близкими к этой модели являются модели XaaS (Anything as a Service – «какой-нибудь ресурс как сервис») [15] и *aaS (*as a Service – «любой ресурс как сервис»). В соответствии с моделью EaaS все ресурсы предполагаются реализуемыми в виде услуг. Предполагается также, что цены на различные облачные услуги неодинаковы, то есть час пользования каждой отдельной услугой имеет разную стоимость (тарифный план). Потенциальные объемы предоставления облачной услуги провайдером при имеющихся аппаратных и программных ресурсах в каждый конкретный момент формально не ограничены [16].

В рамках дальнейших исследований авторы статьи рассматривают модель EaaS, как наиболее полно адаптируемую к специфике крупномасштабных научных исследований.

3. Референтная модель облачной вычислительной среды

3.1 Основные понятия и определения

Согласно [17], референтная модель — концептуальная модель, формализующая рекомендованные практики ведения бизнеса в определенной области. Отличительными признаками референтной модели являются: отражение наилучших практик ведения бизнеса; универсальность применения (референтная модель представляет не отдельное предприятие, а класс предприятий); возможность повторного использования. Референтная модель является подвидом концептуальной модели, отражает основные

характеристики определенного класса предприятий, может быть использована для проектирования множества информационных систем и включает: функциональную структуру; объектную модель предметной области; процессную модель; функциональную модель; набор потенциальных точек контроля; набор операционных показателей деятельности предприятия.

Для построения модели облачной вычислительной среды, адаптированной для крупномасштабных научных вычислений в рамках концепции «сервис-ориентированной науки» авторами предлагается использовать эталонную архитектуру NIST [4-6], открытый интерфейс облачных вычислений, разработанный рабочей группой Open Cloud Computing Initiative международной организации Open Grid Forum [18] и методологию ITSM (IT Service Management), описанную в библиотеке ITIL (IT Infrastructure Library) [19-21]. В работах [18,19] даны ряд определений, применимых в референтной модели облачной среды.

Ресурс − физический или виртуальный объект с ограниченной доступностью. Физические ресурсы представляют собой вычислительные, запоминающие и коммуникационные ресурсы. Виртуальные ресурсы это, как правило, сервисы, которые предоставляют прямо или косвенно доступ к физическим вычислительным ресурсам.

Сервис-провайдер − субъект, который организует доступ к необходимому ресурсу или позволяет выполнить действие на ресурсе. Сервисы в свою очередь могут быть сервисами низкого уровня, которые работают в основном на физических ресурсах или сервисами высокого уровня, которые работают с виртуальными ресурсами (т.е. с другими сервисами). Сервисы проявляют своё назначение посредством интерфейса сервиса.

Система − набор сервисов и ресурсов, интегрированных в единое целое. Система может быть создана путём интеграции нескольких систем. Системы высокого уровня образуются из систем низкого уровня методом агрегации.

Порталы и научные входы − прикладная среда высокого уровня, которая ориентирована на упрощение работы конечного пользователя-исследователя.

Соглашение об уровне предоставления услуги (Service Level Agreement, SLA) − формальный договор между заказчиком услуги и её поставщиком, содержащий описание услуги, права и обязанности сторон и согласованный уровень качества предоставления данной услуги.

Процесс − это логически взаимосвязанная между собой последовательность работ (видов деятельности (activities)), направленная на достижение поставленной цели.

3.2 Акторы облачной вычислительной среды

В эталонной архитектуре облачных вычислений NIST (рис. 3) определен состав участников, деятельность и функции, которые могут быть реализованы

в процессе разработки архитектур облачных вычислений, установлены базовые взаимоотношения между участниками облачных вычислений.

Рис. 3. Эталонная архитектура облачных вычислений NIST

В предлагаемой авторами архитектуре выделено семь взаимодействующих акторов, выполняющих определенные функциональные задачи: Потребитель (П), Провайдер (Пр), Брокер (Б), Аудитор (А), Кризисный менеджер (КМ), Облачный композитный архитектор (ОКА), Регулятор (Р). В табл. 1 отражена связь функций акторов модели с соответствующими процессами поддержки ИТ-сервисов модели ITIL.

Табл. 1. Связь функций акторов модели облачной вычислительной среды с процессами управления ИТ-сервисами модели ITIL

Актор	Service Desk / Управление взаимодействием с пользователем	Предоставление услуг					Поддержка услуг				Управление безопасностью
		Управление финансами	Управление уровнем услуг	Управление непрерывностью ИТ-услуг	Управление доступностью	Управление мощностями	Управление релизами	Управление инцидентами	Управление проблемами	Управление конфигурациями	
П	*	*	*								*
Б	*	*	*	*	*	*					*
Пр	*	*	*	*	*	*	*	*	*	*	*
А	*		*			*		*	*		*
КМ	*		*		*	*		*	*		*
ОКА				*			*			*	
Р	*	*	*								

Далее описываются основные характеристики акторов, с учетом прикладной специфики решения крупномасштабных научных задач. Концептуальная схема архитектуры облачных вычислений представлена на рис. 4.

Облачный Потребитель (далее Потребитель, Клиент или Пользователь) является представителем множества Потребителей – организаций (научных сообществ), взаимодействующих с техническими средствами Провайдера при посредничестве Брокера. Потребитель проводит научные исследования с использованием сервисов облачной среды, предоставляемых Провайдером. В рамках предлагаемой модели, в зависимости от сценария взаимодействия, Потребитель может быть рассмотрен как атомарная сущность, либо как иерархическая структура.

Облачные Потребители категоризируются по трем группам, в зависимости от используемой модели облачных сервисов: SaaS − в этом случае Потребитель использует приложения/сервисы для автоматизации научных исследований; PaaS − Потребитель разрабатывает, тестирует, развертывает и управляет собственными приложениями (мониторинговыми облачными средами, СППР) по обработке данных и может предоставлять результаты своих исследований другим Потребителям, как сервисы; IaaS − в этом случае сервис Провайдера представляет собой своеобразную «социальную сеть» по проведению научных разработок и обработки данных. С точки зрения уровня полномочий Потребителя в проведении исследований быть произведена группировка по ролям: руководитель проекта, исследователь, инженер по сбору данных и т.п.

В качестве метрик оценки эффективности Потребителя с точки зрения управляющего Регулятора могут быть рассмотрены следующие показатели: уровень исследовательской компетенции; уровень технологической компетенции (степень владения технологией взаимодействия с облачным провайдером); уровень заинтересованности в результатах научных исследований (мотивированность).

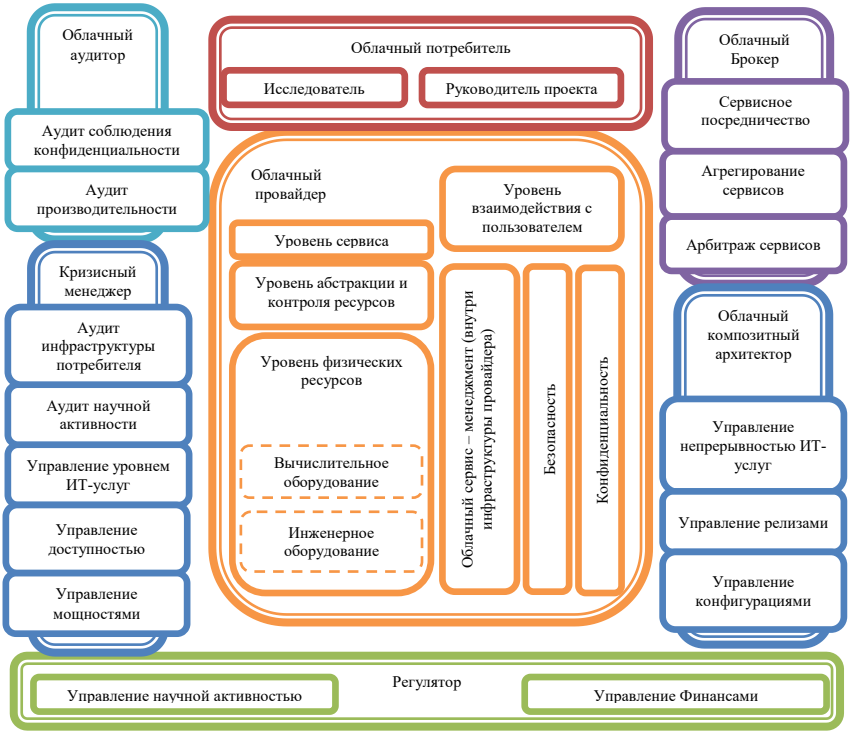


Рис.4. Концептуальная схема референтной архитектуры облачных вычислений

Облачный Провайдер (далее Провайдер) − лицо, организация или сущность, обеспечивает возможность пользования облачными услугами для Потребителя. Предоставляет услуги, включая уровень EaaS, в зависимости от действующего соглашения с Потребителем. В рамках рассматриваемой модели предполагаем, что Провайдер берет на себя функцию Облачного Оператора Связи, либо управление выбором Оператора Связи. Оператор Связи, как отдельный субъект управления не рассматривается. Качество ИТ-сервисов, предоставляемых Провайдером оценивается по метрикам ITIL [20].

Облачный Брокер (далее Брокер). В классической модели − сущность, управляющая использованием, производительностью и предоставлением облачных услуг, а также устанавливающая отношения между Провайдерами и Потребителями. Поскольку сообщество пользователей-исследователей может обслуживаться одновременно несколькими провайдерами на основе модели «гибридного облака» задача установления взаимосвязи «потребитель-провайдер» усложняется. В предлагаемой модели Брокер действует от лица Регулятора, руководствуется его нормативными управляющими воздействиями и целевой функцией с учетом динамически меняющихся ресурсных требований Потребителя. Динамизм ресурсных требований

Потребителя обусловлен спецификой решаемых научных задач и уровнем компетенций представителей Потребителя.

Регулятор – специфическая для рассматриваемого проекта сущность для отражения централизованного управления взаимодействием Потребителей с остальными сущностями модели, объединяющая с точки зрения процессов управления, множество Потребителей, Провайдеров и Брокеров и имеющая общесистемную целевую функцию управления крупномасштабными научными исследованиями. Осуществляет управление на основе анализа данных об эффективности использования информационно-вычислительных ресурсов, качестве ИТ-сервисов, уровне компетенции Потребителей, получаемых от Облачного Аудитора и Кризис-менеджера.

Облачный Аудитор – сущность, выполняющая независимую оценку облачных услуг, обслуживания информационных систем, производительности и безопасности реализации облака. С учетом специфики научных проектов, которые выполняются на основе ряда формальных правил, определяемых Регулятором, по распределению материальных, финансовых и информационных ресурсов, эта сущность будет осуществлять функции внешнего аудита Потребителей, Провайдеров и Брокеров и на основе анализа результатов аудита выделении наиболее критичных сущностей, с точки зрения выбранных Регулятором метрик качества.

Кризисный менеджер – класс специфических для рассматриваемого проекта сущностей, используемых в двух направлениях: ИТ-инфраструктура, научные исследования. Может применяться опционально, как альтернатива Аудитору, но с большими функциональными возможностями. Этот актор предназначен для использования в случае возникновения критических ситуаций в облачной инфраструктуре. Проводит анализ ИТ-инфраструктуры (уровня качества научных исследований) с целью выявления слабых и сильных мест и выработки мероприятий, позволяющих Регулятору вывести облачную инфраструктуру (или проводимое научное исследование) из кризиса с минимальными потерями. В условиях бюджетного финансирования ряда научных разработок внедрение такой сущности обосновано необходимостью минимизации нецелевого расходования бюджетных средств и максимизации инновационного эффекта от инвестиций в научные исследования.

Облачный композитный архитектор (далее Архитектор) – новая специфическая для рассматриваемого проекта сущность. Это лицо или организация, отвечающая за разработку архитектуры композитного приложения и адаптацию его в облачной инфраструктуре для конкретного научного проекта, повышение уровня компетенций Потребителей с целью эффективной проведения исследований в разработанной композитной среде.

Подробнее взаимодействие акторов в архитектуре облачных вычислений рассматривается на примере сценариев, с учетом сценариев изложенных в NIST.

3.3 Базовые сценарии взаимодействия акторов в архитектуре облачных вычислений

В предлагаемых сценариях акторы разбиты на группы, в зависимости от числа взаимодействующих сущностей: <1:1>, <1:N>,<M:N>. Рассмотрим последовательно сценарии взаимодействия акторов в архитектуре облачных вычислений, в порядке их усложнения.

Сценарий 1 (рис.5а): Потребитель запрашивает услугу (сервис) у Облачного Провайдера. Между Провайдером и Потребителем устанавливается соглашение об уровне обслуживания SLA. В случае изменения требований Потребителя к предоставляемым ИТ-сервисам или несоответствие предоставляемых Провайдером услуг метрикам SLA соглашение об уровне обслуживания пересматривается, либо Потребитель меняет Провайдера.

Задача Потребителя минимизировать расходы при максимальном уровне ИТ-сервиса. Задача Провайдера максимизировать доходы от обслуживания Потребителя и минимизировать выделяемые технические и технологические ресурсы при соблюдении гарантированного уровня SLA.

Сценарий 2 (рис. 5б): Множество облачных Потребителей запрашивает услугу (сервис) у одного облачного Провайдера. Взаимодействия между отдельными Потребителями и Провайдером аналогичны Сценарию 1.

а) *б)*

Рис.5. Схема сценариев 1(а) и 2(б)

Перед Провайдером стоит задача эффективного распределения ресурсов и сервисов между множеством Потребителей в условиях реального времени, с учетом динамически меняющихся потребностей отдельных Потребителей.

Сценарий 3 (рис. 6): Иерархически организованное множество Потребителей запрашивает сервис у Провайдера. Рассмотрим внутреннюю структуру сущности Облачный Потребитель. Если в качестве Потребителя выступает атомарная сущность – физическое лицо, организация или подразделение, рассматриваемые как единый объект, то для таких сущностей применимы Сценарии 1 и 2. В случае, когда Потребитель имеет иерархическую структуру, то его атомарной сущностью является Пользователь, непосредственно взаимодействующий с ИТ-сервисами и техническими средствами Провайдера путем генерации первичных или вторичных данных, получения к ним доступа и инициации процессов по корректировке объема потребляемых ресурсов Провайдера. Однако, в этом сценарии Облачный Провайдер невидим Потребителю. Вместо прямого взаимодействия с Провайдером Потребители-пользователи делегируют полномочия взаимодействия с Провайдером одному Потребителю, осуществляющему

руководство научным проектом. Потребители-пользователи формируют заявки на ресурсы и сервисы в зависимости от решаемых ими научных задач. На основе этих данных Потребитель-руководитель требования к соглашению об уровне обслуживания SLA с Облачным Провайдером.

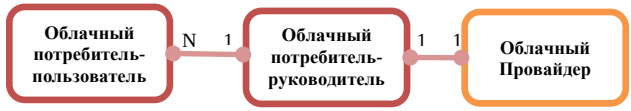


Рис.6. Схема сценария 3

Перед Потребителем-руководителем стоит задача эффективного распределения ресурсов и сервисов между множеством Потребителей-исполнителей в условиях минимизации затрат на услуги Провайдера. Взаимодействие между Потребителем-руководителем и Провайдером аналогично Сценарию 1.

Сценарий 4 (рис. 7): Множество Потребителей, имеющих иерархическую структуру, запрашивает сервис у одного облачного Провайдера.

Рис.7. Схема сценария 4

Данный сценарий является комбинацией Сценария 2 и Сценария 3.

Сценарий 5 (рис. 8): Потребитель запрашивает услугу у Облачного Брокера вместо прямого взаимодействия с Провайдером. Брокер может создать новый сервис, комбинируя набор сервисов или расширяя существующий сервис, в зависимости от требований Потребителя, за счет использования ресурсов и сервисов нескольких Провайдеров. В этом сценарии Провайдер невидим Облачному Потребителю.

Рис.8. Схема сценария 5

Задача Брокера максимизировать экономический эффект от посреднических услуг при взаимодействии Потребителя и Провайдера. Задача Потребителя минимизировать расходы на услуги Брокера при максимальном уровне сервиса. Задача Провайдера максимизировать доходы от взаимодействия с

Брокером и минимизировать выделяемые технические и технологические ресурсы при соблюдении гарантированного уровня SLA.

Сценарий 6: Множество Потребителей запрашивает услугу у Облачного Брокера. Взаимодействия между отдельными Потребителями и Брокером, а так же между Брокером и Провайдером аналогичны Сценарию 5.

Задача Брокера максимизировать совокупный экономический эффект от посреднических услуг при взаимодействии с множеством Потребителей со множеством Провайдеров.

Сценарий 7 (рис.9): Иерархически организованное множество Потребителей запрашивает услугу у Облачного Брокера. Данный сценарий является комбинацией Сценария 3 и Сценария 5. В этом сценарии Провайдер и Брокер невидимы Потребителю-пользователю, Облачный Провайдер не видим так же Потребителю-руководителю.

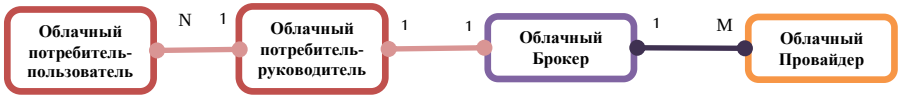


Рис.9. Схема сценария 7

Перед Потребителем-руководителем стоит задача эффективного распределения ресурсов и сервисов между множеством Потребителей-исполнителей в условиях минимизации затрат на посреднические услуги Брокера. Взаимодействие между Потребителем-руководителем и Потребителями-пользователями аналогично Сценарию 3. Взаимодействие Потребитель-руководитель — Брокер, Брокер — Провайдер аналогично Сценарию 5.

Сценарий 8: Множество Облачных Потребителей, имеющих иерархическую структуру, запрашивает услугу у одного Облачного Брокера. Предлагаемый сценарий является комбинацией Сценария 4 и Сценария 7. В данном сценарии Потребитель-пользователь не взаимодействует напрямую с сущностями Брокер и Провайдер. Задача Брокера максимизировать совокупный экономический эффект от посреднических услуг при взаимодействии с множеством Потребителей, имеющих иерархическую структуру, и множеством Провайдеров.

Сценарий 9 (рис.10): Облачный Аудитор проводит независимую оценку обслуживания и безопасности реализации облачной услуги. В случае выявления Аудитором несоответствия предоставляемых Провайдером услуг метрикам SLA соглашение об уровне обслуживания между Потребителем и Провайдером пересматривается, либо Потребитель меняет Провайдера.

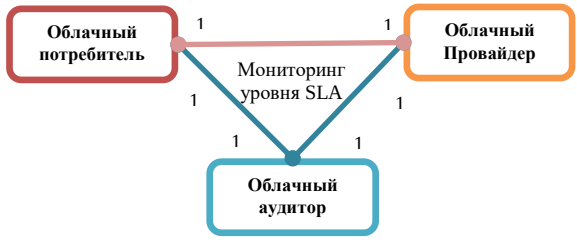


Рис.10. Схема сценария 9

В задачи Аудитора входят только мониторинговые функции и их оценка в соответствии с эталонными значениями метрик качества ИТ-сервисов. Принятие управляющих решений осуществляется на стороне Потребителя.

Сценарий 10 (рис.11): Аудитор проводит независимую оценку обслуживания и безопасности реализации облачной услуги для иерархически организованного множества облачных Потребителей. На основе данных о качестве ИТ-сервисов, получаемых от Потребителей-пользователей облачный Потребитель-Руководитель инициирует процедуру аудита ИТ-сервисов. Предлагаемый сценарий базируется на сценарии 9. Функции первичного мониторинга качества ИТ-сервисов могут быть делегированы Потребителям-пользователям, в этом случае данные о результатах мониторинга передаются через Потребителя-руководителя Аудитору. Прямого взаимодействия между Аудитором и Потребителем-пользователем сценарий не предусматривает.

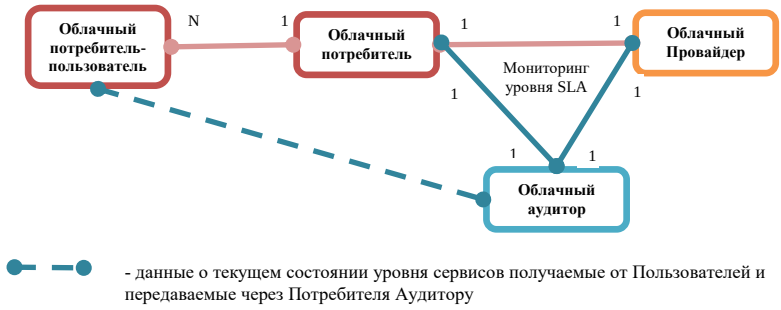


Рис.11. Схема сценария 10

В задачи Аудитора входят обработка результатов первичного мониторинга и их оценка в соответствии с эталонным значениями метрик качества ИТ-сервисов. Принятие управляющих решений осуществляется Потребителем-руководителем.

Сценарий 11 (рис.12): Аудитор проводит независимую оценку обслуживания и безопасности реализации облачной услуги для множества иерархически организованных групп облачных Потребителей. Аудит группы облачных

потребителей инициируется Регулятором, взаимодействия между Аудитором и Регулятором выделены в отдельном сценарии.

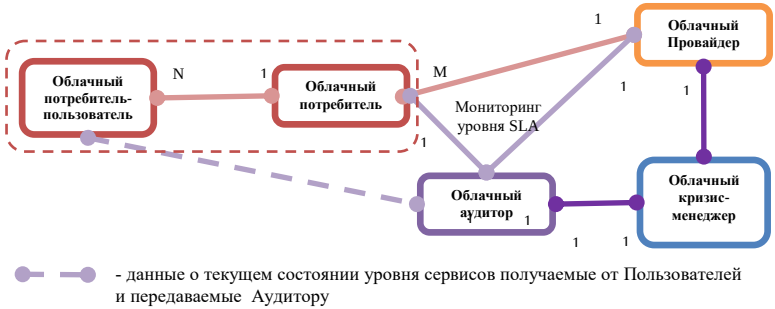


Рис.12. Схема сценария 11

В задачи Аудитора входят: обработка результатов первичного мониторинга; их оценка в соответствии с эталонными значениями метрик качества ИТ-сервисов; классификация ИТ-инфраструктур, поддерживаемых Провайдером по степени критичности и передача информации о наиболее критичных инфраструктурах Кризис-менеджеру. Кризис-менеджер на основе экспертных оценок и собственной базы знаний формирует набор рекомендаций по реинжинирингу критичных ИТ-инфраструктур для Провайдера.

Сценарий 12: Аудитор проводит независимую оценку качества проводимых исследований, проводимых множеством иерархически организованных групп облачных Потребителей. Аудит группы Потребителей инициируется Регулятором, взаимодействия между Аудитором и Регулятором выделены в отдельном сценарии. В предлагаемой схеме каждая из иерархически организованных групп Потребителей выполняет свой обособленный набор функциональных задач в рамках научных исследований.

В задачи Аудитора входят: обработка результатов первичного мониторинга; их оценка в соответствии с эталонными значениями метрик качества научных исследований; классификация групп облачных Потребителей-исполнителей, под управлением Потребителя-руководителя по степени критичности и передача информации о наиболее критичных инфраструктурах Потребителей Кризис-менеджеру. Кризис-менеджер на основе экспертных оценок и собственной базы знаний формирует набор рекомендаций по реинжинирингу бизнес-процессов задач научных исследований в критичных инфраструктурах Потребителей. На основе этих рекомендаций формируются сценарии взаимодействия для Регулятора и Композитного архитектора.

Сценарий 13 (рис.13): Композитный Архитектор разрабатывает оригинальную архитектуру композитного приложения для нового научного проекта Потребителя и адаптирует приложение к облачной инфраструктуре выбранного Провайдера.

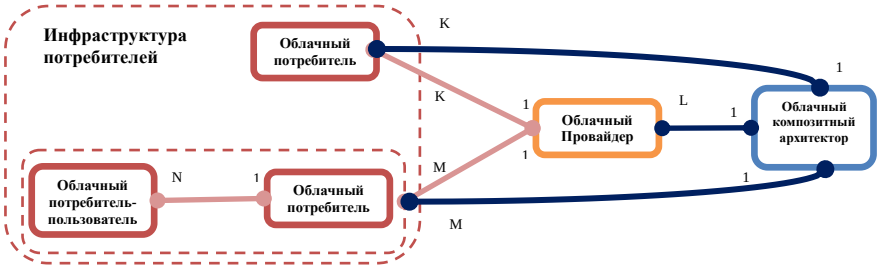


Рис.13. Схема сценария 13

Сценарий 14: Облачный Кризис-менеджер в случае отсутствия готовых решений по разрешению кризисных ситуаций при взаимодействии Потребителей и Провайдеров передает управление на более высокий уровень эскалации – Композитному Архитектору. Архитектор разрабатывает оригинальную архитектуру композитного приложения для конкретного научного проекта Потребителя и адаптирует приложение к облачной инфраструктуре выбранного Провайдера. Предложенный сценарий функционально расширяет сценарий 13.

Сценарий 15 (рис.14): Регулятор получает данные от внешних систем мониторинга и Аудитора.

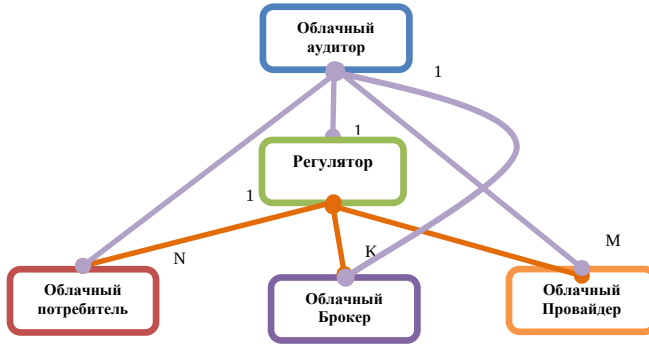


Рис.14. Схема сценария 15

От инфраструктуры Потребителей: данные о состоянии проводимых научных исследований, включая текущие значения метрик оценки качества; данные о необходимых ИТ - сервисах; данные о расходовании бюджетов на научно-исследовательские работы (НИР). От Облачных Брокеров, посредством Аудитора или напрямую Регулятор получает данные: о текущих требованиях к ИТ-сервисам; о составе предоставляемых ИТ-сервисов конкретному Потребителю; о группе Провайдеров, предоставляющих полный комплекс сервисов Потребителю. От Провайдеров Регулятор получает информацию об эффективности использования отдельным Пользователем предоставляемых ИТ-сервисов. Задача Регулятора – посредством встроенной системы

поддержки принятия решений на основе полученных мониторинговых данных осуществлять эффективное управление всей облачной инфраструктурой.

4. Заключение

Анализ работ в области моделирования облачных сервисов с учетом специфики крупномасштабных научных задач показал, что на сегодняшний день актуальным является развитие направления поддержки ИТ-сервисов «Все как сервис» (EAAS). Вследствие чего существующие модели концептуального описания взаимодействия акторов облачной среды нуждаются в расширении с учетом специфики поставленных задач. В данной статье на общесистемном уровне описаны процессы взаимодействия акторов облачного сервиса при проведении крупномасштабных научных исследований. Исследована архитектура и особенности реализации модели облачных вычислений, основанной референтной модели NIST, но включающей дополнительных акторов: кризис-менеджер и облачный композитный архитектор. Особенностью предлагаемой референтной модели является адаптация к специфике крупномасштабных научных задач, решаемых в распределенных средах на базе модели гибридного облака. В предлагаемой модели, в отличии от модели NIST, учитывается процессный подход к управлению взаимодействием акторов облачной среды на основе методологии ITSM. В работе приведены схемы базовых сценариев взаимодействия акторов облачной инфраструктуры, которые могут быть расширены и дополнены в зависимости от специфики функциональных задач.

В дальнейшем планируется концептуальное описание отдельных актров предлагаемой референтной модели, включающее: параметры, наиболее полно характеризующие актора; целевые функции акторов. На базе этого разработка диаграмм концептуальных классов в нотации UML, позволяющей перевести описание сценариев поведения облачной вычислительной среды в единую инфологическую модель.

Список литературы

[1]. Miller R. Who Has the Most Web Servers? May 14, 2009 (http://www.datacenterknowledge.com/archives/2009/05/14/whos-got-the-most-web-servers/)

[2]. Сухорослов О. Реализация и композиция проблемно-ориентированных сервисов в среде MathCloud. Вестник ЮУрГУ, Серия «Математическое моделирование и программирование», вып. 8, №17 (234), 2011 г. стр. 101-112.

[3]. Foster I. Service-Oriented Science. Science, 308 (5723), 2005. P. 814-817.

[4]. Liu F., Tong J., Mao J., Bohn R., Messina J., Badger L., Leaf D. NIST Cloud Computing Reference Architecture. Recommendations of the National Institute of Standards and Technology. Cloud Computing Program Information Technology Laboratory National Institute of Standards and Technology Gaithersburg, MD 20899-8930 September 2011 (http://www.nist.gov/customcf/get_pdf.cfm?pub_id=909505)

[5]. Mell P., Grance T. The NIST Definition of CloudComputing. Recommendations of the National Institute of Standards and Technology, 2010.

[6]. Hogan M., Liu F., Sokol A., Tong J. NIST Cloud Computing Standards Roadmap. Computer Security Division Information Technology Laboratory National Institute of Standards and Technology Gaithersburg, MD 20899-8930 July 2011 (http://www.nist.gov/customcf/get_pdf.cfm?pub_id=909024)

[7]. Прудникова А., Садовникова Т. Анализ облачных сервисов с точки зрения информационной безопасности. T-Comm - Телекоммуникации и Транспорт, вып.7, 2012 г. стр. 153-156

[8]. Pazzini M., Benoit J. Lheureux. Integration Platform as a Service: Moving Intergation to the Cloud. Gartner RAS Core Research Note G00210747, March 7, 2011 (https://www-304.ibm.com/industries/publicsector/fileserve?contentid=250736)

[9]. Еловиков А. Новые тенденции интеграции в облаке. Научный журнал КубГАУ, №83(09), 2012 г. (http://ej.kubagro.ru/2012/09/pdf/36.pdf)

[10]. Афанасьев С. Облачные сервисы, онтологическое моделирование. Труды СПИИРАН, вып. 4(23), 2012 г. стр. 392-399

[11]. Тим Джонс М. Cloud Computing и Linux Платформы и приложения для Cloud Computing. Developer Works, 25.12.2008 (https://www.ibm.com/developerworks/ru/library/l-cloud-computing/l-cloud-computing-pdf.pdf)

[12]. Рогальский Е. Облачные технологии и их роль в развитии электронного обучения. Исследования наукограда, №1(7), 2014 г. стр. 42-49

[13]. Баженова И. Применение облачных технологий при дистанционном обучении языкам программирования. Вестник Московского государственного лингвистического университета, №13 (699), 2014 г. стр. 45-52

[14]. Everything as a Service (EaaS). URL: http://www.csc. com/business_drivers/offerings/78750-everything_as_a_service_eaas (дата обращения: 15.01.2014)

[15]. XaaS (anything as a service). URL: http://searchcloudcomputing.techtarget.com/definition/XaaS-anything-as-a-service (дата обращения: 15.01.2014)

[16]. Заложнев А., Чистов Д., Шуремов Е. Об одном подходе к реализации облачных услуг на основе модели EAAS. Программные продукты и системы, №2 (106), 2014 г. стр.188-192

[17]. Куприйчук А., Овчинников П. Референтная модель. Электронный журнал: Управление предприятием, №3(14), март 2012 (http://consulting.1c.ru/ejournalPdfs/ovchinnikov_pSUn.pdf)

[18]. Jha S., Merzky A., Fox G. Using Clouds to Provide Grids Higher-Levels of Abstraction and Explicit Support for Usage Modes. Open Grid Forum. GFD-I.150, 2009 (https://www.ogf.org/OGF_Special_Issue/cloud-grid-saga.pdf)

[19]. Ингланд Р. Введение в реальный ITSM: пер. с англ. – М.: Лайвбук, 2010. 132 с.

[20]. Брукс П. Метрики для управления ИТ-услугами: пер. с англ. – М: Альпина Бизнес Букс, 2008. 283 с.

[21]. Бон Я.В., Кеммерлинг Г., Пондаман Д. ИТ Сервис-менеджмент, введение . – М: IT Expert, 2003. 215 с.

Expansion of reference model for the cloud computing environment in the concept of large-scale scientific researches

A. Skatkov < AVSkatkov@sevsu.ru>
V. Shevchenko < VIShevchenko@sevsu.ru>
Sevastopol State University, 33 University Str.,
Sevastopol, 299053, Russian Federation

Abstract. Actual problem of the development of new approaches to the implementation of fundamental service-oriented cloud environments to support large-scale scientific research is reviewed in the article. Review of modern technologies and models of cloud services, cloud computing is executed in the publication. The relevance of using the cloud service model "Everything As a Service" in distributed environments research is revealed. Architecture and features of realization of cloud computing, based on the reference model NIST were investigated. The structure of the reference model of a cloud computing environment for the scientific researchers is offered. The composition, objectives and functions of the actors of cloud infrastructure are described on the system level. In the model introduced new actors: the regulator, the crisis manager, the architect of composite applications. Actors interrelation of model of cloud computing environment with the business-processes of IT service management model is installed. Examples which illustrate basic scenarios of interaction of actors of cloud infrastructure, is given. On the basis of the reference model, which was proposed in the future, it is planned to develop a set of models, methods and algorithms that will allow to maintain a guaranteed level of IT services to the cloud, specializing in solving large-scale scientific problems. The work is supported by RFBR (grant 15-29-07936).

Keywords: cloud computing environment; adaptation; cloud broker; the level of IT services; convergent infrastructure.

DOI: 10.15514/ISPRAS-2015-27(6)-18

For citation: Skatkov A., Shevchenko V. Expansion of reference model for the cloud computing environment in the concept of large-scale scientific researches. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 285-306 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-18

References

[1]. Miller R. Who Has the Most Web Servers? May 14, 2009 (http://www.datacenterknowledge.com/archives/2009/05/14/whos-got-the-most-web-servers/)

[2]. Sukhoroslov O. Realizatsiya i kompozitsiya problemno-orientirovannykh servisov v srede MathCloud [Implementation and composition of domain-oriented services in the MathCloud environment]. Vestnik YuUrGU, Seriya «Matematicheskoe modelirovanie i programmirovanie» [Bulletin of the South Ural State University. Series «Mathematical

Modelling, Programming & Computer Software»] vol. 8, №17 (234), 2011, pp. 101-112. (in Russian)]
[3]. Foster I. Service-Oriented Science. Science, 308 (5723), 2005. P. 814-817.
[4]. Liu F., Tong J., Mao J., Bohn R., Messina J., Badger L., Leaf D. NIST Cloud Computing Reference Architecture. Recommendations of the National Institute of Standards and Technology. Cloud Computing Program Information Technology Laboratory National Institute of Standards and Technology Gaithersburg, MD 20899-8930 September 2011 (http://www.nist.gov/customcf/get_pdf.cfm?pub_id=909505)
[5]. Mell P., Grance T. The NIST Definition of CloudComputing. Recommendations of the National Institute of Standards and Technology, 2010.
[6]. Hogan M., Liu F., Sokol A., Tong J. NIST Cloud Computing Standards Roadmap. Computer Security Division Information Technology Laboratory National Institute of Standards and Technology Gaithersburg, MD 20899-8930 July 2011 (http://www.nist.gov/customcf/get_pdf.cfm?pub_id=909024)
[7]. Prudnikova A., Sadovnikova T. Analiz oblachnykh servisov s tochki zreniya informatsionnoy bezopasnosti [Analysis of cloud services in terms of information security]. T-Comm - Telekommunikatsii i Transport [T-Comm - Telecommunications and Transport], № 7, 2012, pp. 153-156. (In Russian)
[8]. Pazzini M., Lheureux Benoit J. Integration Platform as a Service: Moving Intergation to the Cloud. Gartner RAS Core Research Note G00210747, March 7, 2011 (https://www-304.ibm.com/industries/publicsector/fileserve?contentid=250736)
[9]. Elovikov A. Novye tendentsii integratsii v oblake [EMERGING TRENDS OF INTEGRATION IN THE CLOUD]. Nauchnyi zhurnal KubGAU [Scientific Journal of KubSAU], №83(09), 2012 (http://ej.kubagro.ru/2012/09/pdf/36.pdf) (In Russian)
[10]. Afanasiev S. Oblachnye servisy, ontologicheskoe modelirovanie [CLOUD SERVICES, ONTOLOGICAL MODELING OF TAXONOMY]. Trudy SPIIRAN [SPIIRAS Procedings], issue 4(23), 2012. pp. 392-399 (In Russian)
[11]. Tim Dzhons M. Cloud Computing i Linux Platformy i prilozheniya dlya Cloud Computing [Cloud Computing and Linux. Platforms and Applications for Cloud Computing]. IBM Developer Works, 25.12.2008 (https://www.ibm.com/developerworks/ru/library/l-cloud-computing/l-cloud-computing-pdf.pdf). (In Russian)
[12]. Rogalskiy E. Oblachnye tekhnologii i ikh rol v razvitii elektronnogo obucheniya [Role of cloud technologies in the development of the e-learning]. Issledovaniya naukograda [The research of the science city]. №1(7), 2014, pp. 42-49 (In Russian)
[13]. Bazhenova I. Primenenie oblachnykh tekhnologiy pri distantsionnom obuchenii yazykam programmirovaniya [THE USE OF CLOUD TECHNOLOGY IN DISTANCE LEARNING PROGRAMMING LANGUAGE]. Vestnik Moskovskogo gosudarstvennogo lingvisticheskogo universiteta [Bulletin of the MSLU], №13 (699), 2014, pp. 45-52. (In Russian)
[14]. Everything as a Service (EaaS). URL: http://www.csc. com/business_drivers/offerings/78750-everything_as_a_service_eaas (дата обращения: 15.01.2014)
[15]. XaaS (anything as a service). URL: http://searchcloudcomputing.techtarget.com/definition/XaaS-anything-as-a-service (дата обращения: 15.01.2014)
[16]. Zalozhnev A., Chistov D., Shuremov E. Ob odnom podkhode k realizatsii oblachnykh uslug na osnove modeli EAAS [EaaS model based approach to cloud services

provision]. Programmnye produkty i sistemy [Software & Systems], №2 (106), 2014, pp. 188-192. (In Russian)

[17]. Kupriychuk A., Ovchinnikov P. Referentnaya model [Reference model]. Elektronnyi zhurnal: Upravlenie predpriyatiem [Electronic Journal: Enterprise Managemen], №3(14), march 2012 (http://consulting.1c.ru/ejournalPdfs/ovchinnikov_pSUn.pdf). (In Russian)

[18]. Jha S., Merzky A., Fox G. Using Clouds to Provide Grids Higher-Levels of Abstraction and Explicit Support for Usage Modes. Open Grid Forum. GFD-I.150, 2009 (https://www.ogf.org/OGF_Special_Issue/cloud-grid-saga.pdf)

[19]. Ingland R. Vvedenie v realnyi ITSM [Introduction to real itsm]: per. s angl. – M.: Laivbuk [Livebook], 2010. 132 p. (In Russian)

[20]. Bruks P. Metriki dlya upravleniya IT-uslugami [Metrics for IT Service Management]: per. s angl. – M: Alpina Biznes Buks [Alpina business books], 2008. 283 p. (In Russian)

[21]. Bon Ya.V., Kemmerling G., Pondaman D. IT Servis-menedzhment, vvedenie [IT Service Management, introduction]. – M: IT Expert, 2003. 215 p. (In Russian)

Динамическая оптимизация нагрузки на вычислительных узлах частных, публичных и гибридных облаков

А.С. Чадин <a.chadin@servionica.ru>
Сервионика, Россия, г. Москва, ул. Кедрова, 15

Аннотация. Данная система предназначена для автоматизированного распределения нагрузки в кластере путем анализа загруженности вычислительных узлов и последующей миграции виртуальных машин с загруженных узлов на менее загруженные. Помимо стабилизации нагрузки рассмотрена возможность снижения энергопотребления путем разгрузки слабонагруженных узлов и перевода их в ждущий режим. Стабилизация нагрузки в кластере ведет к повышению стабильности и сокращению времени выполнения запросов.

Ключевые слова: Балансировка; Кластер; Облачные вычисления

DOI: 10.15514/ISPRAS-2015-27(6)-19

Для цитирования: Чадин А.С. Динамическая оптимизация нагрузки на вычислительных узлах частных, публичных и гибридных облаков. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 307-314. DOI: 10.15514/ISPRAS-2015-27(6)-19.

1. Введение

Система OpenStack [1], представляющая собой комплекс проектов свободного обеспечения для создания облачной инфраструктуры, имеет положительную репутацию в среде поставщиков и разработчиков облачных технологий. OpenStack не имеет на данный момент автоматическую систему балансировки виртуальных машин, что отрицательно сказывается на стабильности вычислительного кластера. Ввиду того, что в проекте OpenStack Nova уже созданы все необходимые инструменты для проведения ручной балансировки, предлагается автоматизировать процесс балансировки внутри вычислительного кластера.

Целью данной работы является разработка автоматизированной системы балансировки нагрузки в контексте системы OpenStack.

В рамках данной работы было проведено исследование и предоставлено одно из возможных решений построения сервиса автоматической балансировки для OpenStack. Решение является модульным и позволяет использовать различные подходы для реализации механизма балансировки. Предложенный в работе подход выполнен на основе среднеквадратичного отклонения.

2. Структура сервиса балансировки загруженности узлов

Разработанный сервис OpenStack Nova LoadBalancer позволяет автоматизировать балансировку нагрузки между узлами. Он собирает данные, анализирует их и принимает решение о миграции виртуальных машин с одного узла на другой. Сервис LoadBalancer является частью системы OpenStack Nova, что позволяет вызывать внутренние функции без использования внешнего API.

Рассмотрим структуру Nova LoadBalancer:

- Модуль сбора статистических данных
- Модуль анализа собранных данных и принятия решений.
- Модуль балансировки виртуальных машин (overload-алгоритм).
- Модуль определения недостаточности загруженности вычислительных узлов (underload-алгоритм).
- Вспомогательные службы (API, правила балансировщика).

Данные модули имеют базовые классы, что позволяет использовать собственные реализации классов.

Рассмотрим работу балансировщика поподробнее.

2.1 Сбор статистических данных

Для анализа требуется получать с каждого узла данные о загруженности вычислительных ресурсов. Показатель загруженности CPU узла получается с использованием Python-библиотеки psutil. Показатель свободной оперативной памяти узла вычисляется как сумма свободной, кэшируемой и буферной памяти. Сведения о памяти можно найти в файле meminfo директории /proc/.

Библиотека управления виртуализацией libvirt позволяет получать актуальные данные об использовании ресурсов виртуальной машиной. Сервис nova-compute был модифицирован таким образом, чтобы он с определенным интервалом отправлял данные по загруженности узла (процентная загруженность CPU, загруженность RAM в мегабайтах) и загруженности каждой виртуальной машины (процессорное время, загруженность RAM в мегабайтах), расположенной на нем, в сервис Nova Conductor, обеспечивающий взаимодействие с базой данных. Полученная информация записывается в таблицы «compute_node_stats» и «instances_stat». Данные о

загруженности виртуальных машин и узлов заполняются за счет использования атомарного метода UPSERT.

Схема сбора статистических данных и используемые компоненты OpenStack Nova представлена на рис. 1. Необходимо отметить, что расположение сервисов OpenStack Nova может различаться в зависимости от конфигурации.

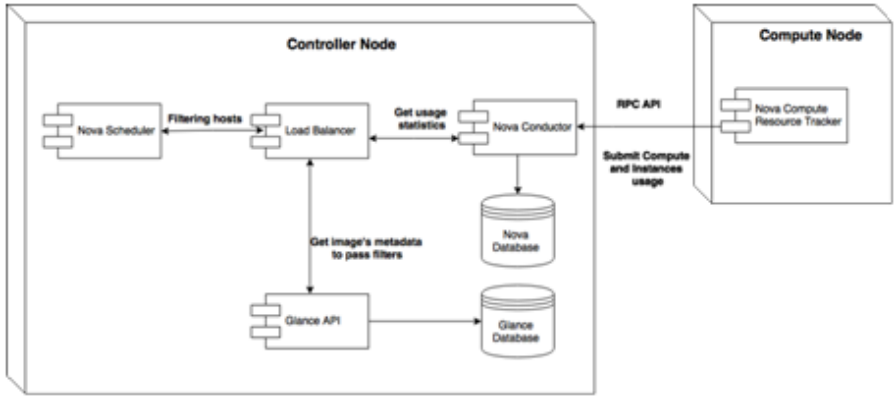


Рис.1. Диаграмма развёртывания сервиса OpenStack Nova LoadBalancer.

2.2 Анализ собранных данных и принятие решений

По показателям CPU и RAM в конфигурационном файле Nova устанавливаются пороговые значения среднеквадратичного отклонения. Данный модуль запускается с определенной в конфигурационном файле периодичностью.

Собранные данные необходимо преобразовать в процентные показатели. Следует нормализовать показатели в границах [0,1]. Для этого процессорное время переводится в загруженность CPU по следующей формуле:

$$CPU = \frac{(cpu_{time} - oldcpu_{time})}{(time_{current} - time_{old}) * vcpus} * 10^7,$$

где:

- cpu_time – процессорное время, затраченное виртуальной машиной для обработки процессов (задач). Выражено в наносекундах.
- oldcpu_time – процессорное время, затраченное виртуальной машиной в предыдущий момент времени (во время time_old).
- time_current – время, в которое был снят показатель cpu_time. Выражено в секундах.
- time_old – время, в которое был снят показатель oldcpu_time. Выражено в секундах.
- vcpus – количество ядер, выделенное данной виртуальной машине.

Загруженность RAM вычисляется как отношение занятой памяти к общему количеству памяти узла.

После обработки загруженности по всем узлам, вычисляется среднее арифметическое по CPU и RAM и среднеквадратичное отклонение. Полученные значения сравниваются с пороговыми значениями, указанными в конфигурационном файле, на основании чего делается вывод о необходимости стабилизации кластера. В случае, если полученные значения ниже пороговых, балансировщик проверит кластер underload-алгоритмом на энергоэффективность. В противном случае, запускается модуль балансировки виртуальных машин.

2.3 Балансировка виртуальных машин

Данный модуль позволяет создавать собственные реализации балансировки, либо воспользоваться реализацией на основе среднеквадратичного отклонения. Рассмотрим ее алгоритм поведения.

Балансировщик получает из Nova Conductor текущие значения загруженности всех узлов и их виртуальных машин. На основе полученных данных проводится симуляция перемещения каждой виртуальной машины на каждый узел (кроме исходного для виртуальной машины узла), в ходе которой меняются показатели загруженности для определения нового вероятного среднеквадратичного отклонения. Результаты симуляции записывается в список в виде словаря {узел, виртуальная машина, отклонение}. Данный список, по завершении всех симуляций, сортируется по возрастанию с ключом среднеквадратичного отклонения. Полученный список характеризует все возможные случаи перемещений виртуальных машин, из которых нас интересуют те, что сводят отклонение к минимуму. Взятая пара {узел, виртуальная машина} должна пройти фильтрацию сервисом Nova Scheduler. Фильтры сервиса Nova Scheduler позволяют определить, возможно ли запустить виртуальную машину на заданном узле. В случае, если один из фильтров не завершился успешно, пара {узел, виртуальная машина} пропускает дальнейшую фильтрацию и отбрасывается.

В случае, если пара прошла фильтрацию, вызывается метод *live-migration,* перемещающий виртуальную машину с исходного узла на целевой. В ходе миграции создается новый домен в libvirt, в который копируется содержимое оперативной памяти виртуальной машины. Когда копирование близко к завершению, гипервизор приостанавливает работу виртуальной машины, копирует оставшиеся данные и активирует домен виртуальной машины, возобновляя ее работу. По окончанию миграции, модуль балансировки прекращает свою работу до следующего вызова в случае перегрузки.

Если пара сервисом Nova Scheduler не прошла фильтрацию, из списка берется следующая пара.

Подобная реализация позволяет поэтапно стабилизировать нагрузку в кластере до тех пор, пока среднеквадратичное отклонение по узлам будет

ниже порогового значения. Изменение порогового значения СКО позволяет регулировать “чувствительность” балансировщика к перегрузкам.

2.4 Повышение энергоэффективности кластера

Существуют ситуации, когда не все вычислительные узлы кластера достаточно загружены. Т.к. вопрос энергопотребления для вычислительного центра стоит особенно остро, необходимо выработать методики сокращения энергопотребления. Предлагается следующая методика: определять слабонагруженные узлы, мигрировать с них виртуальные машины и приостанавливать работу узлов, помещая их в состояние ACPI S3 (“Ждущий режим”). Рассмотрим этапы поподробнее.

Как было замечено ранее, в случае, если балансировщик не определил перегрузки в кластере, будет проверена достаточность загруженности каждого вычислительного узла в кластере. Для этого устанавливаются пороговые значения в конфигурационном файле. Если показатели загруженности узла не превышают порог, принимается решение о необходимости введения узла в ждущий режим.

Клиент, подписывающий соглашение об уровне предоставления услуг (SLA) с поставщиком облачных решений, рассчитывает на определенный уровень доступности его виртуальных машин. Таким образом, перед переводом узла в ждущий режим, требуется комплекс мер по обеспечению сохранности доступа к виртуальным машинам.

Перевод узла в ждущий режим включает в себя следующие действия:

- Установка значения “suspending” полю “suspend_state” таблицы “compute_nodes” для узла, вводимого в ждущий режим.
- Попытка миграции всех виртуальных машин с узла. На данном этапе действует фильтр “max_migrations”, ограничивающий количество одновременных миграций с узла. Если не удается мигрировать одну из виртуальных машин в результате действия других фильтров, в поле “suspend_state” записывается значение “active” и перевод узла в ждущий режим прекращается. Это значит, что есть минимум одна виртуальная машина, для которой подходящее окружение находится только на текущем узле.
- Если все виртуальные машины были перемещены с “suspending”-узла, балансировщик запрашивает у узла MAC-адрес интерфейса, по которому будет возможно “разбудить” узел; записывает адрес в поле “mac_to_wake” таблицы “compute_nodes” и ставит флаг g параметру wol сетевого интерфейса. После данной подготовки, узлу посылается команда “systemctl suspend” (для RHEL/CentOS), что переводит узел в состояние ACPI S3. Узел в базе данных Nova помечается как suspended.

Для корректной работы необходим также suspend-фильтр, не разрешающий балансировщику взаимодействовать с узлами, которые находятся в состояниях suspended или suspending.

2.5 Управление балансировкой

Ввиду того, что в вычислительный кластер зачастую входят узлы, конфигурации которых разнятся, запускать балансировку на определенных узлах может быть излишним. OpenStack позволяет объединять узлы в различные группы по функциональным и логическим признакам [2]. Группировка по функциональным признакам «Host Aggregates» позволяет разграничить узлы в соответствии с ресурсными различиями. Группировка по логическим признакам «Availability Zones» позволяет физически разграничить узлы по местоположению в вычислительных центрах. Предлагается ввести систему правил в сервис балансировки для определения набора хостов, на которых можно разрешить или запретить его работу.

Правило является словарем вида {тип, регулярное выражение, действие}, где:

- “Тип” – логическая единица, к которой применяется правило. Допустимые значения: «host», «az», «ha».
- “Регулярное выражение” – шаблон, по которому проверяется принадлежность логической единицы к правилу.
- “Действие” – разрешить или запретить балансирование данной логической единицы.

Правила “читаются” системой в порядке следования, формируя в конце список из разрешенных узлов. Узлы из полученного списка участвуют в балансировке.

3. Экспериментальные результаты

В результате лабораторного тестирования были взяты замеры среднеквадратичного отклонения загруженности трех вычислительных узлов, между которыми балансировались виртуальные машины. Граница отклонения установлена на отметке 7%. Результаты взятия до и после включения балансирования представлены на рис. 2.

Рис. 2. Среднеквадратичное отклонение загруженности узлов.

Резкое снижение отклонения с 13.23% до 5.92% свидетельствует о перемещении той виртуальной машины, которая сводит отклонение к минимуму. Последующие колебания являются следствием изменений нагрузки на узлах.

Девид Мейзнер [3] утверждает, что энергопотребление блейд-сервера при полной мощности составляет 450 Вт, при средней загруженности 270 Вт и в ждущем режиме 10.4 Вт. Перевод двух узлов в ждущий режим в кластере, состоящем из 10 средненагруженных вычислительных узлов, позволит достичь 19.23% снижения энергопотребления.

4. Заключение

Разработанный сервис OpenStack Nova LoadBalancer показал практический результат в балансировании загруженности кластера. Архитектура сервиса построена таким образом, чтобы для решения задач была возможность использовать различные средства. Полученный опыт и знания позволят разрабатывать новые теоретические и практические подходы в решении задачи балансировки. Страница [4] сервиса на Launchpad позволит следить за ходом разработки и стать участником проекта.

Список литературы

[1]. OpenStack Overview. http://www.openstack.org/software/
[2]. OpenStack Scailing. http://docs.openstack.org/openstack-ops/content/scaling.html
[3]. Meisner D, Gold B, Wenisch T. PowerNap: eliminating server idle power. ACM SIGPLAN Notices 2009; 44(3): 205–216.
[4]. OpenStack Compute Load Balancer. https://launchpad.net/nova-loadbalancer

Dynamic Optimization of Workload on Compute Nodes in Private, Public and Hybrid Clouds

A. Chadin, <a.chadin@servionica.ru>
Servionica, 15 Kedrova Str., Moscow, Russian Federation

Abstract. Cluster's overload is a complex problem and can be solved by using one of stabilization methods: by migrating VMs from one node to another in case of compute nodes. The main goal of this research is to create nova load-balancer service that manages cluster loads and makes decisions of stabilization based on statistics data. This system is designed for automatic workload distribution in a cluster by analyzing workload of compute nodes and migrating VMs from overloaded nodes to underloaded ones. It runs periodically and checks latest statistics to compare with threshold values. If one of the statistics values is greater than the threshold value, the nova load-balancer runs cluster stabilization process to reduce load on cluster by performing live-migration of VMs. Besides workload stabilization, the system also provides an opportunity to reduce power consumption by unloading and suspending underloaded nodes. If statistics values of node are less than underload-threshold values, then all VMs of current node will be migrated to other nodes. When there are no VMs on the node, LB marks this node as suspended and is going to suspended mode (ACPI S3). If nova load-balancer wants to activate suspended nodes to reduce overall cluster load, it uses WOL technology. Workload stabilization in a cluster increases stability and reduces system response time.

Keywords: Balancing; Cluster; Cloud Computing

DOI: 10.15514/ISPRAS-2015-27(6)-19

For citation: Chadin A. Dynamic Optimization of Workload on Compute Nodes in Private, Public and Hybrid Clouds. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 307-314 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-19

References

[1]. OpenStack Overview. http://www.openstack.org/software/
[2]. OpenStack Scailing. http://docs.openstack.org/openstack-ops/content/scaling.html
[3]. Meisner D, Gold B, Wenisch T. PowerNap: eliminating server idle power. ACM SIGPLAN Notices 2009; 44(3): 205–216.
[4]. OpenStack Compute Load Balancer. https://launchpad.net/nova-loadbalancer

Обработка больших объемов сырых астрономических данных с помощью модели вычислений MapReduce[1]

[1]С.В. Герасимов < gerasimov@mlab.cs.msu.su >
[2]А.В. Мещеряков < mesch@iki.rssi.ru >
[1]И.Ю. Колосов < zackwag32@gmail.com >
[1]Е.С. Глотов < glot.unltd@gmail.com >
[1]И.С. Попов < ivan@mlab.cs.msu.su >
[1] Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1., стр. 52, факультет ВМК
[2] Институт космических исследований РАН, 117342, Россия, г. Москва, Профсоюзная ул., 84/32

Аннотация. Экспоненциальный рост объемов, повышение качества данных в современных и будущих обзорах неба открывают перед астрофизиками новые горизонты, однако требуют применения новых подходов к их обработке, а именно технологий больших данных и облачных вычислений. В работе предлагается подход, основанный на модели MapReduce, для решения одной из самых масштабных и важных вычислительных задач астрофизики — задачи обработки сырых данных астрономических изображений.

Ключевые слова: MapReduce; Hadoop; небесный обзор; большие данные; облачные вычисления; обработка изображений

DOI: 10.15514/ISPRAS-2015-27(6)-20

Для цитирования: Герасимов С.В., Мещеряков А.В., Колосов И.Ю., Глотов Е.С., Попов И.С. Обработка больших объемов сырых астрономических данных с помощью модели вычислений MapReduce. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 315-334. DOI: 10.15514/ISPRAS-2015-27(6)-20.

1. Введение

Развитие наблюдательной астрономии на современном этапе характеризуется взрывным ростом объема данных, получаемых телескопами в рамках программ *небесных обзоров* — больших наборов цифровых фотографий неба в заданном спектральном диапазоне (фильтре), покрывающих большую область

[1] Работа поддержана грантом РФФИ №15-29-07085 офи_м

неба. Крупнейший проект - Слоановский цифровой обзор неба (англ., SDSS) [1] — проводился в 1998-2009 годах и был завершен в январе 2011 года выпуском официального релиза и размещением всех данных обзора в публичном доступе. Бурное развитие цифровых приемников излучения (ПЗС-матриц) за последние 30 лет [2] к настоящему времени сделало возможным за одно наблюдение на современном оптическом телескопе с большим полем зрения получать широкоформатную фотографию в заданном фильтре участка ночного неба площадью несколько квадратных градусов (1 снимок неба ~10^9 пикселов имеет объем ~1ГБ для современных обзорных телескопов Subaru-HSC [3], DES [4], PanSTARRS [5]). За одну ночь оптический телескоп может делать до тысячи фотографий неба, обеспечивая поток научных данных около 1ТБ/сутки. Скорость получения астрономических данных будет продолжать экспоненциально расти вместе с вводом в строй обзорных телескопов следующего поколения, таких как Большой синоптический обзорный телескоп (англ. LSST) [6], [7], который должен быть построен к 2019 году. При потоке научных данных ~15ТБ/сутки ожидаемый объем всех астрономических изображений, полученных одним телескопом LSST за 10 лет работы, составляет 114 петабайт.

В настоящее время объем астрономических изображений, хранящихся в открытых архивах в центрах обработки данных обсерваторий по всему миру, составляет несколько петабайт, это сотни миллионов астрономических изображений.

Рост объемов данных наблюдений, повышение качества астрономических данных открывает перед астрофизиками новые горизонты, однако требует применения новых современных инженерных и математических подходов к их обработке, среди которых *технологии больших данных*, *облачные вычисления*.

Для извлечения данных о небесных объектах, содержащихся в больших массивах сырых изображений, полученных цифровыми камерами телескопов, используется цепочка преобразований, называемая *конвейер для обработки астрономических изображений* (далее просто *конвейер, англ., pipeline*). Финальной целью работы конвейера является получение *каталога небесных объектов* — таблицы чисел, содержащей значения свойств объектов, обнаруженных на изображениях.

Типичный конвейер по обработке сырых изображений, полученных телескопами, состоит из следующих этапов.

- *Первичная обработка изображений* с телескопа включает в себя вычитание шума считывания, коррекцию неравномерности чувствительности ПЗС-матрицы, удаление горячих пикселов и другие этапы предварительной обработки изображений, которые делаются в обсерватории перед тем, как данные попадут в архив.
- *Астрометрическая калибровка изображений*. На каждом изображении детектируются все яркие объекты, и карта этих объектов

сравнивается с картой той области неба, куда был направлен телескоп в момент наблюдения. В результате их сравнения строится система мировых координат на изображении (WCS, англ. World Coordinate System).

- *Фотометрическая калибровка изображения*. Величина яркости всех источников на изображении калибруется относительно яркости стандартных звезд.

Перечисленные выше этапы обработки, как правило, уже сделаны для изображений, входящих в состав официальных релизов астрономических данных небесных обзоров (в частности, обзора SDSS, с данными которого мы работаем в данной статье). Далее любой астрономический конвейер должен реализовывать два основных этапа, это:

- *Объединение изображений*. Каждый небесный обзор состоит из множества пересекающихся изображений — проекций небольших участков неба на плоскость ПЗС-матрицы телескопа в моменты наблюдений. Из-за движения Земли и телескопа, меняется ориентация ПЗС-матрицы в пространстве, и каждый новый кадр, полученный телескопом, имеет свои параметры проекции (которые определяются на этапе астрометрической калибровки изображения). В процессе объединения изображений все доступные кадры в заданном фильтре и для данного участка неба (по-разному ориентированные друг относительно друга в системе мировых координат) объединяются в одно большое изображение в заданной проекции. Основные подэтапы объединения изображений: (i) проецирование всех кадров в общую систему координат, (ii) удаление из изображений фоновой компоненты и (iii) сложение кадров для получения объединенной картинки заданного участка неба. Объединенное изображение обладает важными преимуществами по сравнению с каждым отдельным кадром: (а) на нем видно больше слабых объектов (сложение нескольких кадров увеличивает глубину астрономического изображения), (б) свойства объектов могут быть измерены более детально за счет увеличения динамического диапазона объединенного изображения, (в) свойства протяженных объектов (сравнимых с размером кадра) могут быть точно измерены на большом изображении, составленным из нескольких кадров и, наконец, (г) объединение изображений заметно сокращает размер астрономических данных без потери информации, что уменьшает затраты на их дальнейшую обработку.
- *Создание астрономического каталога*. Каталог представляет собой таблицу всех небесных объектов, обнаруженных на астрономических изображениях с фиксированным набором свойств, “измеренных” для каждого объекта. Основные подэтапы создания каталога для каждого изображения: (i) вычитание фоновой компоненты из изображения, (ii)

создание маски “плохих” областей на изображении (например, области вокруг ярких звезд), где данные по объектам сильно искажены, и исключение этих областей из дальнейшего анализа, (iii) поиск (детектирование) групп объектов на изображении, (iv) удаление отдельных артефактов (“искусственных” объектов, таких как следы от космических лучей, самолетов и спутников, блики от звезд и т.д.) из общего списка групп объектов, (v) пространственное разделение объектов внутри каждой из групп (деблендирование), (vi) экстракция разнообразных свойств (характеристик яркости, размера, формы, морфологии) для каждого объекта на изображении, (vii) классификация объектов на точечные/протяженные (звезды/галактики) на основе их свойств, (viii) измерение профилей ярких звезд на изображении и построение эмпирической модели функции отклика на точечный источник (англ. PSF), (ix) “измерение” свойств объектов, скорректированных моделью PSF[2], (x) сохранение всех свойств объектов в виде таблицы-каталога и объединение каталогов по всем изображениям.

Последовательность, набор этапов, качество реализации каждого из этапов, перечень “измеряемых” свойств могут варьироваться в зависимости от реализации конвейера. Как правило, конвейеры реализуются в рамках проектов небесных обзоров, и астрофизики имеют возможность либо скачать и самостоятельно обработать сырые данные (изображения), либо воспользоваться готовым каталогом соответствующего обзора и никоим образом не могут повлиять на алгоритмы конвейера, например, оптимизировать один из его этапов или “измерить” новую характеристику небесных объектов на всех изображениях и добавить ее в каталог.

Если построение своего конвейера для обработки изображений небольшого объема (несколько ГБ) на персональном компьютере еще представляется возможным, например, с помощью распространенных астрономических пакетов SWarp, SExtractor и PSFEx [8], то настраиваемая обработка больших подвыборок (от нескольких ТБ) сырых изображений современных небесных обзоров и архивов астрономических обсерваторий практически невозможна. В связи со взрывным ростом объемов данных в небесных обзорах следующего поколения (таких как PanSTARRS, LSST) ситуация будет резко усугубляться.

Необходимо отметить, что от алгоритмической базы конвейера напрямую зависит *качество данных в каталогах*, с которыми большинство астрофизиков

[2] Земная атмосфера и оптика телескопа искажают изображения небесных объектов — точечный источник на картинке с телескопа имеет конечный размер и сложную форму, которая меняется даже внутри одного изображения. Цель построения пространственной модели PSF на каждом изображении — свести к минимуму влияние атмосферы и телескопа на “измеряемые” свойства объектов.

будет работать в своих моделях и на которых будут проверяться научные гипотезы. Группы астрофизиков, работающие в таких предметных областях, как наблюдательная космология, физика галактик, звездная астрономия, исследование астероидов, могут быть заинтересованы в максимальной оптимизации тех (зачастую различных) элементов конвейера, которые наиболее критичны для решения их задач.

Цель настоящей работы — исследование и разработка горизонтально масштабируемого *конвейера для обработки астрономических изображений*, доступного для применения коллективами астрофизиков для пакетной обработки больших объемов сырых данных современных и будущих небесных обзоров, а также данных изображений архивов обсерваторий. Архитектура конвейера должна предоставлять широкие возможности по настройке и модификации алгоритмов, используемых на всех его этапах, включая возможность добавления возможности “измерения” новых свойств небесных объектов.

Одним из ключевых требований является легкость в развертывании на недорогих вычислительных мощностях астрономических групп разного масштаба или в облаке, отсутствие необходимости в привлечении высококвалифицированных программистов и системных администраторов. Как следствие, разработанный конвейер обладает рядом уникальных функциональных и нефункциональных возможностей:

- настраиваемая алгоритмическая база этапов конвейера;
- горизонтальная масштабируемость;
- легкость развертывания на ”дешевом” кластере и в облаке;
- ряд оптимизированных шагов обработки, в т.ч. корректная обработка протяженных объектов (часто отсутствует в современных конвейерах).

2. Обзор существующих решений по параллельным конвейерам

В работе [9] представлен способ параллельного выполнения одного из шагов конвейера — объединения изображений. Объединение производится для небольшого (градус или его доли) целевого фрагмента неба и заданного цветового фильтра с использованием инфраструктуры Apache Hadoop [10]. Алгоритмы объединения реализованы авторами непосредственно в процедурах map и reduce. В качестве источника изображений используется обзор SDSS. Основное внимание в работе уделено повышению общей производительности обработки за счет эффективной фильтрации изображений, входящих в небольшой по размеру искомый участок неба с помощью предварительной фильтрации имен FITS-файлов (Flexible Image Transport System - формат, совмещающий растровое астрономическое изображение и мета-информацию о нем) регулярными выражениями либо

использовании информации из СУБД SDSS, позволяющей сформировать набор изображений, соответствующих целевому фрагменту неба.

Проект Montage [11] представляет собой известную open-source реализацию конвейера для обработки больших объемов сырых изображений. Архитектура Montage основывается на MPI [12] и Pegasus [13] — фреймворке для отображения графа научных задач на вычислительные ресурсы HTC (англ. High-Throughput Computing). Montage реализует следующие этапы параллельной обработки изображений:

- проецирование входных изображений;
- удаление фоновой компоненты;
- создание мозаики (объединенного изображения).

Причем для самого вычислительно сложного этапа обработки — проецирования изображений — дано несколько оптимизированных версий.

В работе [14] реализован конвейер на основе MapReduce [15] по получению каталога из изображений обзора SDSS. В качестве базового астрофизического ПО использовался SExtractor. Вычисления проводились на локальном кластере из 54 узлов (432 ядер).

В цепочку обработки для каждого изображения вошли следующие этапы:

- вычитание фона;
- поиск источников;
- деблендирование;
- измерение свойств объектов без коррекции на PSF;
- таблицы объектов объединялись в общий каталог.

В работе [16] реализован конвейер по получению каталога из изображений в архиве Национальной оптической астрономической обсерватории (англ., NOAO) из обзора DES (30ТБ данных изображений). Вычисления проводились на кластере Darwin Кембриджского университета и на локальном кластере, состоящем из 8 узлов / 96 ядер. В качестве базового астрофизического ПО использовались: SExtractor и PSFEx. В цепочку обработки для каждого изображения вошли следующие этапы:

- вычитание фона;
- поиск источников;
- деблендирование;
- отбор звезд и построение модели PSF;
- измерение свойств объектов с коррекцией на PSF;
- этап фотометрической калибровки в настоящее время отсутствует для данных обзора DES, поэтому авторы провели его самостоятельно.

Обработка 30ТБ данных изображений заняла у авторов около 15000 процессорных часов.

3. Предложенное решение

3.1 Базовые технологии распределенного хранения и пакетной обработки

В качестве основы архитектуры разработанного параллельного конвейера используются *распределенная файловая система HDFS* и *модель вычислений MapReduce*, реализованная в *Apache Hadoop*. Этот выбор обусловлен несколькими факторами.

1. Концепция планирования вычислений на основе *локальности данных* хорошо “ложится” на разновидности задач классического конвейера по обработке сырых данных небесных обзоров:

- обработка отдельного изображения, результатом которого является новое изображение (проецирование, удаление фоновой компоненты);
- обработка нескольких изображений для получения объединенного изображения;
- извлечение набора объектов и их свойств из отдельного изображения.

2. Apache Hadoop обеспечивает горизонтальное масштабирование, восстановление процесса вычислений после сбоев, надежность хранения данных, что необходимо для хранения и пакетной обработки больших массивов данных.

3. Apache Hadoop прост в администрировании и использовании, поддерживается популярными *облачными сервисами*, например, Microsoft Azure HDInsight, Amazon EMR. Существуют сторонние дистрибутивы Apache Hadoop, например, Cloudera и Hortonworks, предоставляющие дружественные пользовательские интерфейсы для разворачивания и мониторинга кластера Hadoop.

4. Стек Hadoop включает в себя набор технологий, например, индексированные хранилища данных Apache HBase[17] и Hive[18], которые могут быть использованы на этапах обработки данных, следующих за обработкой изображений: систематизация данных небесных каталогов больших объемов, объединение данных из нескольких небесных обзоров, обработка запросов пользователей к данным каталогов.

5. Выбранные базовые технологии позволяют осуществить интеграцию с in-memory MapReduce платформами (например, Apache Spark[19]) для эффективной итеративной обработки данных больших объемов алгоритмами машинного обучения.

3.2 Реализация базовых алгоритмов работы конвейера

Для реализации базовых алгоритмов работы конвейера используются де-факто признанные в мире астрофизиков пакеты ПО:

- SWarp (проецирование, измерение и удаление фоновой компоненты из изображений, объединение изображений);
- SExtractor (измерение и удаление фоновой компоненты, детектирование объектов на изображениях, классификация на точечные/протяженные и последующее “измерение” свойств объектов, в том числе исправленных на PSF, — при наличии готовых моделей функции отклика, построенных в PSFEx);
- PSFEx (определение модели инструментальной функции отклика на основе свойств точечных объектов, “измеренных” SExtractor).

Перечисленные пакеты программ могут быть эффективно использованы на отдельных персональных компьютерах с многоядерными процессорами для обработки небольших объемов изображений. К примеру, SWarp может совместить на современном персональном компьютере с использованием двух рабочих нитей 17 изображений общим размером 180 МБ за 4 минуты 41 секунду. Указанные пакеты ПО свободно лицензируется (GPL), написаны на языке Си и обладают продуманной модульной структурой, упрощающей их доработку. Все перечисленные утилиты имеют файловый интерфейс входных/выходных данных, настройки обработки задаются в виде конфигурационных файлов. Следует отметить, что имеется возможность *ограничивать данные пакеты ПО по использованию ресурсов*: количеству нитей, объему ОЗУ и жестокого диска.

3.3 Реализация параллельного конвейера

Целевой участок неба, для которого выполняется обработка, разделяется на прямоугольные *клетки* с фиксированными сторонами. Клетки нумеруются двумя индексами, соответствующими строке и столбцу (рис. 1).

Рис.1. Покрытие исходных изображений "клетками".

На 1-м этапе работы предложенного алгоритма параллельной конвейеризации сырых изображений (рис. 2), хранящихся в виде файлов в формате FITS, производятся операции фильтрации изображений по попаданию в целевой участок неба, а также удаление фона, проецирование, объединение изображений с помощью пакета SWarp. Все перечисленные операции кроме объединения проводятся независимо над каждым исходным изображением и могут быть распараллелены в рамках map задач с задействованием локальности данных. Функция map:

- пропускает входное изображение, если оно не попадает в целевую область;
- в случае попадания модифицирует изображение и возвращает номер клетки, к которой принадлежит изображение и само модифицированное изображение в качестве пары ключ-значение.

Следует отметить, что изображение может пересекаться сразу с несколькими клетками, в этом случае map вернет соответствующее число пар: номер клетки, модифицированное изображение.

В reduce попадают изображения, принадлежащие одной клетке. Над ними производится операция объединения.

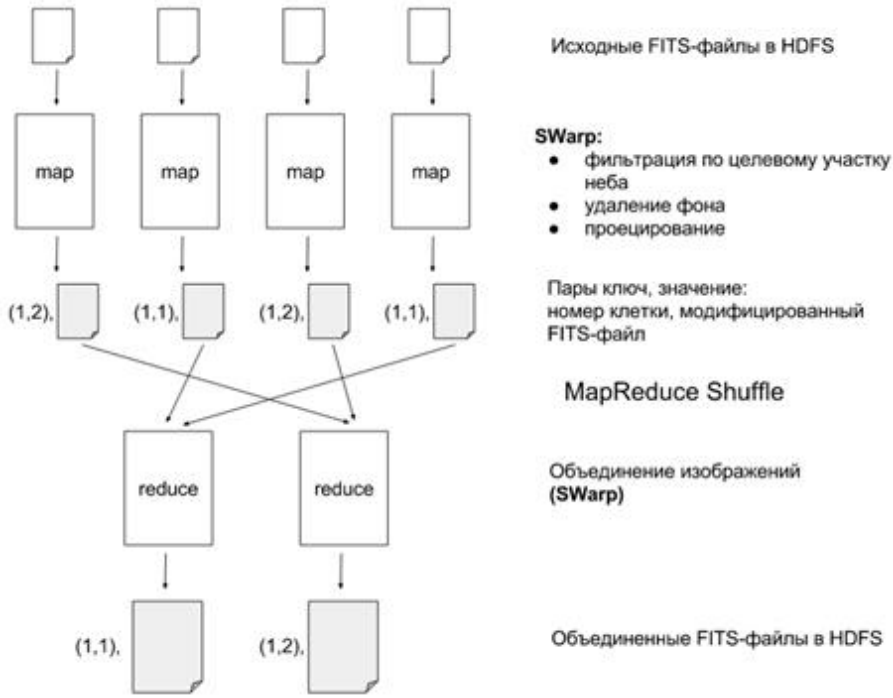


Рис.2. 1-й этап работы параллельного конвейера.

На 2-м этапе параллельной обработки производится извлечение небесных объектов и их свойств из изображений-клеток. Для этого в каждой map функции:

- осуществляется обнаружение объектов на изображениях, классификация точечных источников и извлечение свойств с помощью SExtractor;
- на основе свойств точечных объектов с помощью пакета PSFEx строится пространственная модель PSF;
- SExtractor осуществляет “измерение” свойств каждого объекта с учетом PSF-модели.

Рис.3. 2-й этап работы параллельного конвейера.

Следует отметить, что для корректной обработки на 2-м этапе больших объектов на краю клетки, уже на 1-м этапе клетки формируются внахлест за счет дополнительной *рамки*, ширина которой больше, чем характерный размер объектов, свойства которых мы хотим надежно “измерять”. При этом обработка и измерение свойств объектов будут производиться корректно (объект на краю клетки не будет обработан дважды и “по частям”): каждый объект будет обработан в пределах единственной клетки. Объект, находящийся на границе, будет обрабатываться в той клетке, на стороне которой он находится относительно границы клеток без учета рамок (рис.4).

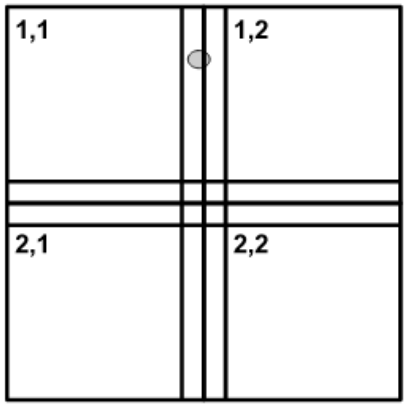


Рис.4. Обработка объекта на границе клеток. Данный объект попадет в обрамленную клетку 1,1.

Для встраивания пакетов SWarp, SExtractor и PSFEx использовался интерфейс локальных файлов, создаваемых во временных директориях узлов. В будущем планируется адаптировать интерфейсы пакетов для использования стандартных потоков ввода, вывода. Следует отметить, что благодаря встроенной возможности ограничения ресурсов, используемых перечисленными пакетами, в особенности используемого объема ОЗУ, стала возможной мотивированная настройка параметров контейнеров YARN, необходимых map и reduce задачам (см. следующий раздел).

4. Эксперименты

В качестве данных для экспериментов использовалось подмножество сырых изображений красного фильтра набора *Stripe82* небесного обзора SDSS DR12. SDSS камера (рис.5) представлена 6 вертикальными рядами ПЗС-матриц, фиксирующих излучение в одном из 5 фильтров видимого спектра волн (условно названных r, i, u, z, g). Вертикальные ряды ПЗС-матриц имеют “слепые” зоны между собой. Снятие изображений непрерывных полос неба обеспечивается за счет перемещения небесных объектов “по вертикали” (вдоль рядов ПЗС-матриц) благодаря движению Земли. Каждый сеанс снятия изображений называется *проходом (англ., run)*. Для устранения “слепых” вертикальных зон обычно выполняется два прохода, второй со смещением камеры по “горизонтали” (эти проходы называются северный и южный).

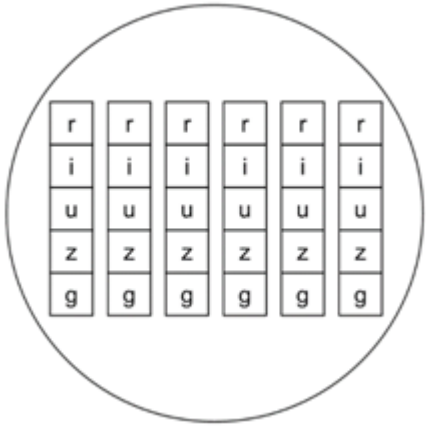


Рис. 5. Камера SDSS

Эксперименты проводилась в облачной инфраструктуре Azure[3] на сборке Hadoop 2.6.0 HDInsight. Сырые изображения, полученные с сервера архива проекта SDSS, были предварительно преобразованы в формат SequenceFile.

[3] Предоставлена компанией Microsoft в виде гранта по программе “Azure for Research”

Дело в том, что модель MapReduce эффективна в том случае, когда фрагмент данных, обрабатываемый одним Mapper (англ. split), по объему не сильно меньше размера блока HDFS. На сегодняшний момент с учетом характеристик существующих жестких дисков наиболее распространенным размером блока HDFS является 128МБ. А каждый исходный FITS-файл занимает около 12 МБ. Поэтому для эффективной обработки исходные файлы были преобразованы в файлы в формате SequenceFile, позволяющие разбивать их на логические части, близкие к размеру блока.

Эксперименты проводились на кластере HDInsight, построенном на узлах D12 из линейки типов узлов Azure, обладающих следующими характеристиками:

- число ядер: 4
- объем ОЗУ: 28 ГБ

Разработанный экспериментальный образец написан на Java и запускается с помощью командной строки, поддерживающей следующие опции:

- RA, DEC — координаты центра запрашиваемого участка в экваториальной системе координат выраженные в градусах. RA — аббревиатура от англ. Right Ascension (прямое восхождение), DEC — аббревиатура от англ. Declination (склонение). Для этих координат также используются обозначения α, δ.
- RA_SPAN, DEC_SPAN — ширина запрашиваемого участка по обеим координатам.
- RA_OVERLAP, DEC_OVERLAP — ширина перекрытия соседних клеток, на которые делится участок;
- RA_SUBREGIONS, DEC_SUBREGIONS — число клеток, на которые будет разбиваться участок вдоль обеих координат. Эти параметры позволяют задавать размеры одной клетки.
- FILTER — диапазон спектра, изображения в котором будут совмещаться. Один из пяти диапазонов, присутствующих в небесном обзоре SDSS DR12, обозначаемых буквами u, g, r, i и z.
- PIXEL_SCALE — размер одного пикселя выходного изображения в арксекундах. Например, в изображениях небесного обзора SDSS размер пикселя составляет 0.396 арксекунд.

Из всех указанных параметров на время работы прототипа непосредственно влияют те, которые задают *размер целевого участка неба* и *размеры клеток*. Число клеток выбиралось таким образом, чтобы размер одной клетки составлял примерно 0.7 градуса. В соответствии с этим был выбран участок неба с центром в точке с координатами α=51.5912 град., δ=0.0131 град. и размером 30 градусов по α и 2.5 градуса по δ. Размер по δ выбран так, чтобы участок вмещал в себя изображения, полученные всеми 6 столбцами ПЗС-камер телескопа (см. рис. 5). Участок разбивался на 60 частей по α и на 5 частей по δ, а перекрытие клеток (рамка) было выбрано равным 0.2 градусам,

таким образом, каждая клетка имела размеры 0.7 x 0.7 градусов и перекрывалась с соседними клетками.

Помимо параметров самого прототипа на время работы также влияют настройки MapReduce, в частности, размер фрагмента данных, подаваемого на вход одной задаче отображения (англ., map), и число задач свертки (англ., reduce). В ходе экспериментов число задач свертки выбиралось равным числу клеток. Размер фрагмента данных составил 128 МБ.

Пакет SWarp, лежащий в основе реализации, также имеет свои настройки, влияющие на производительность. Так, он позволяет задавать число рабочих нитей, которое бралось равным 2, и размер буфера для совмещения изображений в памяти. Использовался буфер размером в 1 ГБ.

Настройки кластера, на котором производились эксперименты, предусматривали 3 ГБ памяти для задач отображения и 5 ГБ памяти для задач свертки. Настройки, задающие максимальный размер кучи, выделяемой виртуальной машине Java (JVM), выбирались так, чтобы соответствовать этим значениям.

В приведенных ниже таблицах указан объем данных, занимаемых входными изображениями, общее время выполнения, а также время выполнения стадий шага свертки. Этих стадий три. Первая стадия, стадия перемешивания (англ., shuffle), включает в себя передачу вывода шага отображения задачам свертки. Стадия перемешивания включает в себя передачу данных по сети. Вторая стадия, стадия сортировки (англ., sort) сортирует ключи и значения, переданные задаче свертки. Благодаря сортировке значения разбиваются на группы. Наконец, следует стадия свертки (англ., reduce), на которой значения каждой группы сворачиваются и вывод записывается в распределенную файловую систему. Свертка в Hadoop может начинаться до того, как завершится отображение, поэтому общее время выполнения, вообще говоря, не является суммой времени выполнения шагов отображения и свертки. Временные показатели работы 1-го этапа обработки представлены в табл.1 и 2.

Табл.1. Результаты 1-го этапа обработки на 6 рабочих узлах

Объем данных	Время выполнения в минутах				
	Общее	Отображение	Перемешивание	Сортировка	Свертка
14 ГБ	34	19	10	1	14
21 ГБ	60	33	17	1	24
33 ГБ	79	45	24	1	31

Табл.2. Результаты 1-го этапа обработки на 12 рабочих узлах

Объем данных	Время выполнения в минутах				
	Общее	Отображение	Перемешивание	Сортировка	Свертка
14 ГБ	27	16	17	1	10
21 ГБ	33	17	17	1	14
33 ГБ	49	23	27	1	21

Рассматривая полученные результаты, стоит отметить, что затраты времени на шаге свертки в основном складываются из затрат на перемешивание результатов отображения и затрат на собственно свертку, то есть совмещение изображений. Время, требуемое на перемешивание, ожидаемо растет с увеличением объема данных. При этом затраты на перемешивание увеличиваются при использовании двенадцати рабочих узлов против изначальных шести, что можно объяснить издержками на передачу данных по сети.

В экспериментах по 2-му этапу обработки получены следующие результаты по производительности (табл.3, табл.4).

Табл.3. Результаты 2-го этапа обработки на 6 рабочих узлах

Объём данных	Время выполнения
14 ГБ	3 минуты 27 секунд
21 ГБ	3 минуты 47 секунд
33 ГБ	6 минут 6 секунд

Табл.4. Результаты 2-го этапа обработки на 12 рабочих узлах

Объём данных	Время выполнения
14 ГБ	1 минута 54 секунды
21 ГБ	2 минуты 55 секунд
33 ГБ	4 минуты 11 секунд

Графики на рис.6 иллюстрируют масштабируемость работы экспериментального образца.

По 1-му этапу обработки были проведены эксперименты на бóльших объемах данных (для 12 узлов кластера), результаты которых показаны в виде графика на рис. 7.

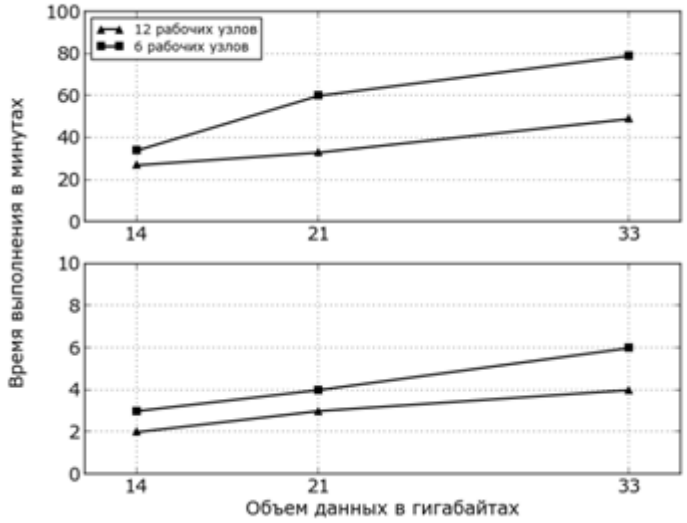


Рис 6. Время работы 1-го (сверху) и 2-го (снизу) этапов обработки в зависимости от количества узлов в кластере и объемов данных.

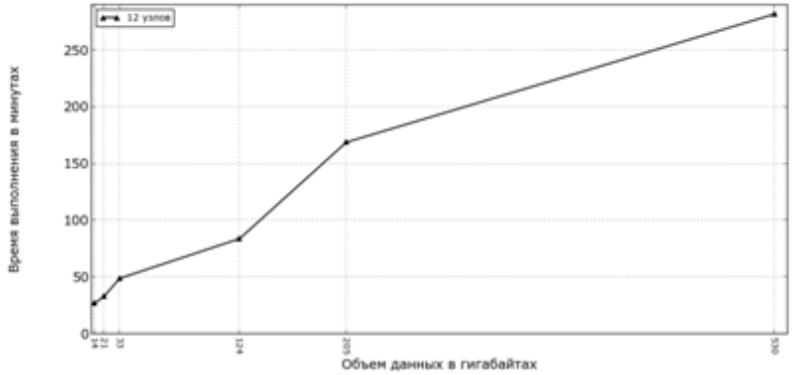


Рис. 7. Время работы 1-го этапа в зависимости от объема данных на 12 узлах

5. Вывод

Проведенные эксперименты подтвердили целесообразность применения предложенной архитектуры параллельного конвейера, основанной на MapReduce, для обработки наборов астрономических изображений, объемы которых находятся в пределах нескольких десятков ГБ. Легко доступная и дешевая инфраструктура Hadoop позволяет астрофизику заменить собой персональный компьютер в разы ускоряя вычисления (последовательная обработка 33ГБ данных из экспериментов заняла бы более 12 часов на персональном компьютере с 2 ядрами). Для апробации метода на бóльших объемах данных (5-20 ТБ) запланированы дополнительные эксперименты на обзорах неба SDSS и DES. Эксперименты позволят уточнить эффективность масштабирования и работоспособность решения на объемах изображений, соответствующих современным небесным обзорам.

Также планируется реализация ряда оптимизаций, в частности:

- оптимизация интерфейсов взаимодействия пакетов SWarp, SExtractor, PSFEx и инфраструктуры Hadoop;
- проработка механизма, позволяющего интегрировать в цепочку обработки свои реализации любых шагов конвейера и апробация механизма на оптимизированных версиях нескольких алгоритмов, в частности:
    - вычитание фоновой компоненты, основанное на нескольких соседних исходных изображениях (на 1-м этапе обработки);
- добавление возможности расширения набора “измеряемых” свойств небесных объектов (на 2-м этапе обработки).

Литература

[1]. The Sloan Digital Sky Survey (SDSS) http://www.sdss.org/
[2]. Burke B., Gregory J., Cooper M., Loomis A., Young D., Lind T., Doherty P., Daniels P., Landers D., Ciampi J., Johnson K., O’Brien P. CCD Imager Development for Astronomy. Lincoln Laboratory Journal, 2007, Volume 16, Number 2
[3]. Subaru-HSC http://www.naoj.org/Projects/HSC/
[4]. The Dark Energy Sky Survey (DES) http://www.darkenergysurvey.org/
[5]. Pan-STARRS http://pan-starrs.ifa.hawaii.edu/public/
[6]. The Large Synoptic Survey Telescope (LSST) http://www.lsst.org/
[7]. Zhang Y., Zhao Y. Astronomy in the Big Data Era. Data Science Journal, 2015
[8]. http://www.astromatic.net/
[9]. Wiley K., Connolly A., Gardner J., Krughof S., Balazinska M., Howe B., Kwon Y., Bu Y. Astronomy in the Cloud: Using MapReduce for Image Coaddition. Publications of the Astronomical Society of the Pacific, 2011, Vol. 123, No. 901, pp. 366-380
[10]. Apache Hadoop http://hadoop.apache.org
[11]. Montage: an astronomical image mosaic engine http://montage.ipac.caltech.edu/
[12]. Message Passing Interface Forum http://www.mpi-forum.org/
[13]. Pegasus: workflow management system http://pegasus.isi.edu/

[14]. Farivar R., Brunner R., Santucci R., Campbell R. Cloud Based Processing of Large Photometric Surveys. Astronomical Data Analysis Software and Systems XXII, 2013, p.91
[15]. Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, December 2004
[16]. Koposov S., Belokurov V., Torrealba G., Wyn N. Evans Beasts of the Southern Wild: Discovery of nine Ultra Faint satellites in the vicinity of the Magellanic Clouds. The Astrophysical Journal, March 2015
[17]. Apache HBase http://hbase.apache.org
[18]. Apache Hive http://hive.apache.org
[19]. Apache Spark http://spark.apache.org

Processing of Raw Astronomical Data of Large Volume by MapReduce Model[4]

[1]S. Gerasimov < gerasimov@mlab.cs.msu.su >
[2]A. Mesheryakov < mesch@iki.rssi.ru >
[1]I. Kolosov < zackwag32@gmail.com >
[1]E. Glotov < glot.unltd@gmail.com >
[1]I. Popov < ivan@mlab.cs.msu.su >
[1]Lomonosov Moscow State University Faculty CMC, 2nd Education Building, GSP-1, Leninskie Gori, Moscow, 119991, Russian Federation
[2]Space Research Institute of the Russian Academy of Sciences, 84/32 Profsoyuznaya Str, Moscow, Russian Federation, 117997

Abstract. Exponential grow of volume, increased quality of data in current (SDSS, DES, PanSTARRS) and incoming sky surveys (LSST) open new horizons for astrophysics but require new approaches to data processing especially big data technologies and cloud computing. This work presents a MapReduce-based approach to solve a major and important computational task in astrophysics - raw astronomical image data processing. We present architecture of Hadoop-based astrophysical pipeline which combines following steps of data processing: background removal, projection, co-addition, PSF-modelling, sky objects features extraction from images. The architecture uses modern implementations of astrophysical image processing algorithms from software packages SWarp, PSFEx, SExtractor. These tools are integrated in MapReduce procedures. The pipeline steps are joined in two phases. First phase - "raw" data processing - includes background removal, projection and images co-edition. Results of the first phase are preprocessed and co-added images into so called cells. Cells interleave by borders. Interleavings help us to process correctly on the second stage large sky objects on borders. The second stage includes steps of PSF-modelling and creation of the sky catalogue by extraction of sky object properties from cells. Experiments showed linear scalability of all processing steps and small impact of Hadoop infrastructure on entire

[4] The project is supported by RFBR grant number 15-29-07085 ofi_m

performance costs. We used one filter data (red) from the Stripe82 dataset. All experiments are made inside cloud platform Microsoft Azure HDInsight.

Keywords: MapReduce; Hadoop; sky survey; big data; cloud computing; image processing

DOI: 10.15514/ISPRAS-2015-27(6)-20

For citation: Gerasimov S., Mesheryakov A., Kolosov I., Glotov E., Popov I. Processing of Raw Astronomical Data of Large Volume by MapReduce Model. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 315-334 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-20

References

[1]. The Sloan Digital Sky Survey (SDSS) http://www.sdss.org/
[2]. Burke B., Gregory J., Cooper M., Loomis A., Young D., Lind T., Doherty P., Daniels P., Landers D., Ciampi J., Johnson K., O'Brien P. CCD Imager Development for Astronomy. Lincoln Laboratory Journal, 2007, Volume 16, Number 2
[3]. Subaru-HSC http://www.naoj.org/Projects/HSC/
[4]. The Dark Energy Sky Survey (DES) http://www.darkenergysurvey.org/
[5]. Pan-STARRS http://pan-starrs.ifa.hawaii.edu/public/
[6]. The Large Synoptic Survey Telescope (LSST) http://www.lsst.org/
[7]. Zhang Y., Zhao Y. Astronomy in the Big Data Era. Data Science Journal, 2015
[8]. http://www.astromatic.net/
[9]. Wiley K., Connolly A., Gardner J., Krughof S., Balazinska M., Howe B., Kwon Y., Bu Y. Astronomy in the Cloud: Using MapReduce for Image Coaddition. Publications of the Astronomical Society of the Pacific, 2011, Vol. 123, No. 901, pp. 366-380
[10]. Apache Hadoop http://hadoop.apache.org
[11]. Montage: an astronomical image mosaic engine http://montage.ipac.caltech.edu/
[12]. Message Passing Interface Forum http://www.mpi-forum.org/
[13]. Pegasus: workflow management system http://pegasus.isi.edu/
[14]. Farivar R., Brunner R., Santucci R., Campbell R. Cloud Based Processing of Large Photometric Surveys. Astronomical Data Analysis Software and Systems XXII, 2013, p.91
[15]. Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, December 2004
[16]. Koposov S., Belokurov V., Torrealba G., Wyn N. Evans Beasts of the Southern Wild: Discovery of nine Ultra Faint satellites in the vicinity of the Magellanic Clouds. The Astrophysical Journal, March 2015
[17]. Apache HBase http://hbase.apache.org
[18]. Apache Hive http://hive.apache.org
[19]. Apache Spark http://spark.apache.org

Спектрально-аналитический метод распознавания неточных повторов в символьных последовательностях

[1]*А.Н. Панкратов <pan@impb.ru>*
[1]*Р.К. Тетуев <ruslan.tetuev@gmail.com>*
[1]*М.И. Пятков <mpyatkov@gmail.com>*
[2]*В.П. Тойгильдин <vladislav.toigildin@cs.msu.su>*
[2]*Н.Н. Попова <popova@cs.msu.su>*
[1] *Институт математических проблем биологии РАН,*
142290, Россия, г. Пущино Московской обл., ул. Институтская, дом 4
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1., стр. 52, факультет ВМК

Аннотация. Предложены теоретическое обоснование и алгоритмическая реализация спектрально-аналитического метода распознавания повторов в символьных последовательностях. Теоретическое обоснование основывается на теореме об эквивалентном представлении символьной последовательности вектором непрерывных характеристических функций. Сравнение фрагментов характеристических функций производится в стандартной метрике в евклидовом пространстве коэффициентов разложения рядов Фурье по ортогональным многочленам. Существенным свойством данного подхода является способность оценивать повторы на разных масштабах. Другим важным свойством является возможность эффективного распараллеливания по данным. При разработке алгоритмов предпочиталась схема вычислений с минимальным количеством обращений к оперативной памяти, подразумевающая повторяющиеся и отложенные вычисления. В данной парадигме разработан алгоритм вычисления коэффициентов разложения по ортогональным многочленам за счет использования рекуррентных соотношений. Показано, что алгоритм вычисления коэффициентов разложения по ортогональным многочленам может быть эффективно векторизован за счет вычислений с фиксированной длиной вектора. Распараллеливание и векторизация реализованы с использованием стандарта OpenMP и расширения Cilk Plus языка C/C++. Разработанный метод эффективно масштабируется в зависимости от параметров задачи и числа ядер процессора на системах с общей памятью.

Ключевые слова: спектрально-аналитический метод; ряды Фурье; ортогональные многочлены; рекуррентные соотношения; OpenMP; Cilk Plus

DOI: 10.15514/ISPRAS-2015-27(6)-21

Для цитирования: Панкратов А.Н., Тетуев Р.К., Пятков М.И., Тойгильдин В.П., Попова Н.Н. Спектрально-аналитический метод распознавания неточных повторов в символьных последовательностях. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 335-344. DOI: 10.15514/ISPRAS-2015-27(6)-21.

1. Введение

Спектрально-аналитический подход является комбинированным численно-аналитическим методом решения информационных задач, основанным на представлении функций отрезками ортогональных рядов с последующей обработкой в пространстве коэффициентов разложения. Его применение к задаче поиска повторов в биоинформационных последовательностях было показано в работах [1-6].

При разработке программы большое внимание было уделено эффективной реализации предложенного метода, поскольку сравнение с существующими методами поиска повторов, основанными на методах дискретной математики, возможно было на основе законченных программ. Перевод задачи в область математического анализа стимулировал также разработку математического обоснования такой редукции.

2. Описание и обоснование метода

Для адаптации спектрально-аналитического подхода к задачам биоинформатики потребовалось обобщить понятие точечной матрицы. При этом данный подход не теряет общности и может быть применен к поиску повторов в любых символьных последовательностях. В данной работе подход описан в максимально общем виде с сохранением части терминологии и примеров из области биоинформатики.

2.1 Теорема о разложении символьной последовательности

Метод основан на спектральном разложении функций, составляющих характеристическое описание текстовой последовательности, при котором важны следующие свойства функций: 1) полнота и 2) непрерывность. Полнота описания означает, что исходная последовательность может быть восстановлена по характеристическим кривым. Второе свойство необходимо для оценки повторов по форме изменения характеристик. Его выполнение в случае символьных последовательностей обеспечивается тем, что вычисляются кривые содержания подмножеств нуклеотидов в окне заданной длины вдоль последовательности макромолекулы. К этому типу кривых относится хорошо известная и изученная в биоинформатике кривая GC-содержания. При этом размер окна, который является параметром такого описания, вводит фактически понятие масштаба для рассматриваемой символьной последовательности.

Для общего случая сформулируем и докажем следующую **теорему**:

для произвольной символьной последовательности в алфавите из M символов существует $Log_2 M$ характеристических функций, из которых исходная последовательность может быть восстановлена, при этом функции являются - значными, где K – параметр масштаба.

Для доказательства закодируем символы последовательности числовым вектором в двоичной системе счисления. Потребуется не меньше $Log_2 M$ бит. Теперь рассмотрим скользящее окно ширины K, и просуммируем количество единиц определенного бита всех символов последовательности в этом окне. Определенную таким образом функцию, зависящую от начала положения окна в последовательности, назовем характеристической функцией последовательности, соответствующей заданному биту двоичной кодировки символов. Каждый бит каждого символа последовательности можно восстановить из соответствующей ему характеристической функции. При этом значение характеристической функции равно количеству символов, в кодировке которых в соответствующем бите стоит единица, т.е. является функцией содержания некоторого подмножества символов (не менее половины из всего алфавита) в окне, скользящем вдоль последовательности.

Например, в случае геномных последовательностей, заданных в 4-х буквенном алфавите $\{A, T, G, C\}$, можно использовать двухбитную кодировку, достигая таким образом 4-х кратного сжатия геномных файлов, заданных исходно в 8-битной кодировке. При этом в качестве характеристических функций могут выступать кривые содержания нуклеотидов G,C и G,A в скользящем окне длины K [5].

2.2 Структурная схема метода

Характеристические кривые, которые составляют описание объекта, разбиваются на перекрывающиеся фрагменты длины W с шагом d. После этого производится попарно сравнение всех фрагментов f_i, g_i, рассматриваемых как дискретные функции с нумерацией отсчетов в пределах окна длины W, на основе стандартной метрики в евклидовом пространстве:

$$\rho(f,g) = (f-g, f-g) = \frac{1}{W}\sum_{i=1}^{W}(f_i - g_i)^2$$

Для сокращения вычислений расстояний между фрагментами используется аппроксимация фрагментов характеристических функций отрезками ортонормированного ряда. Поэтому оценка расстояния осуществляется по формуле:

$$\rho(f,g) = \sum_{i=1}^{N}(c_i - d_i)^2$$

где c_i, d_i – коэффициенты разложения ряда Фурье, а N – их количество (причем, $N \ll W$). Использование спектрального разложения позволяет не только экономно производить оценку расстояния, но также производить преобразования для оценки инвертированных и комплементарных последовательностей в пространстве коэффициентов разложения, что означает одновременное распознавание всех типов повторов без преобразования самой последовательности [5].

Для распознавания повторов используется пороговое решающее правило: если $\rho < \varepsilon$, где ε - пороговое значение, то фрагменты считаются похожими, а если $\rho \geq \varepsilon$, фрагменты не похожи. При наличии нескольких характеристических кривых, составляющих полное описание объекта, распознавание по ним ведется одновременно, а итоговый результат является логическим умножением решающих правил по каждой из характеристических функций. Такой подход улучшает устойчивость распознавания к ошибкам. Это следует из того, что решающее правило срабатывает в районе минимумов метрики ρ, рассматриваемой как функции от номера фрагмента. Таким образом, множество минимумов определяет множество кандидатов на повтор. В случае двух признаков, например, GC- и GA- кривых, множество повторов берется как пересечение множеств кандидатов на повтор, полученных по каждому из признаков отдельно.

После проведения этих операций результаты сравнения отображаются на точечной матрице, одна точка на которой, однако, соответствует сравнению двух целых фрагментов, а не просто сайтов последовательности. Точечная матрица является одним из наглядных стандартных представлений результатов сравнения двух последовательностей, позволяющим отобразить выравнивание неточных повторов, а также их взаимное расположение. Обобщенная точечная матрица позволяет получить новые возможности для выравнивания неточных повторов. Например, было показано, что неточный протяженный тандемный повтор может быть отображен совершенным квадратом на точечной матрице. Это достигается за счет правильного подбора соотношения между размерами окна и шага окна аппроксимации. На основе этого важного результата построен полностью автоматизированный метод распознавания тандемных повторов и найдены неизвестные ранее повторы.

Структурная схема метода выглядит следующим образом:

1) Предварительная обработка символьной последовательности. На этом этапе происходит формирование исходного алфавита: удаление ненужных символов, перекодировка символов последовательности.
2) Преобразование символьной последовательности в пучок непрерывных характеристических функций на основе доказанной теоремы.
3) Преобразование характеристических функций в спектральное представление. В отличие от предыдущих шагов этот этап подразумевает необратимое сжатие информации.

4) Спектральное сравнение фрагментов последовательностей.
5) Отображение и анализ точечной матрицы с целью выявления протяженных повторов, тандемных повторов и исследования взаимного расположения повторов.
6) Верификация повторов путем выравнивания методами динамического программирования.

3. Эффективная реализация метода

Критически важными с самого начала развития метода были вопросы его вычислительной сложности и эффективной реализации. Все этапы алгоритма являются независимыми друг от друга и хорошо распараллеливаются по данным. Вычислительная сложность всех этапов линейно зависит от длин последовательностей, кроме построения точечной матрицы, которое в общем случае является квадратичным по сложности вычислительным процессом, т.к. зависит от произведения длин анализируемых последовательностей. Однако, введенное функциями содержания естественное понятие масштаба при анализе нуклеотидных последовательностей позволяет утверждать, что построение точечной матрицы фиксированного размера на разных масштабах производится за линейное время в зависимости от длин анализируемых последовательностей [6].

3.1 Основные принципы

При реализации разработанного метода в виде алгоритмов и программ учитывался современный уровень развития вычислительной техники с широким применением параллельных и векторных вычислений, графических ускорителей и спецвычислителей, распределенных и облачных технологий. Для экономии памяти и пересылок между узлами предпочтение отдается вычислительным схемам с минимальным использованием промежуточных результатов вычислений.

Например, при вычислении коэффициентов разложения значения ортогональных многочленов вычисляются по рекуррентным соотношениям и не запоминаются в промежуточных массивах. Это позволяет эффективно использовать кэш процессора, а также приводит к хорошей масштабируемости на многоядерных процессорах и графических ускорителях, так как снижает нагрузку на оперативную память при многопоточных вычислениях.

Другой пример экономии памяти связан с использованием точечной матрицы. Для лучшей масштабируемости приложения в зависимости от числа процессоров и длины последовательности предлагается не сохранять матрицу, а вычислять значения ее элементов по требованию. Это позволяет даже длинные последовательности обрабатывать на одном узле.

3.2 Рекуррентный алгоритм вычисления коэффициентов разложения

Рассмотрим более подробно процесс вычисления коэффициентов разложения по ортогональным многочленам на примере многочленов Чебышева. Коэффициенты разложения функции $f(t)$ вычисляются по формулам:

$$c_j = \frac{2}{W}\sum_{i=1}^{W} f(t_i)\,T_j(t_i)$$

где t_i – узлы квадратурной формулы Гаусса, $T_j(t)$ - многочлены Чебышева непрерывного аргумента t, удовлетворяющие рекуррентному соотношению с начальными условиями:

$$T_0(t) = 1, T_1(t) = t, T_{j+1}(t) = 2tT_j(t) - T_{j-1}(t)$$

В [7] представлен алгоритм вычисления коэффициентов разложения на языке C++. Ниже представлена оптимизированная версия этого алгоритма на языке C++11 с расширением Cilk Plus:

```
void chebft(int n, double t[n], double f[n], int m, double c[m], int l){
        double p1[l], p2[l], p3[l];
        c[:] = 0.0;
        for (int i = 0; i < n; i += l){
                if (i > n - l) l = n - i;
                p2[0:l] = t[i:l] * f[i:l] * 2.0 / n;
                p1[0:l] = f[i:l] * 2.0 / n;
                c[0] += __sec_reduce_add(p1[0:l]);
                for (int j = 1; j < m; j++){
                        p3[0:l] = p2[0:l];
                        p2[0:l] = p1[0:l];
                        p1[0:l] = 2.0 * t[i:l] * p2[0:l] - p3[0:l];
                        c[j] += __sec_reduce_add(p1[0:l]);
                }
        }
}
```

В отличие от исходного алгоритма [7] здесь сделаны следующие усовершенствования:

1) применено рекуррентное соотношение для вычисления значений многочленов;

2) умножение на значение функции и нормировка включены в рекуррентное соотношение умножением на начальные условия;

3) произведена векторизация по соседним узлам сетки, l – длина вектора.

В соответствии с внесенными изменениями представленный алгоритм назовем векторно-рекуррентным с фиксированной глубиной векторизации.

Единственные временные переменные в этом алгоритме – это массивы $p1, p2, p3$, размер которых зависит от длины l, значения которой следует выбирать равным длине векторных регистров процессора.

Разработанная вычислительная схема обладает не только высокой эффективностью, но масштабируемостью на многоядерных процессорах. При вычислении коэффициентов разложения от массива функций в задаче поиска повторов достигается практически идеальное масштабирование по числу используемых ядер при распараллеливании с помощью директив OpenMP.

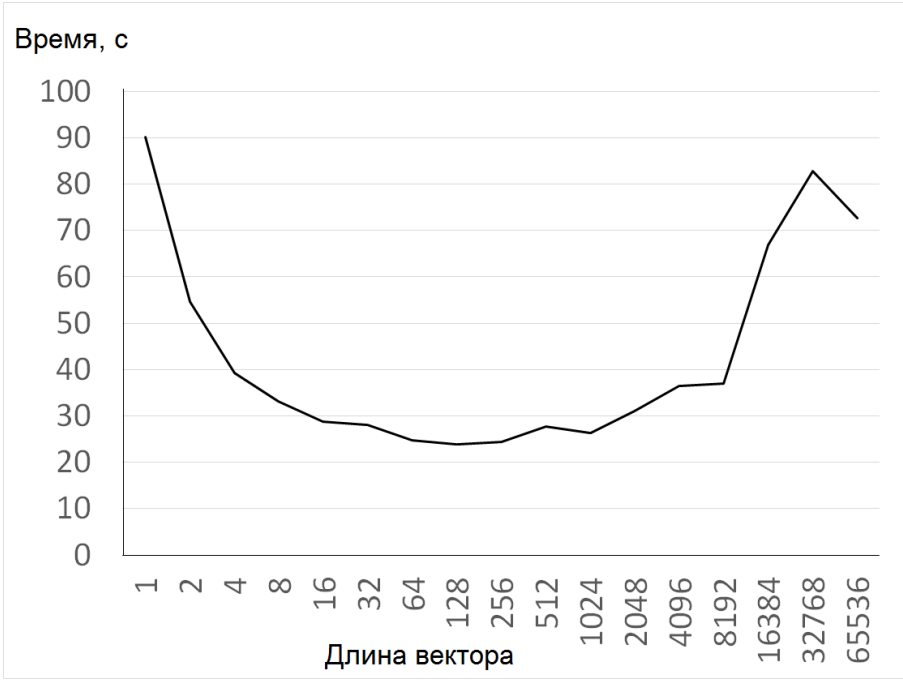


Рис. 1 Время выполнения программы в зависимости от длины вектора.

На рис. 1 представлены результаты расчетов $N = 10^5$ коэффициентов разложения на сетке $W = 10^5$ в зависимости от длины вектора. Вычисления производились на процессоре AMD Phenom. Следует отметить, что при максимальной длине вектора, приближающейся к длине всей сетки, эффект от использования векторных операций процессора практически исчезает. Это происходит из-за того, что в этом случае кэш-память процессора фактически не используется. Кроме того, оптимальным значением глубины векторизации кода для данного процессора является 128, при этом достигается 4-х кратное ускорение за счет утилизации векторных инструкций процессора (SSE, AVX). Таким образом, построен алгоритм, который эффективно масштабируется в

зависимости от длины сетки, т.к. использует фиксированную глубину векторизации.

Заключение

Можно выделить существенные свойства спектрально-аналитического метода, определяющие его эффективность при распознавании неточных повторов:

1) интегральное оценивание повторов, которое позволяет нивелировать локальные неточности в повторах сигнала;
2) выбор масштаба за счет изменения размеров окна и его шага, что позволяет производить гибкое выравнивание неточных повторов;
3) использование спектрального разложения сигналов, которое обуславливает значительное сокращение вычислений;
4) высокая степень распараллеливания и векторизации вычислений.

В данной работе показана принципиальная возможность поиска повторов и высокая вычислительная эффективность предложенных алгоритмов в случае построения точечных матриц. Достигнутые результаты позволяют сделать вывод о возможности дальнейшего совершенствования качества работы метода и его применимости к конкретным задачам. Например, перспективным является вопрос о применимости разработанного метода в задаче поиска неточных повторов по заданному образцу среди множества геномов. Решение этой задачи возможно с привлечением алгоритмического аппарата, связанного с распределенными и облачными вычислениями.

Работа выполняется при поддержке грантов РФФИ №14-07-00654, 14-07-00924, 14-07-31306, 15-29-07063.

Список литературы

[1]. Дедус Ф.Ф., Куликова Л.И., Махортых С.А., Назипова Н.Н., Панкратов А.Н., Тетуев Р.К. Аналитические методы распознавания повторяющихся структур в геномах. Доклады Академии Наук, 2006, т.411, №5, с.599-602, doi: 10.1134/S1064562406060354.

[2]. Дедус Ф.Ф., Куликова Л.И., Махортых С.А., Назипова Н.Н., Панкратов А.Н., Тетуев Р.К. Распознавание структурно-функциональной организации генетических последовательностей. Вестник московского университета. Серия 15: Вычислительная математика и кибернетика, 2007, т.31, №2, с.12-16, doi: 10.3103/S0278641907020021.

[3]. Pankratov A.N., Gorchakov M.A., Dedus F.F., Dolotova N.S., Kulikova L.I., Makhortykh S.A., Nazipova N.N., Novikova D.A., Olshevets M.M., Pyatkov M.I., Rudnev V.R., Tetuev R.K., and Filippov V.V. Spectral Analysis for Identification and Visualization of Repeats in Genetic Sequences. Pattern Recognition and Image Analysis, 2009, Vol. 19, №4, pp. 687–692, doi: 10.1134/S105466180904018X.

[4]. Тетуев Р.К., Назипова Н.Н., Панкратов А.Н., Дедус Ф.Ф. Поиск мегасателлитных тандемных повторов в геномах эукариот по оценке осцилляций кривых GC-содержания. Математическая биология и биоинформатика, 2010, Т.5, №1, с.30-42, doi: 10.17537/2010.5.30.

[5]. Панкратов А.Н., Пятков М.И., Тетуев Р.К., Назипова Н.Н., Дедус Ф.Ф. Поиск протяженных повторов в геномах на основе спектрально-аналитического метода. Математическая биология и биоинформатика, 2012, Т.7, №2, с.476–492, doi: 10.17537/2012.7.476.

[6]. Pyatkov M.I., Pankratov A.N. SBARS: fast creation of dotplots for DNA sequences on different scales using GA-, GC-content. Bioinformatics, Vol. 30, №12, 2014, pp. 1765–1766, doi: 10.1093/bioinformatics/btu095.

[7]. W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery Numerical Recipes. The Art of Scientific Computing. Third Edition. Cambridge University Press, 2007, 1256 pp.

Spectral Analytical Method of Recognition of Inexact Repeats in Character Sequences

[1]A.N. Pankratov <pan@impb.ru>
[1]R.K. Tetuev <ruslan.tetuev@gmail.com>
[1]M.I. Pyatkov <mpyatkov@gmail.com>
[2]V.P. Toigildin <vladislav.toigildin@cs.msu.su>
[2]N.N. Popova <popova@cs.msu.su>
[1] Institute of Mathematical Problems of Biology, 4 Institutskaya Str., Pushchino, Moscow Region, 142290, Russian Federation
[2]Lomonosov Moscow State University, 2nd Education Building, Faculty CMC, GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation

Abstract. Proposed are theoretical basis and algorithmic implementation of spectral-analytical method of recognition of repeats in character sequences. The theoretical justification is based on the theorem on equivalent representation of the character sequence by the vector of continuous characteristic functions. Comparison of fragments of characteristic functions is performed in the standard metric in Euclidean space of expansion coefficients of the Fourier series of orthogonal polynomials. An essential feature of this approach is the ability to evaluate repeats at different scales. Another important feature is the possibility of efficient parallelization of data. In the development of algorithms we preferred scheme of computing with a minimal amount of references to memory, implying repetitive calculations and evaluations on demand. In this paradigm, proposed is an algorithm for calculating the coefficients of expansions in the orthogonal polynomials through the use of recurrence relations. It is shown that the algorithm for calculating the coefficients of expansions in the orthogonal polynomials can be effectively vectorized by computing with a fixed vector length. Parallelization and vectorization implemented using the OpenMP standard and extension Cilk Plus of language C/C++. The developed method effectively scales, depending on the parameters of the problem and the number of processor cores on systems with shared memory.

Keywords: spectral-analytical method; Fourier series; orthogonal polynomials; recurrence relations; OpenMP; Cilk Plus

DOI: 10.15514/ISPRAS-2015-27(6)-21

For citation: Pankratov A.N., Tetuev R.K., Pyatkov M.I., Toigildin V.P., Popova N.N. Spectral Analytical Method of Recognition of Inexact Repeats in Character Sequences. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 335-344 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-21

References

[1]. Dedus F.F., Kulikova L.I., Makhortykh S.A., Nazipova N.N., Pankratov A.N. and Tetuev R.K. Analytical Recognition Methods for Repeated Structures in Genomes. Doklady Mathematics, 2006, Vol. 74, №3, pp. 926-929, doi: 10.1134/S1064562406060354.

[2]. Dedus F.F., Kulikova L.I., Makhortykh S.A., Nazipova N.N., Pankratov A.N., and Tetuev R.K. Recognition of the Structural–Functional Organization of Genetic Sequences. Moscow University Computational Mathematics and Cybernetics, 2007, Vol. 31, No. 2, pp.49–53, doi: 10.3103/S0278641907020021.

[3]. Pankratov A.N., Gorchakov M.A., Dedus F.F., Dolotova N.S., Kulikova L.I., Makhortykh S.A., Nazipova N.N., Novikova D.A., Olshevets M.M., Pyatkov M.I., Rudnev V.R., Tetuev R.K., and Filippov V.V. Spectral Analysis for Identification and Visualization of Repeats in Genetic Sequences. Pattern Recognition and Image Analysis, 2009, Vol. 19, №4, pp. 687–692.

[4]. Tetuev R.K., Nazipova N.N., Pankratov A.N., Dedus F.F. Search for Megasatellite Tandem Repeats in Eukaryotic Genomes by Estimation of GC-content Curve Oscillations. Math. Biol. Bioinf. 2010, 5(1):30-42, doi: 10.17537/2010.5.30.

[5]. Pankratov A.N., Pyatkov M.I., Tetuev R.K., Nazipova N.N., Dedus F.F. Search for Extended Repeats in Genomes Based on the Spectral-Analytical Method. Math. Biol. Bioinf. 2012;7(2):476-492, doi: 10.17537/2012.7.476.

[6]. Pyatkov M.I., Pankratov A.N. SBARS: fast creation of dotplots for DNA sequences on different scales using GA-, GC-content. Bioinformatics, Vol. 30, №12, 2014, pp. 1765–1766, doi: 10.1093/bioinformatics/btu095.

[7]. W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery Numerical Recipes. The Art of Scientific Computing. Third Edition. Cambridge University Press, 2007, 1256 pp.

Облачный сервис ОИЯИ: статус и перспективы

[1]Н. А. Балашов <balashov@jinr.ru>
[1]А. В. Баранов <baranov@jinr.ru>
[1,2]В. В. Кореньков <korenkov@jinr.ru>
[1,2]Н.А. Кутовский <kut@jinr.ru>
[1]А. В. Нечаевский <nechav@jinr.ru>
[1,2]Р.Н. Семенов <roman@jinr.ru>
[1]Объединенный Институт Ядерных Исследований,
141980, Россия, Московская обл., г. Дубна, ул. Жолио-Кюри, 6
[2]Российский Экономический Университет им. Г.В. Плеханова,
117997, Российская Федерация, г.Москва, Стремянный пер., 36

Аннотация: Рассмотрены основные принципы и примеры использования облачных вычислительных сред различными научными лабораториями. Описаны работы, проводимые в Объединенном институте ядерных исследований (ОИЯИ) и направленные на развитие имеющейся облачной инфраструктуры, принятые стратегии повышения эффективности, отказоустойчивости и надежности, а также подходы к интеграции с другими ОВС.

Ключевые слова: распределенные вычисления; облачные вычисления; виртуализация; ЦОД.

DOI: 10.15514/ISPRAS-2015-27(6)-22

Для цитирования: Балашов Н.А., Баранов А.В., Кореньков В.В., Кутовский Н.А., Нечаевский А.В., Семенов Р.Н. Облачный сервис ОИЯИ: статус и перспективы. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 345-354. DOI: 10.15514/ISPRAS-2015-27(6)-22.

Введение

И в науке, и в бизнесе в настоящий момент широко используются облачные вычислительные среды (ОВС). ОВС характеризуют высокая гибкость архитектуры и возможность снижения расходов на содержание вычислительной инфраструктуры организации. Широкое распространение облачных вычислений сейчас не вызывает сомнений. В значительной степени этому способствовала возросшая потребность в суперкомпьютерных ресурсах, которые были бы доступны конечным пользователям для проведения вычислений.

По всему миру на данный момент создано и успешно работают множество крупномасштабных центров обработки данных. И сейчас многие мировые научные организации разворачивают собственные частные ОВС, перенося в них свои вычисления и информационные сервисы.

Например, в Европейской организации по ядерным исследованиям – ЦЕРН (European Organization for Nuclear Research, CERN) развернута одна из крупнейших облачных инфраструктур в мире, которая включает четыре частных облака, территориально размещённых в двух ЦОДах и интегрированных в единую систему обработки данных с большого адронного коллайдера – БАК (Large Hadron Collider, LHC) [1].

Национальная ускорительная лаборатория им. Энрико Ферми (Fermi National Accelerator Laboratory, FNAL) также имеет собственное частное облако FermiCloud, используемое для обработки данных с физических экспериментов этой организации и поддержки их информационных сервисов [2].

В Объединенном институте ядерных исследований (ОИЯИ) есть своя облачная вычислительная среда, реализованная Лабораторией информационных технологий (ЛИТ) [3]. В ЛИТ ОИЯИ в настоящий момент активно ведутся работы по исследованию возможностей ОВС, разрабатываются методики применения облачных технологий для решения различного класса задач. ОВС ОИЯИ основана на модели «инфраструктура как сервис», и схема ее работы показана на рис. 1.

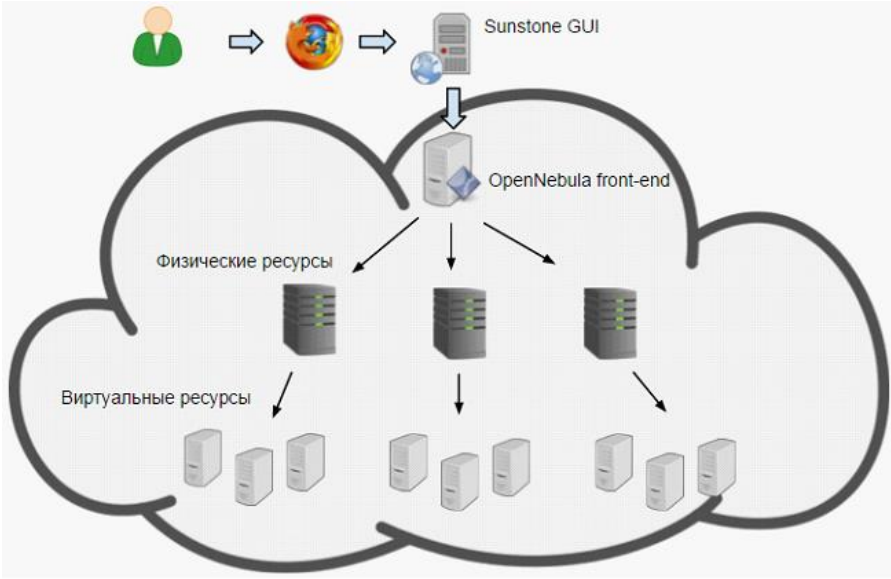


Рис. 1. Схема работы ОВС ОИЯИ.

1. Использование облачного сервиса ОИЯИ сторонними экспериментами

ОИЯИ принимает активное участие во многих научных экспериментах, в том числе в таких как BES-III и NOvA. В рамках работы над экспериментом BES-III был развернут интерфейс доступа к облачному сервису ОИЯИ, основанный на протоколе Open Cloud Computing Interface (OCCI), что позволило использовать ОВС ОИЯИ для запуска вычислительных задач данного эксперимента [4]. Для этого используется система управления заданиями DIRAC – программный фреймворк (framework), предназначенный для организации распределенных вычислений, поддерживающий интерфейсы для запуска задач как в грид, так и в локальные кластеры и ОВС. Интерфейс OCCI позволяет осуществлять операции управления виртуальными машинами ОВС (создание, запуск, удаление и т.д.) и проводить их так называемую контекстуализацию – передачу необходимых сетевых параметров и конфигурационных скриптов внутрь ВМ, которые будут выполнены при старте виртуальной машины и таким образом подготовят ее к работе. Схема работы интеграции DIRAC и облачного сервиса ОИЯИ представлена на рис. 2.

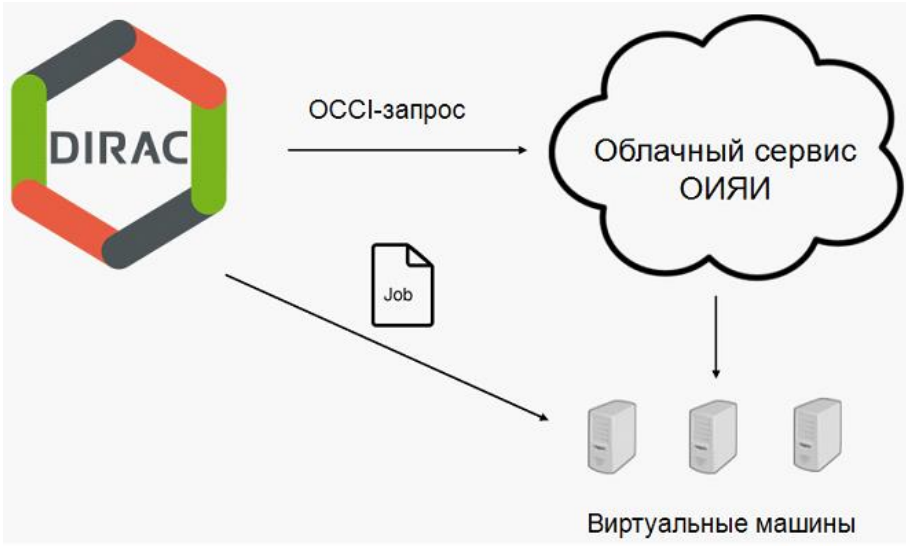


Рис. 2. Схема взаимодействия системы DIRAC и облачного сервиса ОИЯИ.

Эксперимент NOvA – это нейтринный ускорительный эксперимент нового поколения, который является одним из первых, построенных на базе ускорителя в FNAL. В ОВС ОИЯИ в ходе участия в этом эксперименте была развернут фреймворк (framework) Art, позволивший использовать данный

сервис для проведения физического анализа, поддержки набора и контроля экспериментальных данных, моделирования нейтринных событий. Также в рамках эксперимента NOvA планируется создание выделенного виртуального Центра Обработки Данных (ЦОД), представляющего собой совокупность серверов, объединенных в кластер. Ресурсы данного ЦОД, выделенных в отдельный виртуальный кластер рамках ОВС ОИЯИ, планируется предоставлять исключительно пользователям, входящим в рабочую группу эксперимента NOvA. Данные ресурсы в ходе эксперимента планируется использовать как для решения задач локальных пользователей из ОИЯИ (членов рабочей группы), так и для поддержки обработки данных эксперимента путем интеграции с американской грид-инфраструктурой Open Science Grid (OSG) (интеграция подобного рода также реализована и в эксперименте BES-III).

ОВС ОИЯИ используется для решения различных по ресурсоемкости задач. Так в ОИЯИ ведутся работы по строительству коллайдера НИКА, для моделирования событий которого (моделирование столкновений тяжелых ядер таких элементов, как золото, медь, уран) используется ОВС, так как требуется существенное количество доступной физической памяти и высокая производительность используемых процессоров. В то же время облачный сервис используется в ОИЯИ и для решения менее ресурсоемких задач (например, таких как разработка информационных систем, тестирование обновлений ПО, обучение студентов работе с облачными и грид-технологиями [5] и т.д.).

2. Выгрузка вычислительной нагрузки в сторонние облачные среды

Частные облачные среды обычно имеют гораздо меньше вычислительных ресурсов, чем коммерческие публичные облака, а потребности в вычислительных ресурсах растут все быстрее. С ростом вычислительных потребностей велика вероятность, что в определенный момент времени внутренних ресурсов частного облачного сервиса станет недостаточно, чтобы справляться с пиковыми нагрузками. Все это обуславливает актуальность направления исследований в области разработки методов повышения эффективности использования вычислительных ресурсов, а также поиск возможностей расширения ресурсов за счет интеграции различных облачных сред между собой.

Один из возможных путей решения проблемы дефицита ресурсов – это передача части нагрузки в облачный сервис стороннего провайдера [6], в роли которого может выступать либо коммерческий облачный сервис (например, такой как Amazon EC2 или Rackspace), либо партнерская инфраструктура, которая предоставляет какой-либо API. Сервисы, имеющие возможность выгрузить часть своей нагрузки в сторонний облачный сервис, называются гибридными облачными сервисами.

В настоящий момент облачный сервис ОИЯИ перегружен и перестал справляться с быстро растущими потребностями пользователей в вычислительной мощности. Было принято решение о сотрудничестве с Институтом Физики Национальной Академии Наук Азербайджана (ИФ НАН), располагающего достаточным количеством свободных ресурсов, и в рамках данного сотрудничества попробовать интегрировать оба сервиса, используя т.н. подход «cloud bursting».

Платформа OpenNebula, на базе которой построены облачные сервисы ОИЯИ и ИФ НАН Азербайджана, уже имеет встроенную поддержку механизма «cloud bursting», однако поставляемый вместе этой облачной платформой набор драйверов поддерживает только несколько популярных облачных сервисов, таких как Amazon EC2, IBM SoftLayer и Microsoft Azure. Тот факт, что OpenNebula является платформой с открытым исходным кодом, позволяет разрабатывать собственные драйвера. Команда разработчиков облачного сервиса ЛИТ ОИЯИ разработала драйвер интеграции облаков, основанных на платформе OpenNebula, используя комбинацию встроенного программного интерфейса XML-RPC и стороннего OCCI. Решение об использовании двух разных интерфейсов было принято, исходя из необходимости будущей интеграции с облачным сервисом ИТФ им. Н.Н. Боголюбова, который развёрнут с использованием облачного ПО OpenStack. В данный момент работа по интеграции ведется сразу в нескольких направлениях: идёт тестирование уже выполненной интеграции с облаком Российского Экономического Университета им. Г.В. Плеханова; производится доработка драйвера для интеграции облака ОИЯИ с облаком ИТФ им. Н.Н. Боголюбова, ведутся работы по интеграции облака ОИЯИ и ИФ НАН Азербайджана.

3. Федеративное облако ЕГИ

В данный момент рассматривается возможность подключения облачного сервиса ОИЯИ в качестве провайдера ресурсов в Федеративное облако Европейской грид-инфраструктуры (EGI Federated Cloud) [7]. Федеративное Облако ЕГИ – это интеграция частных академических облаков и виртуализованных ресурсов, построенная на открытых стандартах и нацеленная на решение задач научного сообщества. Результатом этой интеграции является новый тип инфраструктуры, основанной на федеративном управлении сервисами, и предлагающей пользователям четыре модели использования инфраструктуры: хостинг сервисов (веб-серверы, базы данных и т.д.), высоконагруженные вычисления и обработка данных, репозиторий данных, одноразовые и тестовые среды.

4. Интеллектуальное планирование ресурсов

В ОИЯИ также ведутся исследования методов организации внутренней инфраструктуры ОВС для повышения скорости выделения и развертывания

виртуальных ресурсов, а также повышения отказоустойчивости и надежности их работы.

4.1 Динамическое перераспределение ресурсов

В настоящий момент на базе ЛИТ ОИЯИ, кроме прочих, ведутся работы по созданию на базе платформы OpenNebula информационной системы динамического перераспределения ресурсов, которая реализует подход, основанный на методике ранжирования виртуальных машин и серверов, составляющих облачный сервис. Данная методика базируется на анализе исторической информации о фактически потребляемых физических ресурсах. Практической реализацией методики будет программное обеспечение, реализующее на основе сопоставления рангов оптимального в данный конкретный момент времени распределения виртуальных машин на физических серверах. Иллюстрация данного подхода показана на рис. 3.

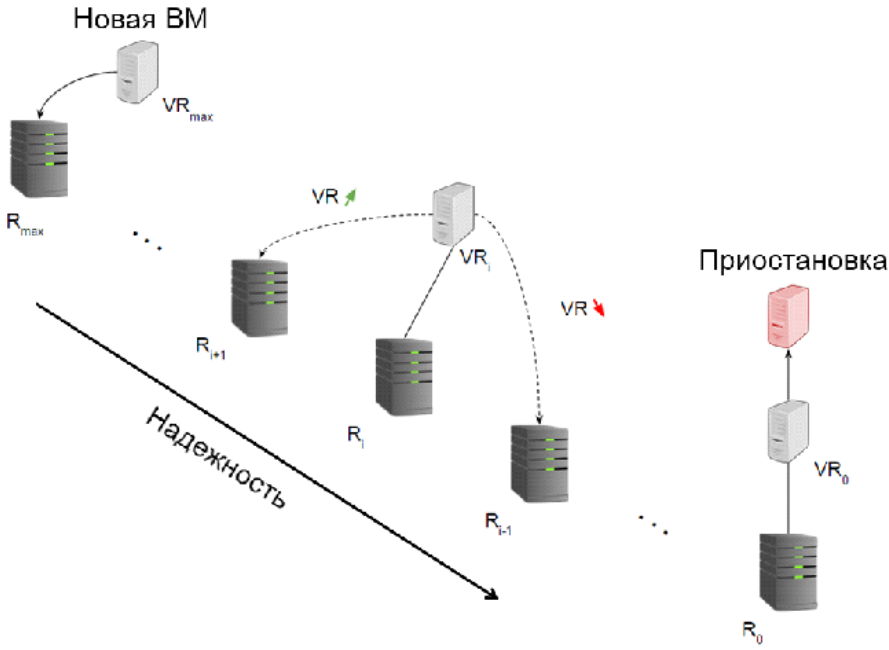


Рис. 3. Схема работы системы динамического перераспределения ресурсов.

4.2 Прототип системы мониторинга

Первым шагом в реализации вышеописанного планировщика является сбор данных о фактической нагрузке виртуальных машин (и, возможно, других

метрик производительности). С этой целью был разработан прототип подобной системы сбора исторических данных. Сбор информации осуществляется скриптами, написанными на языке ruby, которые получают метрики всех виртуальных машин от гипервизора каждого сервера и передают их на центральный сервер, где они записываются в единый отформатированный определенным образом текстовый файл. Данные из этого файла затем заносятся в две базы данных: MySQL и InfluxDB. Для анализа данных важна скорость обработки данных и для этой задачи больше подходит СУБД MySQL, в то время как InluxDB была специально спроектирована для работы с временными рядами и метриками и больше подходит для визуализации данных. В качестве системы визуализации была использована система Grafana. Схема работы прототипа системы мониторинга показана на рис. 4.

Рис. 4. Схема работы прототипа системы мониторинга.

Заключение

Все вышесказанное доказывает, что применение облачных вычислительных систем сейчас – задача актуальная и значимая. Для построения облачных сред научные лаборатории во многих случаях используют свободные облачные платформы с открытым исходным кодом. Например, облачная среда ЦЕРН построена на платформе OpenStack, FermiCloud – на платформе OpenNebula, и та и другая являются свободными платформами с открытым исходным кодом. Использование открытых платформ позволяет свободно модифицировать их,

проводить исследования по совершенствованию и оптимизации облачных вычислительных сред, построенных на их основе. Развитие облачных технологий привело на сегодняшний день уже к следующему этапу – интеграции облачных вычислительных сред, что накладывает дополнительные требования как на используемые платформы, так и на основные принципы и методы работы ОВС в целом, и порождает целый спектр новых задач и исследований. Проводимые работы поддержаны грантами РФФИ №15-29-07027 и №14-07-90405.

Список литературы

[1]. “Virtualization, clouds and IaaS at CERN”, Helge Meinhard, VTDC '12 Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing, pp. 27-28, ACM New York, NY, USA ©2012

[2]. “Automatic Cloud Bursting under FermiCloud”, Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), 2013, pp. 681 - 686.

[3]. A. V. Baranov, N. A. Balashov, N. A. Kutovskiy, R. N. Semenov, “Cloud Infrastructure at JINR”, Computer Research and Modeling, 2015, vol. 7, no. 3, pp. 463–467.

[4]. “BES-III Distributed Computing Status”, Computer Research and Modeling, 2015, vol. 7, no. 3, pp. 469–473.

[5]. “Cloud Autonomous Grid Infrastructures for Training, Research, Development and Testing”, Scientific Report 2012-2013, 2014, LIT JINR, Dubna, pp. 33-35.

[6]. “Automatic Cloud Bursting under FermiCloud”, Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), 2013, pp. 681 - 686.

[7]. “Federated Cloud”, https://www.egi.eu/infrastructure/cloud/, дата обращения: 11.11.2015

JINR Cloud Service: Status and Perspectives

[1]N. Balashov <balashov@jinr.ru>
[1]A. Baranov <baranov@jinr.ru>
[1,2]V. Korenkov <korenkov@jinr.ru>
[1,2]N. Kutovskiy <kut@jinr.ru>
[1]A. Nechaevskiy <nechav@jinr.ru>
[1,2]R. Semenov <roman@jinr.ru>
[1]Joint Institute for Nuclear Research,
6, Joliot-Curie, Dubna, Moscow region, Russia, 141980
[2]Plekhanov Russian University of Economics,
Stremyanny lane, 36, Moscow, 117997, Russia

Abstract. Many large-scale cloud data centers have been deployed all over the world and successfully function both in business and science. Currently many scientific organizations deploy their own private clouds moving their computations and IT services into them. Cloud

technologies are actively developed and many scientific teams conduct research in the field of cloud environments structure. One of the main directions of research in cloud computing at the moment is research of methods for efficiency increase of computing resources. Development of cloud technologies led to the next stage of their evolution - integration of cloud computing environments. This imposes new requirements on used platforms as well as on basic principles and modes of operation of cloud environments integrally and raises a wide range of new problems and researches. The paper describes some basic topics and use cases of cloud computing environments in various scientific laboratories. It describes activities of the Joint Institute for Nuclear Research (JINR) aimed at the development of the cloud infrastructure (which is built completely on open-source components) deployed in the Laboratory of Information Technologies in JINR, as well as applied strategies of the improvement of efficiency, fault tolerance and reliability of the JINR cloud service, and also its integration with other cloud services.

Keywords: distributed computing; cloud computing; virtualization; datacenters.

DOI: 10.15514/ISPRAS-2015-27(6)-22

For citation: Balashov N., Baranov A., Korenkov V., Kutovskiy N., Nechaevskiy A., Semenov R. JINR Cloud Service: Status and Perspectives. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 345-354 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-22.

References

[1]. “Virtualization, clouds and IaaS at CERN”, Helge Meinhard, VTDC '12 Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing, pp. 27-28, ACM New York, NY, USA ©2012

[2]. “Automatic Cloud Bursting under FermiCloud”, Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), 2013, pp. 681 - 686.

[3]. A. V. Baranov, N. A. Balashov, N. A. Kutovskiy, R. N. Semenov, “Cloud Infrastructure at JINR”, Computer Research and Modeling, 2015, vol. 7, no. 3, pp. 463–467.

[4]. “BES-III Distributed Computing Status”, Computer Research and Modeling, 2015, vol. 7, no. 3, pp. 469–473.

[5]. “Cloud Autonomous Grid Infrastructures for Training, Research, Development and Testing”, Scientific Report 2012-2013, 2014, LIT JINR, Dubna, pp. 33-35.

[6]. “Automatic Cloud Bursting under FermiCloud”, Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), 2013, pp. 681 - 686.

[7]. “Federated Cloud”, https://www.egi.eu/infrastructure/cloud/, accessed on: 11.11.2015

Min_c: стратегия неоднородной концентрации задач для энергосберегающих компьютерных расписаний [1]

[1]*Ф. Армента-Кано <armentac@cicese.edu.mx>*
[1]*А. Черных <chernykh@cicese.mx>*
[1]*Х.М. Кортес-Мендоза <jcortes@cicese.edu.mx>*
[2]*Р. Яхьяпур <ramin.yahyapour@gwdg.de>*
[3]*А.Ю. Дроздов <alexander.y.drozdov@gmail.com>*
[4]*П. Буври <Pascal.Bouvry@uni.lu>*
[4]*Д. Клязович <Dzmitry.Kliazovich@uni.lu>*
[5]*А. Аветисян <arut@ispras.ru>*
[6]*С. Несмачнов <sergion@fing.edu.uy>*
[1]*Исследовательский центр CICESE, Энсенада, Мексика*
[2]*GWDG – Геттенгенский университет, Геттенген, Германия*
[3]*МФТИ, Москва, Россия*
[4]*Люксембургский университет, Люксембург*
[5]*ИСП РАН, Москва, Россия*
[6]*Республиканский университет, Мотневидео, Уругвай*

Аннотация. В этой статье мы описываем энергосберегающие онлайн расписания вычислительных задач и механизмы повышения энергоэффективности, учитывая конфликты использования ресурсов. Мы предлагаем модель оптимизации и новый подход к распределению задач, принимая во внимание типы приложений и их концентрацию. Разнородные задачи, решаемые на процессорах, включают в себя приложения, интенсивно использующие процессоры, диски, устройства ввода-вывода, память, сети и т.д. Когда задачи одного типа назначаются на один и тот же ресурс, они могут создать конфликты при использовании CPU, памяти, диска или сети. Это может привести к деградации общей производительности системы и увеличению потребления энергии. Мы описываем энергетические характеристики приложений, учитывая, что выполнение различных задач по-разному влияет на потребляемую мощность за счет использования разного оборудования. Мы предлагаем нелинейную гибридную модель потребления энергии, которая учитывает потребление энергии отдельных приложений и их комбинации. Мы показываем, что умные стратегии распределения задач могут

[1] Работы выполнены при финансовой поддержке Минобрнауки России (Соглашение № 02.G25.31.0061 12/02/2013).

дополнительно улучшить энергопотребление по сравнению с традиционными подходами. Мы предлагаем алгоритмы консолидации разнородных задач и показываем их эффективность на реальных данных в различных сценариях, используя CloudSim для моделирования облачных вычислений. Мы анализируем несколько алгоритмов планирования в зависимости от типа и объема информации, который они используют. Результаты детального моделирования показывают, что с точки зрения минимизации энергопотребления, стратегия, которая балансирует концентрацию задач различных типов Min_c превосходит другие алгоритмы и стабильна в различных сценариях. Эта стратегия приводит к результатам, которые доминируют почти во всех тестах.

Ключевые слова: энергосберегающие алгоритмы, типы приложений, конфликты использования ресурсов, компьютерные расписания

DOI: 10.15514/ISPRAS-2015-27(6)-23

Для цитирования: Армента-Кано Ф., Черных А., Кортес-Мендоза Х.М., Яхьяпур Р., Дроздов А.Ю., Буври П., Клязович Д., Аветисян А., Несмачнов С. Min_c: стратегия неоднородной концентрации задач для энергосберегающих компьютерных расписаний. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 355-380. DOI: 10.15514/ISPRAS-2015-27(6)-23.

1. Введение

Облачные вычисления – это глобально распределенная интернет-среда, предоставляющая разнообразные сервисы для использования вычислительных ресурсов и широко распространенная в государственных и частных организациях.

Поставщики облачных вычислений предлагают свои ресурсы и услуги с гарантией качества обслуживания (QoS). Одной из существенных эксплуатационных затрат провайдеров являются энергетические расходы.

Неэффективное управление ресурсами оказывает прямое негативное воздействие на производительность и стоимость. В распределенных средах часто бывает трудно оптимизировать потребление энергии физическими ресурсами и виртуальными машинами (VM) с различными типами задач (интенсивно использующими процессоры, диски, устройства ввода/вывода, память, сеть и т.д.). Необходимо детальное энергетическое управление ресурсами на всех уровнях выполнения задач для оптимизации их использования и увеличения прибыльности [1].

В этой статье мы предлагаем модель оптимизации, в которой принимаются во внимание различные типы приложений, и алгоритм неоднородной консолидации задач для энергосберегающего планирования. Мы оцениваем эффективность наших алгоритмов на реальных данных в различных сценариях и сравниваем их с известными алгоритмами.

Статья имеет следующую структуру: в разделе 2 приводится обзор литературы и рассматриваются алгоритмы по оптимизации

энергопотребления; в разделе 3 описываются постановка задачи и цели исследования; предлагаемые алгоритмы рассматриваются в разделе 4; в разделе 5 приводятся детали экспериментов; раздел 6 описывает методологию, используемую для анализа результатов; экспериментальные результаты приводятся в разделе 7; и, наконец, в разделе 8 описываются основные выводы и направления будущей работы.

2. Обзор литературы

Энергия, необходимая для работы компьютера, блока питания и систем охлаждения вносит существенный вклад в общий объем операционных затрат. Снижение потребления энергии стало одной из основных целей в промышленности, и задачей в научных исследованиях.

В этом разделе мы кратко обсудим известные энергосберегающие алгоритмы назначения ресурсов для выполнения задач, описанные в литературе.

EMVM - Energy-aware resource allocation heuristics for efficient management [2]. Авторы представили алгоритм назначения ресурсов с использованием динамической консолидации виртуальных машин и описали принципы эффективного использования энергии в среде облачных вычислений. В работе показано, что консолидация ведет к значительному сокращению потребления энергии по сравнению с использованием статического распределения ресурсов. Использована следующая модель энергопотребления:

$$\mathbf{P(u) = k * Pmax + (1 - k) * Pmax * u},$$

где Pmax – это максимальная потребляемая мощность, когда сервер используется полностью; k – доля мощности, потребляемой сервером в режиме ожидания (т.е. 70%); u это загрузка процессора. Общее потребление энергии E определяется следующим образом:

$$\boldsymbol{E} = \int_{\mathbf{t_0}}^{\mathbf{t_1}} \mathbf{P(u(t))dt}.$$

Когда несколько VM используют не все ресурсы, они могут быть объединены на минимальном количестве физических ресурсов. Тогда неработающие узлы могут переключаться в спящий режим, чтобы уменьшить общее потребление энергии.

Рис. 1 описывает энергопотребление в соответствии с описанной моделью, в зависимости от нагрузки процессора.

Рис. 1. Процент энергопотребления в зависимости от загрузки процессора (%).

HSFL – Hybrid Shuffled Frog Leaping алгоритм [3]. Эта схема управления ресурсами не только гарантирует качество обслуживания пользователей (QoS), указанное в соглашении об уровне обслуживания (SLA), но также обеспечивает экономию энергии. Авторы используют технологию миграции VM для консолидации ресурсов. Слабо используемые и неиспользуемые ресурсы переводятся в режим экономии энергии, обеспечивая при этом гарантии качества обслуживания. Авторы считают, что потребление энергии сервером осуществляется почти в линейной зависимости от использования процессора. Кроме того, потребление энергии в ждущем режиме составляет 70% от потребления энергии при полной нагрузке, учитывая энергию, потребляемую при миграции VM. Потребление энергии в заданный момент времени h определяется следующим образом:

$$\mathbf{E(h) = 0.7E_{max}(h) + 0.3Utlz(h)E_{max}(h) + 0.1E_{max}} \sum_{\mathbf{i \in v}} \mathbf{T(i)}.$$

$E_{max}(h)$ – это потребление энергии в режиме полной нагрузки. $Utlz(h)$ —- это средний коэффициент использования процессора за единицу времени, v – это множество мигрирующих виртуальных машин, и $T(i)$ – это время миграции VM_i. Потребление энергии в зависимости от процента использования процессора такое же, как показано на рис. 1.

AETC – Algorithm of Energy-aware Task Consolidation [4]. Авторы предлагают алгоритм консолидации задач для минимизации потребления электроэнергии. Алгоритм работает в центре обработки данных для миграции виртуальных машин, которые назначены на процессоры, находящихся в одной стойке, или на стойках, пропускная способность каналов связи между которыми относительно постоянна. Алгоритм ограничивает использование процессора не выше заданного максимального порога в 70% за счет консолидации задач на виртуальных кластерах. Кроме того, в расходах на электроэнергию учитывается время задержки в сети, необходимое для перемещения задачи в другой виртуальный кластер. Энергопотребление виртуальных машин в состоянии простоя и при передаче их по сети рассматривается как константы. Рис.2 показывает энергопотребление в зависимости от нагрузки процессора,

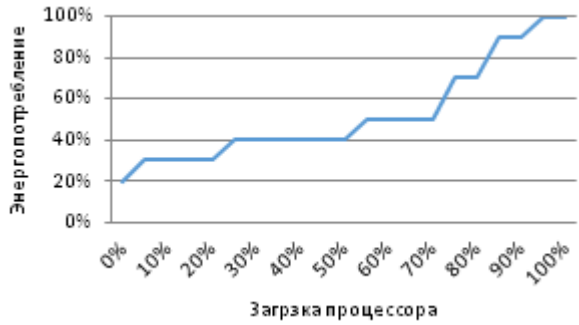


Рис. 2. Энергопотребление в зависимости от загрузки процессора (%).

Эта модель предполагает потребление энергии $E(V_i) = \alpha\ W/s$ в режиме ожидания. Когда загрузка процессора увеличивается, требуется дополнительная энергия β

$$E(Vi) = \begin{cases} \boldsymbol{\alpha\, W/s}, & \boldsymbol{if\ is\ idle} \\ \boldsymbol{\beta + \alpha\, W/s}, & \boldsymbol{if\ 0\% < CPU\ util \leq 20\%} \\ \boldsymbol{3\beta + \alpha\, W/s}, & \boldsymbol{if\ 20\% < CPU\ util \leq 50\%} \\ \boldsymbol{5\beta + \alpha\, W/s}, & \boldsymbol{if\ 50\% < CPU\ util \leq 70\%} \\ \boldsymbol{8\beta + \alpha\, W/s}, & \boldsymbol{if\ 70\% < CPU\ util \leq 80\%} \\ \boldsymbol{11\beta + \alpha\, W/s}, & \boldsymbol{if\ 80\% < CPU\ util \leq 90\%} \\ \boldsymbol{12\beta + \alpha\, W/s}, & \boldsymbol{if\ 90\% < CPU\ util \leq 100\%} \end{cases}$$

Общее потребление энергии виртуальной машиной V_i в течение периода времени $t_0 \sim t_m$ вычисляется по следующей формуле:

$$\mathbf{E_{0,m}(V_i)} = \sum_{\mathbf{t=0}}^{\mathbf{m}} \mathbf{E_t(V_i)}.$$

Для заданного виртуального кластера VC_k, состоящего из n виртуальных машин, потребление энергии в течение периода времени $t_0 \sim t_m$ рассчитывается следующим образом:

$$\mathbf{E_{0,m}(VC_k)} = \sum_{\mathbf{i=0}}^{\mathbf{n}} \mathbf{E_{0,m}(V_i)}.$$

CTES – Cooperative Two-Tier Energy-Aware Scheduling [5]. Авторы рассматривают кооперативный двухуровневый подход к планированию задач с регулированием скорости их выполнения, с целью достижения оптимального использования процессора, вместо миграции задач на другие узлы. Используются несколько стратегий планирования с прогнозом выполнения задач для оптимального назначения их на доступные виртуальные машины. Результаты моделирования показывают, что этот подход уменьшает

общее потребление энергии в облаке. Загрузка процессора определена как: $u_i(t) = \frac{ah_i(t)}{mh_i}$, где $ah_i(t)$ выделенная скорость (MIPS) процессора $host_i$ в момент времени t; mh_i - максимальная вычислительная мощность $host_i$. Авторы полагают, что неиспользуемая машина будет выключена немедленно. Таким образом, суммарная мощность процессора определяется как:

$$\mathbf{P} = \begin{cases} \mathbf{P^{static} + P^{dynamic}} & \mathbf{u > 0} \\ \mathbf{0} & \mathbf{o.w} \end{cases}$$

P^{static} – это мощность, потребляемая в течение времени простоя вычислительного узла. Она определяется как $P^{static} = \alpha P^{max}$, где P^{max} – потребляемая мощность при работе с максимальной загрузкой. Загрузка α – это постоянное соотношение между статической мощностью и максимальной мощностью ($0 < \alpha \leq 1$), которое зависит от физических свойств процессора.

Динамическое энергопотребление определяется как:

$$\mathbf{P_i^{dynamic} = \left(P_i^{max} - P_i^{static}\right)\ u_i^{\gamma}(t).}$$

Если система использует мощность P(u), энергопотребление будет $E = \int_0^{t_{min/u}} P(u)dt$, где t_{min} – это время, в течение которого процессор работает на максимальной вычислительной мощности. Доля потребления энергии в результате использования процессора похожа на показаную на рис. 1. Таким образом выполнение инструкций достигается за счет энергопотребления:

$$\mathbf{E = [\alpha + (1 - \alpha)u^{\gamma}]\frac{P^{max}t_{min}}{u}}$$

DVMA – A Decentralized Virtual Machine Migration Approach [6]. Авторы предлагают децентрализованную миграцию виртуальных машин. Они описывают модель системы и ее энергопотребления, включающие вектора загрузки, сбор информации о загрузке, выбор VM для миграции и пункт назначения миграции. Результаты оценки производительности показывают, что такой подход может обеспечить улучшение балансировки нагрузки и меньшее энергопотребление в сравнении с другими стратегиями.

Пусть α будет доля мощности, потребляемая в момент простоя по сравнению с полной загрузкой; θ – доля мощности, потребляемая в момент текущего использования процессора. Энергопотребление вычисляется следующим образом:

$$\mathbf{P_i = \alpha * P_i^{max} + (1 - \alpha) * P_i^{max} * \theta,}$$

где P_i^{max} – потребляемая мощность, когда процессор используется полностью (то есть, достигает 100% загрузки). Энергопотребление, в зависимости от использования процессора, похоже на энергопотребление, представленное на рис. 1.

EDRP – Energy and Deadline aware Resource Provisioning [7]. Авторы сосредоточились на проблеме минимизации затрат на облачные системы, повышая эффективность использования энергии, но гарантируя сроки выполнения пользовательских задач, определенных в соглашениях по качеству обслуживания (SLA). Они принимают во внимание два типа задач, независимые пакетные задачи и задачи с зависимостями.

Их модель расчета потребления электроэнергии в момент времени t включает статическое $P^x_{static}(t)$ и динамическое $P^x_{dynamic}(t)$ энергопотребление. Обе характеристики рассчитываются на основе процента загрузки процессора $Util_x(t)$, в котором учитываются только параметры используемой виртуальной машины $Q^x(t)$.

Авторы не делают различия между виртуальными машинами, на которых работают задачи, и виртуальными машинами, находящимися в ждущем режиме, поскольку фоновая деятельность процессора необходима даже в режиме простоя.

$P^x_{static}(t)$ – это константа больше нуля при $Util_x(t) > 0$ и 0 в противном случае.

Отношение между $P^x_{dynamic}(t)$ и $Util_x(t)$ является гораздо более сложным. Серверы имеют оптимальный уровень загрузки с точки зрения производительности на ватт, который определяется как Opt_x. Общепризнано, что для современных серверов $Opt_x \approx 0.7$ и увеличение энергопотребления за пределами этой операционной точки более значительно, чем при $Util_x(t) < Opt_x$.

Несмотря на идентичные условия использования, энергоэффективность различных серверов может отличаться. Это отражается в коэффициентах αx и βx, представляющих увеличение энергопотребления Dx при $Util_x(t) < Opt_x$ и $Util_x(t) \geq Opt_x$ соответственно. $P^x_{dynamic}(t)$ рассчитывается как:

$$\begin{cases} \mathbf{Util_x(t) * \alpha_x} & \mathbf{if\ (Util_x(t) < Opt_x)} \\ \mathbf{Opt_x * \alpha_x + (Util_x(t) - Opt_x)^2 * \beta_x} & \mathbf{if\ (Util_x(t) \geq Opt_x)} \end{cases}$$

Предположим, что L_{max}. – это максимальная длина расписания всех приложений. Общее потребление энергии COSP – это сумма энергопотребления всех серверов по всему интервалу времени:

$$\mathbf{COSP} = \sum_{x=1}^{M} \left(\sum_{t=1}^{L_{max}} \left(\mathbf{P^x_{static}(t) + P^x_{dynamic}(t)} \right) \right).$$

Полученный процент потребляемой мощности показан на рис. 3.

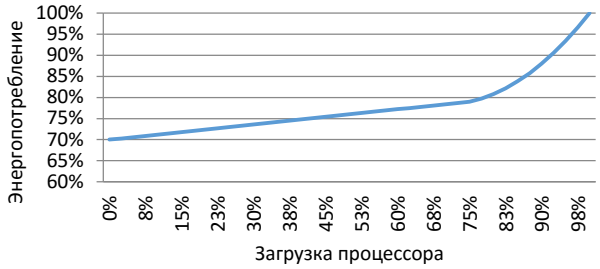


Рис. 3. Нелинейное энергопотребление в зависимости от загрузки процессора (%).

BFDP – Best Fit Decreasing Power [8]. Авторы предлагают методику расчета энергопотребления на основе полиномиальной регрессии Lasso, и алгоритма планирования ресурсов BFDP. Они направлены на повышение энергоэффективности без ухудшения качества обслуживания с учетом четырех типов задач, CPU-интенсивных, интенсивно работающих с памятью, сетью и системами ввода-вывода. Авторы вводят механизм порогов загрузки в BFDP, чтобы решить проблему чрезмерной консолидации. Результаты показали, что этот алгоритм создает меньше нарушений SLA. Нелинейная модель энергии записывается как функция:

$$\boldsymbol{y_i = \beta_0 + \sum_{j=1}^{m} \beta_j \phi_j(x_i) + \varepsilon_i,}$$

где $\phi_j(x_i)$ является ядром выражения y_i; x_i представляет загрузку процессора и памяти; β_i – это константа определяемая на основе модели обучения; ε_i – это константа. Рис. 4 показывает энергопотребление в зависимости от использования процессора и памяти.

PAHD – Power-aware Applications Hybrid Deployment [9]. Авторы рассматривают ресурсоемкие приложения и приложения, интенсивно использующие ввод/вывод для оптимизации ресурсов в виртуальных средах.

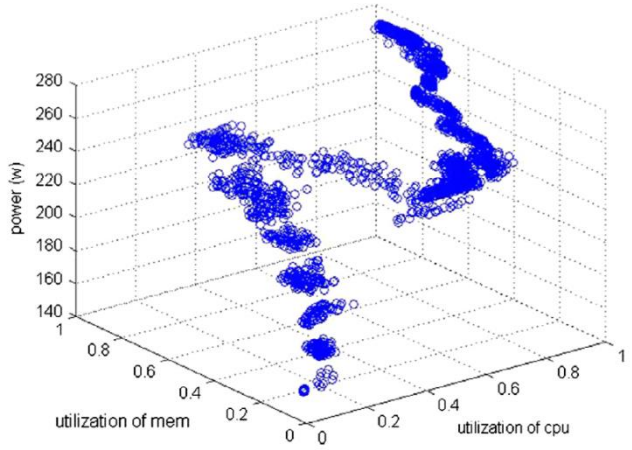


Рис. 4. Общее энергопотребление в зависимости от использования процессора и памяти [8]

Они используют кроссплатформенный гипервизор Xen для мониторинга виртуальных машин. Авторы оценивают эффективность энергосбережения в диапазоне 2%-12% для различных алгоритмов распределения ресурсов, Они приходят к выводу, что если для ресурсоемких приложений выделяется в два раза больше ресурсов, по сравнению с приложениями интенсивно использующих ввод/вывод, то достигается улучшение энергоэффективности.
Таблица 1 суммирует основные характеристики приведенных алгоритмов и критерии, используемые для оценки их производительности.

Таблица 1. Основные характеристики рассматриваемых алгоритмов.

	Применение			Характеристики												Критерии			Лит
	Центры данных	Облака	Гибридные облака	Централизованный	Децентрализованный	Динамическая загрузка	Статическая загрузка	Стат. время выполнения	Дин. время выполнения	Качество обслуживания	Миграция	Статическая	Динамическая	ЦПУ интенсивные	В/В интенсивные	Загрузка	Энергия	Срок выполнения	
EMVM		●		●		●			●	●	●		●	●		●	●	●	[1]
HSFL		●		●		●				●	●		●	●		●	●		[2]
AETC		●		●			●	●		●	●			●		●	●		[3]
CTES		●			●	●		●		●			●	●		●	●	●	[4]
DVMA		●			●					●	●			●		●	●		[5]
EDRP		●		●			●	●		●	●	●		●		●	●	●	[6]
BFDP		●		●				●		●	●		●	●	●	●	●		[7]

PAHD	•								•				•	•	•	•		[8]

3. Общая формулировка проблемы оптимизации

Мы рассматриваем m однородных серверов, описываемых множеством {s, mem, band, eff }, где s – это скорость исполнения инструкций (MIPS), mem – объем памяти (МБ), band - доступная полоса пропускания (Мбит/с), и eff – эффективность использования энергии (MIPS на ватт). Мы исходим из того, что компьютеры имеют достаточно ресурсов для выполнения задач. Основная цель предлагаемых стратегий заключается в минимизации общего энергопотребления E.

3.1 Модель задачи

Мы рассматриваем n независимых задач $J_1, J_2, \dots, J_n$, где каждая задача J_j описывается множеством $J_j = (r_j, p_j, type_j)$. $r_j \geq 0$ это время запуска задачи, p_j – время решения задачи. Время запуска задачи r_j неизвестно до момента ее запуска. $type_j$ характеризует тип задачи.

3.2 Модель энергопотребления

Мы используем нелинейную гибридную модель потребления энергии, предложенную в [25]. Мы считаем, что выполнение задач различных типов на одном и том же процессоре по-разному влияет на энергопотребление в связи с использованием различных аппаратных средств.

Наша модель учитывает энергопотребление индивидуальных задач и их комбинаций. Из-за разнообразия задач и их комбинаций, мы предлагаем использовать суммарную загрузку процессора задачами каждого типа (общей загрузкой, которая обеспечивается каждым типом задач). В этой статье мы рассматриваем 2 типа задач (тип А и тип Б).

Рис.5 описывает нормализованное энергопотребление при решении этих задач в зависимости от загрузки процессора. Назначение двух задач разного типа на одном процессоре может вызвать сниженное энергопотребление, меньшее, чем сумма энергопотребления при решении каждой задачи по отдельности.

Назначение же нескольких задач одного типа на один процессор может негативно влиять на производительность, создавая узкие места в процессоре, диске или сети, что может привести к дополнительной деградации производительности системы и увеличению энергопотребления.

Энергопотребление процессора в момент времени t состоит из двух частей – энергопотребление в режиме ожидания, когда процессор включен, но не используется $e_{idle_i}^{proc}$, и энергопотребление, когда процессор используется $e_{used_i}^{proc}(t)$:

$$\boldsymbol{e_i^{proc}(t) = o_i(t) * \left(e_{idle_i}^{proc} + e_{used_i}^{proc}(t) * U_i(t)^r\right)}$$

где $o_i(t) = 1$, если процессор включен, и $o_i(t) = 0$, в противном случае. $U_i(t)$ загрузка процессора в момент времени t. r – коэффициент, предложенный в [15], чтобы учесть нелинейные свойства энергопотребления.

$$\mathbf{e_{used_i}^{proc}(t)} = \left(\left(e_{max_i}^{proc} - e_{idle_i}^{proc} \right) * \boldsymbol{\beta}(\boldsymbol{\alpha_A(t)}) \right)$$

где $e_{max_i}^{proc}$ – это максимальная потребляемая мощность, когда процессор полностью загружен. $\beta(\alpha_A(t))$ – это коэффициент, который показывает приращение энергопотребления, когда процессор работает с различными типами приложений. Концентрация задач типа A в момент времени t определяется как $\alpha_A(t)$.

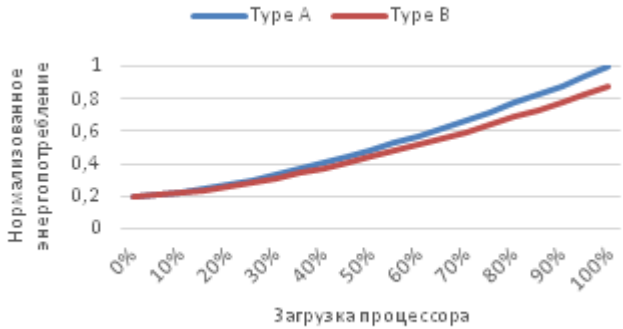


Рис. 5. Нормализованное энергопотребление задач типа A и B в зависимости от загрузки процессора (%).

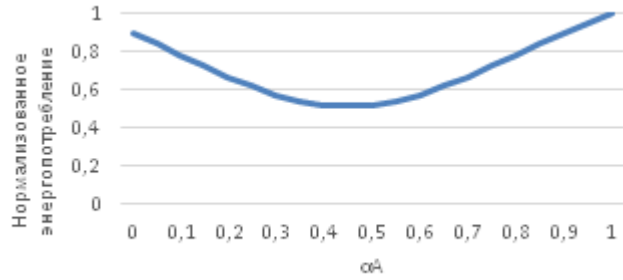


Рис. 6. Нормализованное энергопотребление в зависимости от пропорции задач типа A.

Рис. 6 показывает долю энергопотребления, когда процессор обрабатывает задачи типа А и Б.

Общая мощность, потребляемая системой подсчитывается как интеграл энергопотребления за время решения всех задач.

$$E^{op} = \int_{t=1}^{C_{max}} E^{op}(t)dt, \text{ with } E^{op}(t) = \sum_{i=1}^{m} e_i^{proc}(t)$$

Мы используем $e_{idle}{}_i^{proc} = 0.2 * e_{max}{}_i^{proc}$ и устанавливаем $\beta(\alpha_A) = 1$ для $\alpha_A = 1$ (все задачи это задачи типа А) и $\beta(\alpha_A) = 0.9$ для $\alpha_A = 0$ (все задачи это задачи типа В).

Рис. 7 показывает нормализованное энергопотребление в зависимости от загрузки процессора и доли (%) задач типа А, когда процессор обрабатывает задачи двух типов.

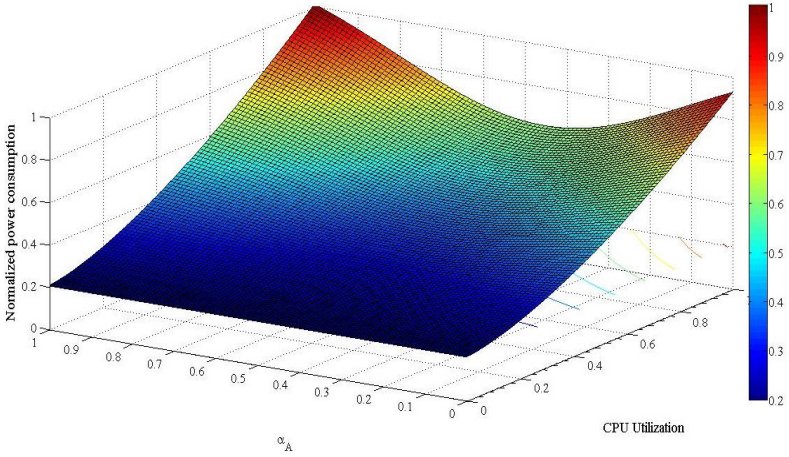


Рис. 7. Нормализованное энергопотребление задач типа А и В в зависимости от загрузки процессора (%) и пропорции задач типа А (%).

Мы видим, что когда загрузка процессора маленькая, энергопотребление низкое (голубая зона). В этом случае любая комбинация задач незначительно влияет на общее энергопотребление, и нижняя голубая зона имеет плоскую поверхность. Левая и правая зоны поверхности отражают преобладание задач типа А и задач типа Б, соответственно. В обоих случаях мы видим, что когда загрузка увеличивается до самого высокого уровня, энергопотребление также увеличивается до самого высокого уровня (красная зона). С другой стороны, если загрузка процессора сбалансирована между двумя типами задач, даже если загрузка увеличивается до самого высокого уровня (80-100%), энергопотребление остается в зеленой зоне и не достигает самых больших значений.

4. Алгоритмы планирования

В этом разделе мы описываем наш подход к планированию задач и энергосберегающие алгоритмы.

Мы используем базовый двухуровневый подход к планированию [12, 13, 14, 15]. На верхнем уровне система имеет общую информацию о задачах и доступных ресурсах, и назначает задачи на машины используя определенный критерий. Локальное распределение ресурсов для выполнения задач происходит на нижнем уровне.

Таблица 2. Стратегии назначения ресурсов.

Тип	Стратегия	Описание
Knowledge Free	Rand	Назначает задачу j на подходящую машину случайно используя равномерное распределение в диапазоне $[1..m]$.
	FFit (First Fit)	Назначает задачу j на первую машину, которая может выполнить ее.
	RR (Round Robin)	Назначает задачу j на машину, которая может выполнить ее используя стратегию Round Robin
	Min_L (Min load)	Назначает задачу j на машину с минимальной загрузкой в момент времени r_j: $min_{i=1..m}\{n_i\}$,
Energy aware	Min_Te (Min-Total_energy)	Назначает задачу j на машину с минимальным общим энерго-потреблением на момент времени r_j: $min_{i=1..m}\left(\sum_{t=1}^{r_j} e_i^{proc}(t)\right)$
	Min_e (Min-energy)	Назначает задачу j на машину с минимальным энергопотреблением в момент времени r_j: $min_{i=1..m}\left(e_i^{proc}(r_j)\right)$
Utilization aware	Min_u (Min-utilization)	Назначает задачу j на машину с минимальным уровнем загрузки (utilization) в момент времени r_j $min_{i=1..m}\left(u_i^{proc}\right)$
	Max_u (Max-utilization)	Назначает задачу j на машину с максимальной загрузкой (utilization) в момент времени r_j $max_{i=1..m}\left(u_i^{proc}\right)$
	Min_ujt (Min-util_job_type)	Назначает задачу j на машину с минимальной загрузкой (utilization) задачами того же типа в момент времени r_j
	Min_c (Min-concentration)	Назначает задачу j на машину с минимальной концентрацией задач того же типа в момент времени r_j

Процесс принятия решений о назначении задач на ресурсы основан на разных критериях. В этой статье мы изучаем 10 стратегий: Rand, FFit (First Fit), RR (Round Robin), Min_L (Min load), Min_Te (Min Total_energy), Min_e (Min energy), Min_u (Min utilization), Max_u (Max utilization), Min_ujt (Min utilization of job type), and Min_c (Min-concentration). (Таб. 2)

Мы разделяем их на три группы в зависимости от типа и количества информации, используемой для принятия решения: (1) knowledge-free, без использования информации о задачах и ресурсах [16, 17, 18]; (2) energy-aware, с информацией об энергопотреблении; (3) uilization-aware, с информацией о загрузке процессора.

5. Параметры экспериментов

В этом разделе мы описываем параметры экспериментов, включая совокупность задач, сценарии и методологию, используемую для анализа. Все эксперименты выполнены с использованием CloudSim: системы моделирования облачных вычислений, инфраструктуры и сервисов. Это стандартная программа, используемая для изучения распределения задач в облачных вычислениях. Мы расширили CloudSim для того, что включить наши алгоритмы, используя Java (JDK 7u51).

5.1 Загрузка задач

Анализ производительности алгоритмов планирования требует тщательного проведения экспериментов. Важным элементом является использование реальных задач. Производительность наших алгоритмов оценена на реальных задачах, взятых из архива задач высокопроизводительных вычислительных систем Parallel Workloads Archive [21], и вычислительных гридов Grid Workload Archive [22].

Загрузка системы (workloads) сформирована из 9 систем (traces): DAS2-University of Amsterdam, DAS2–Delft University of Technology, DAS2–Utrecht University, DAS2–Leiden University, KHT, DAS2–Vrije University Amsterdam, HPC2N, CTC, and LANL. Детальная информация о характеристике этих систем и задачах описана в [21, 22].

Характеристики задач, такие как, их распределение по дням, неделям, часовым поясам, может повлиять на правильную оценку эффективности алгоритмов. Таким образом, нам необходимо использовать нормализацию времени путем сдвига времени запуска задач на определенный интервал, чтобы обеспечить более реальные условия. Мы проводим нормализацию задач по часовым поясам и фильтрацию задач с ошибками. Мы помещаем все задачи в тот же самый часовой пояс, так что загрузка всех машин начинается в тот же самый день недели и время суток. Отметим, что выравнивание связано с местным временем, следовательно, поддерживается разница, соответствующая исходному часовому поясу. Применены несколько фильтров, чтобы удалить задачи с некорректной информацией, например, с отрицательным временем запуска, временем выполнения, количеством требуемых процессоров, пользовательским индексом и т.д. Мы также добавляем два поля в файлы загрузки для определения типа задач и загрузки процессоров.

Мы рассматриваем динамическую проблему планирования, в которой решения принимаются без полной информации о проблеме и задачах. Задачи прибывают одна за другой, время обработки задач неизвестно до тех пор, пока задача не закончится.

Рис. 8 показывает распределение задач по неделям. Он изображает общее количество задач, количество задач типа A и типа B. Мы видим, что

отсутствует преобладание одного типа задач в регистрах загрузки. В некоторые недели обрабатывается больше задач типа А, в другие – типа Б. Чтобы получить правильные статистические данные, мы используем 30 недель.

Рис. А1, А2 и А3 в Приложении показывают гистограмму общего числа задач в час, среднее число задач в день, и в час, соответственно. Рис. А4 и А5 в Приложении демонстрируют распределение каждого вида задач в неделю и в час для более детального анализа.

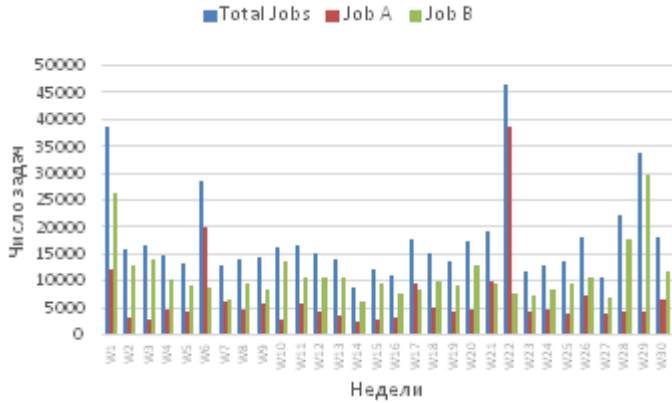


Рис. 8. Общее число задач, число задач типа A и типа B по неделям.

5.2 Сценарии

Следуя статье [11], которая описывает энергопотребление процессора Fujitsu PRIMERGY TX300 S7, мы используем $e_{max_i}^{proc} = 300$, $e_{idle_i}^{proc} = 0.2 * e_{max_i}^{proc}$ и устанавливаем нелинейное свойство энергопотребления $r = 1.5$.

6. Методология анализа

6.1 Деградация производительности

Чтобы обеспечить правильный выбор наилучшей стратегии, мы проводим анализ энергопотребления на основе методологии средней деградации, предложенной Tsafrir [24], и примененной для планирования задач в [12, 17, 20].

Прежде всего, мы оцениваем деградацию в производительности (относительную ошибку) каждой стратегии для каждой метрики. Эта деградация оценивается относительно наилучшего результата полученного всеми стратегиями для каждой метрики:

$$(\boldsymbol{\gamma - 1}) * \mathbf{100} \text{ with } \boldsymbol{\gamma} = \frac{\boldsymbol{strategy\ metric\ value}}{\boldsymbol{best\ found\ metric\ value}}$$

Затем мы усредняем эти значения и ранжируем стратегии. Наилучшая стратегия самой низкой средней деградации имеет ранг 1. Отметим, что мы пытаемся найти стратегию, которая работает эффективно в разных сценариях, таким образом, мы пытаемся найти компромисс, который учитывает все условия. Например, ранг стратегии может быть разным в разных сценариях, поэтому мы подсчитываем среднюю деградацию, чтобы оценить производительность стратегий и показать есть ли стратегии, которые доминируют над другими. Этот подход анализирует результаты на основе средних значений, однако, небольшая часть данных с большим разбросом может повлиять на средние значения. Для того, чтобы лучше интерпретировать данные, мы используем профили производительности наших стратегий.

6.2 Профиль производительности

Профиль производительности $\rho(\tau)$ это неубывающая, кусочно-постоянная функция, которая представляет вероятность того, что отношение γ находится в факторе τ от лучшего значения [23]. Функция $\rho(\tau)$ это накапливающая функция распределения. Стратегии с большой вероятностью $\rho(\tau)$ при малых τ будут предпочтительнее.

7. Экспериментальный анализ

В этом разделе мы приводим результаты экспериментального анализа предлагаемых стратегий. Во-первых, мы оцениваем их на основе методологии деградации производительности, описанной в разделе 6.1, и ранжируем их. Затем, мы проводим более детальный анализ, основанный на профилях производительностей стратегий.

7.1 Деградация энергопотребления

Рис. 9 показывает среднюю деградацию энергопотребления эвристик за неделю. Небольшой процент деградации показывает, что стратегия получает результаты, которые близки к наилучшем результатам, полученными всеми стратегиями. Таким образом, маленькая деградация демонстрирует лучшие результаты.

Мы наблюдаем, что деградации сильно различаются в зависимости от недели и стратегии. Однако, Max_u показывает худшее поведение среди всех, а Min_c и Min_ujt - лучшее. Последние две стратегии доминируют во всех тестах.

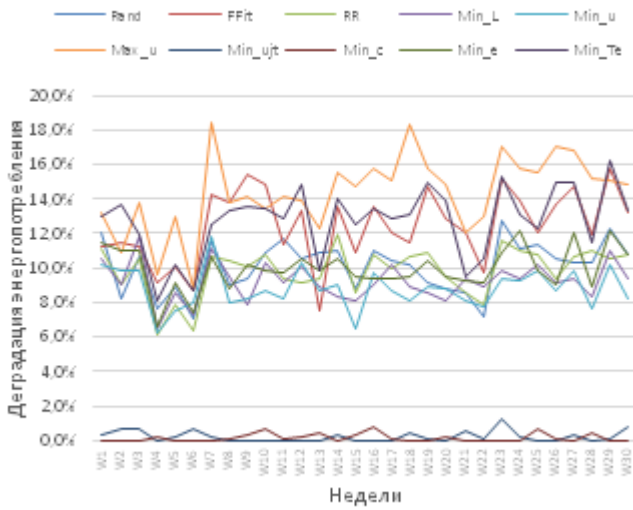


Рис. 9. Деградация энергопотребления по неделям.

Рис. 10 показывает среднюю деградацию энергопотребления по всем тестам. Стратегии Max_u, Min_Te и FFit показывают наихудшие значения деградации и ранжируются 10, 9, 8, соответственно. Min_c и Min_ujt являются наилучшими стратегиями с наименьшей деградацией и имеют ранги 1 и 2, соответственно.

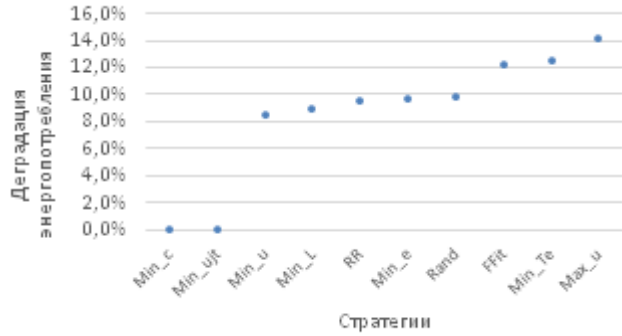


Рис. 10. Средняя деградация энергопотребления.

7.2. Профиль производительности

Как упоминалось в разделе 6, использование средних значений, может привести к неправильным выводам, так как небольшая часть результатов с большими отклонениями может существенно изменить среднее значение. Для того, чтобы проанализировать результаты более детально, мы учитываем профили производительности наших стратегий.

Рис. 11 показывает профили энергопотребления наших 10 стратегий в интервале τ=[0,...,0.2]. Мы видим больший разброс в деградации энергопотребления на значительном количестве тестов.

Min_c имеет наивысший ранг и самую высокую вероятность быть наилучшей стратегией. Если мы хотим получить результатах не хуже чем 1% (в факторе τ=0,01) от лучших найденных, вероятность того, что эта стратегия наилучшая для данной проблемы близка к 1. Min_ujt имеет второй ранг, вероятность того, что это лучшая стратегия с фактором 0.01 равна 0,96.

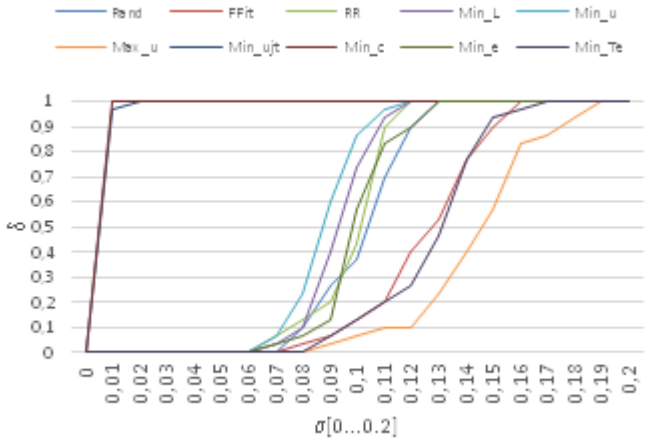


Рис. 11. Профиль деградации энергопотребления 10 стратегий.

8. Заключение

В этой статье мы предлагаем новый подход к распределению ресурсов с учетом характеристик задач. Основная идея нашего подхода основана на том факте, что различные задачи требуют различных ресурсов. Они могут быть; вычислительно-интенсивные (computing bound), интенсивно использующие процессор; интенсивно использующих ввод/вывод (I/O bound), требующие высокой пропускной способности; memory bound, disk bound, и тд.

Когда задачи одного типа назначаются на один и тот же ресурс, они могут создать узкие места и конфликты по использованию ресурсов в процессоре, в

памяти, на диске или в сети. Это может привести к деградации производительности и увеличению энергопотребления.

Современные планировщики не используют информацию о типах задач. Мы предлагаем использовать эту информацию контролируемым способом, для того, чтобы использовать ресурсы более эффективно и снизить энергопотребление.

Мы спроектировали и проанализировали новые алгоритмы планирования сосредоточившись на стратегиях назначения задач на вычислительные, которые принимают во внимание как информацию о динамическом состоянии ресурсов так и о типе задач.

Основные результаты можно сформулировать следующим образом.

(a) Мы сформулировали проблему распределения задач с различными характеристиками для оптимизации энергопотребления;

(b) Мы предложили и провели всестороннее исследование производительности 10 стратегий используя систему моделирования облачных вычислений CloudSim на реальных данных в различных сценариях.

(c) Для того, чтобы обеспечить выбор наилучшей стратегии, мы применили анализ на основе методологии оценки деградации производительности каждой стратегии и использовали их профили производительности.

(d) Мы обнаружили, что информация о загрузке процессора без информации о типе задач не помогает существенно улучшить его энергопотребление.

(e) Основываясь на этих результатах, мы показали, что, учитывая тип задач, мы можем оптимизировать их распределение, тем самым снизить энергопотребление и увеличить производительность, избегая создания узких мест и конфликтов по ресурсам.

(f) Результаты детального моделирования, представленные в статье, показывают, что с точки зрения минимизации энергопотребления, стратегия, которая балансирует концентрацию задач различных типов Min_c превосходит другие алгоритмы. Эта стратегия приводит к результатам, которые доминирую почти во всех тестах. Мы пришли к выводу, что стратегия стабильна в различных условиях. Она обеспечивает незначительную деградацию производительности при различных сценариях.

Тем не менее, необходимо дальнейшее изучение предлагаемого подхода для оценки потребления энергии. Учет нескольких типов задач и их концентраций на реальных вычислительных ресурсах обязателен для оценки фактической эффективности стратегий и предложенного метода.

Важно тщательное профилирование и изучение характеристик задач разных типов. Кроме того, необходимо анализировать типы приложений и

разработать методологию разделения их на категории. Это будет предметом будущей работы для лучшего понимания типов задач и их влияния на энергопотребление.

Приложение

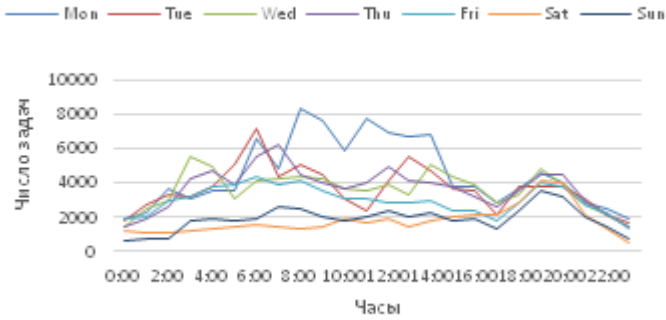


Рис. А1. Общее число задач в час в разные дни недели.

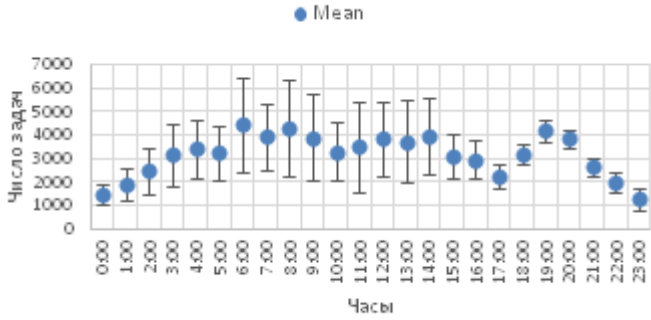


Рис. А2. Среднее число задач в день в течении часа.

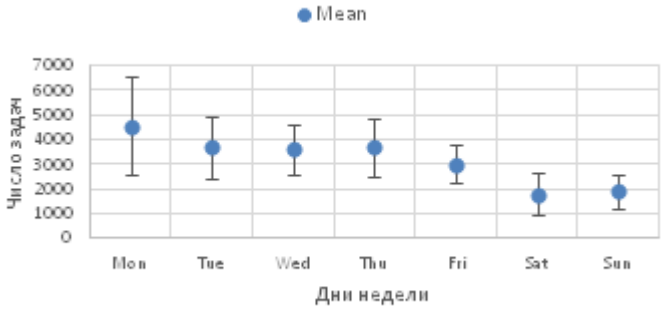


Рис. A3. Среднее число задач в день в течении недели

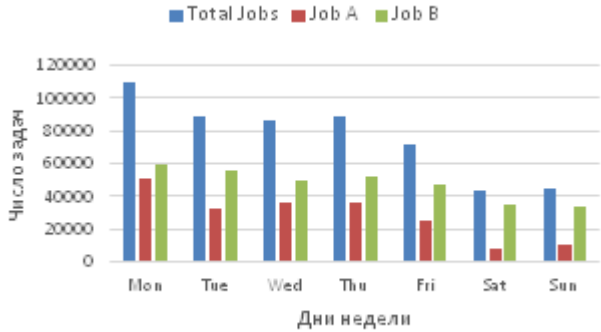


Рис. A4.Общее число задач типа A и B в день.

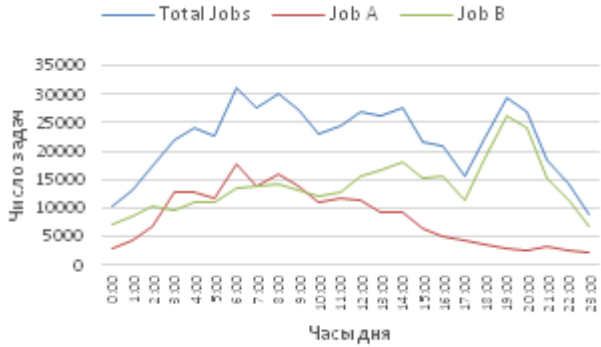


Рис. A5. Общее число задач типа A и B в час.

Литература

[1]. D. Kliazovich, J. E. Pecero, A. Tchernykh, P. Bouvry, S. U. Khan, A. Y. Zomaya, CA-DAG: Modeling Communication-Aware Applications for Scheduling in Cloud Computing, Journal of Grid Computing, 2015.

[2]. A. Beloglazov, J. Abawajy, and R. Buyya, Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing, Future Gener. Comput. Syst., vol. 28, no. 5, pp. 755–768, May 2012.

[3]. J. Luo, X. Li, and M. Chen, Hybrid shuffled frog leaping algorithm for energy-efficient dynamic consolidation of virtual machines in cloud data centers, Expert Syst. Appl., vol. 41, no. 13, pp. 5804–5816, Oct. 2014.

[4]. C.-H. Hsu, K. D. Slagter, S.-C. Chen, and Y.-C. Chung, Optimizing Energy Consumption with Task Consolidation in Clouds, Inf. Sci., vol. 258, pp. 452–462, Feb. 2014.

[5]. S. Hosseinimotlagh, F. Khunjush, and S. Hosseinimotlagh, A Cooperative Two-Tier Energy-Aware Scheduling for Real-Time Tasks in Computing Clouds, in Proceedings of the 2014 22Nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Washington, DC, USA, 2014, pp. 178–182.

[6]. X. Wang, X. Liu, L. Fan, and X. Jia, A Decentralized Virtual Machine Migration Approach of Data Centers for Cloud Computing, Math. Probl. Eng., vol. 2013, p. e878542, Aug. 2013.

[7]. Y. Gao, Y. Wang, S. K. Gupta, and M. Pedram, An Energy and Deadline Aware Resource Provisioning, Scheduling and Optimization Framework for Cloud Systems," in Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, Piscataway, NJ, USA, 2013, pp. 31:1–31:10.

[8]. L. Luo, W. Wu, W. T. Tsai, D. Di, and F. Zhang, Simulation of power consumption of cloud data centers, Simul. Model. Pract. Theory, vol. 39, pp. 152–171, Dec. 2013.

[9]. Z. Liu, R. Ma, F. Zhou, Y. Yang, Z. Qi, and H. Guan, "Power-aware I/O-Intensive and CPU-Intensive applications hybrid deployment within virtualization environments," in 2010 IEEE International Conference on Progress in Informatics and Computing (PIC), 2010, vol. 1, pp. 509–513.

[10]. A. Lezama, A. Tchernykh, R. Yahyapour, Performance Evaluation of Infrastructure as a Service Clouds with SLA Constraints. Computación y Sistemas 17(3): 401–411 (2013).

[11]. S. B. Matthias Splieth, "Analyzing the Effect of Load Distribution Algorithms on Energy Consumption of Servers in Cloud Data Centers," 2015.

[12]. A. Tchernykh, L. Lozano, U. Schwiegelshohn, P. Bouvry, J. Pecero, S. Nesmachnow: Energy-Aware Online Scheduling: Ensuring Quality of Service for IaaS Clouds. International Conference on High Performance Computing & Simulation (HPCS 2014), pp 911–918, Bologna, Italy (2014).

[13]. A. Tchernykh, U. Schwiegelsohn, R. Yahyapour, N. Kuzjurin: Online Hierarchical Job Scheduling on Grids with Admissible Allocation, Journal of Scheduling 13(5):545–552 (2010)

[14]. A. Tchernykh, J. Ramírez, A. Avetisyan, N. Kuzjurin, D. Grushin, S. Zhuk,: Two Level Job-Scheduling Strategies for a Computational Grid. In R. Wyrzykowski et al. (eds.) Parallel Processing and Applied Mathematics, 6th International Conference on Parallel Processing and Applied Mathematics. Poznan, Poland, 2005, LNCS 3911, pp. 774–781, Springer-Verlag (2006).

[15]. B. Dorronsoro, S. Nesmachnow, J. Taheri, A. Zomaya, E-G. Talbi, P. Bouvry: A hierarchical approach for energy-efficient scheduling of large workloads in multicore distributed systems. Sustainable Computing: Informatics and Systems 4:252–261 (2014).

[16]. A. Tchernykh, J. Pecero, A. Barrondo, E. Schaeffer: Adaptive Energy Efficient Scheduling in Peer-to-Peer Desktop Grids, Future Generation Computer Systems, 36:209–220 (2014).

[17]. J.M. Ramírez, A. Tchernykh, R. Yahyapour, U. Schwiegelshohn, A. Quezada, J. González, A. Hirales: Job Allocation Strategies with User Run Time Estimates for Online Scheduling in Hierarchical Grids. Journal of Grid Computing 9:95–116 (2011).

[18]. S. Iturriaga, S. Nesmachnow, B. Dorronsoro, P. Bouvry: Energy efficient scheduling in heterogeneous systems with a parallel multiobjective local search. Computing and Informatics 32(2):273–294 (2013)

[19]. U. Schwiegelshohn, A. Tchernykh: Online Scheduling for Cloud Computing and Different Service Levels, 26th Int. Parallel and Distributed Processing Symposium Los Alamitos, CA, pp. 1067–1074 (2012).

[20]. A. Tchernykh, L. Lozano, U. Schwiegelshohn, P. Bouvry, J. Pecero, S. Nesmachnow, A. Drozdov: Online Bi-Objective Scheduling for IaaS Clouds with Ensuring Quality of Service. Journal of Grid Computing, Springer-Verlag, DOI 10.1007/s10723-015-9340-0 (2015).

[21]. Parallel Workload Archive [Online, November 2014]. Available at http://www.cs.huji.ac.il/labs/parallel/ workload

[22]. Grid Workloads Archive [Online, November 2014]. Available at http://gwa.ewi.tudelft.nl

[23]. E. Zitzler: Evolutionary algorithms for multiobjective optimization: Methods and applications, PhD thesis, Swiss Federal Institute of Technology. Zurich (1999)

[24]. D. Tsafrir, Y. Etsion, D. Feitelson: Backfilling Using System-Generated Predictions Rather than User Runtime Estimates. *IEEE Transactions on Parallel and Distributed Systems* 18 (6), pp.789–803 (2007)

[25]. F. Armenta-Cano, A. Tchernykh, J. M. Cortés-Mendoza, R. Yahyapour, A. Drozdov, P. Bouvry, D. Kliazovich, A. Avetisyan: Heterogeneous Job Consolidation for Power Aware Scheduling with Quality of Service. Proceedings of the 1st Russian Conference on Supercomputing - Supercomputing Days 2015, Moscow, Russia, September 28-29, 2015. Editors V. Voevodin, S. Sobolev. Published on CEUR-WS: 22-Oct-2015, Vol-1482, p. 687-697. ONLINE: http://ceur-ws.org/Vol-1482/, URN: urn:nbn:de:0074-1482-7

Min_c: Heterogeneous Concentration Policy for Power Aware Scheduling[2]

[1]F. Armenta-Cano <armentac@cicese.edu.mx>
[1]A. Tchernykh <chernykh@cicese.mx>
[1]J. M. Cortés-Mendoza <jcortes@cicese.edu.mx>
[2]R. Yahyapour <ramin.yahyapour@gwdg.de>
[3]A. Yu. Drozdov <alexander.y.drozdov@gmail.com>
[4]P. Bouvry <Pascal.Bouvry@uni.lu>
[4]D. Kliazovich <Dzmitry.Kliazovich@uni.lu>
[5]A. Avetisyan <arut@ispras.ru>
[6]S. Nesmachnow <sergion@fing.edu.uy>
[1] CICESE Research Center, Carretera Ensenada-Tijuana No. 3918, Zona Playitas, Código Postal 22860, Apdo. Postal 360, Ensenada, B.C. México
[2]GWDG – University of Göttingen, Wilhelmsplatz 1, 37073 Göttingen, Göttingen, Germany
[3] Moscow Institute of Physics and Technology (State University) 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4]University of Luxembourg. Maison du Savoir, 2, Avenue de l'Université, L-4365 Esch-sur-Alzette, Luxembourg
[5]Institute for System Programming of the RAS, 25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[6]Universidad de la República, Av. 18 de Julio 1968, 11200 Montevideo, Uruguay

Abstract. In this paper, we address power-aware online scheduling of jobs with resource contention. We propose an optimization model and present a new approach to resource allocation based on job concentration. We take into account different types of applications and heterogeneity of workloads that could include CPU-intensive, disk-intensive, I/O-intensive, memory-intensive, network-intensive and other applications. When jobs of one type are allocated to the same resource, they may create a bottleneck and resource contention either in CPU, memory, disk or network. It may result in system performance degradation and increasing energy consumption. The main objective is to minimize the total energy consumption of running heterogeneous workloads. We focus on energy characteristics of applications assuming that applications of different types contribute differently to the total power consumptions due to use different hardware. We propose a nonlinear hybrid model of energy consumption. Our model takes into account power consumption of individual jobs and their combinations. We propose heterogeneous job consolidation algorithms and validate them by conducting a performance evaluation study using the CloudSim toolkit under different scenarios and real data. We analyze several scheduling algorithms depending on the type and amount of information they require. We show that information about resources utilization without knowledge of jobs types does not help much to improve the total energy

[2] The work is partially supported by the Ministry of Education and Science of Russian Federation under contract No02.G25.31.0061 12/02/2013 (Government Regulation No 218 from 09/04/2010).

consumption. In the other hand, being aware of types of applications, intelligent allocation strategies can further improve energy consumption compared with traditional approaches.

Keywords: Energy efficiency, type of applications, resource contention, scheduling.

DOI: 10.15514/ISPRAS-2015-27(6)-23

For citation: Armenta-Cano F., Tchernykh A., Cortés-Mendoza J. M., Yahyapour R., Drozdov A.Yu., Bouvry P., Kliazovich D., Avetisyan A., Nesmachnow S. Min_c: Heterogeneous Concentration Policy for Power Aware Scheduling. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 355-380 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-23.

References

[1]. D. Kliazovich, J. E. Pecero, A. Tchernykh, P. Bouvry, S. U. Khan, A. Y. Zomaya, CA-DAG: Modeling Communication-Aware Applications for Scheduling in Cloud Computing, Journal of Grid Computing, 2015.

[2]. A. Beloglazov, J. Abawajy, and R. Buyya, Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing, Future Gener. Comput. Syst., vol. 28, no. 5, pp. 755–768, May 2012.

[3]. J. Luo, X. Li, and M. Chen, Hybrid shuffled frog leaping algorithm for energy-efficient dynamic consolidation of virtual machines in cloud data centers, Expert Syst. Appl., vol. 41, no. 13, pp. 5804–5816, Oct. 2014.

[4]. C.-H. Hsu, K. D. Slagter, S.-C. Chen, and Y.-C. Chung, Optimizing Energy Consumption with Task Consolidation in Clouds, Inf. Sci., vol. 258, pp. 452–462, Feb. 2014.

[5]. S. Hosseinimotlagh, F. Khunjush, and S. Hosseinimotlagh, A Cooperative Two-Tier Energy-Aware Scheduling for Real-Time Tasks in Computing Clouds, in Proceedings of the 2014 22Nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Washington, DC, USA, 2014, pp. 178–182.

[6]. X. Wang, X. Liu, L. Fan, and X. Jia, A Decentralized Virtual Machine Migration Approach of Data Centers for Cloud Computing, Math. Probl. Eng., vol. 2013, p. e878542, Aug. 2013.

[7]. Y. Gao, Y. Wang, S. K. Gupta, and M. Pedram, An Energy and Deadline Aware Resource Provisioning, Scheduling and Optimization Framework for Cloud Systems," in Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, Piscataway, NJ, USA, 2013, pp. 31:1–31:10.

[8]. L. Luo, W. Wu, W. T. Tsai, D. Di, and F. Zhang, Simulation of power consumption of cloud data centers, Simul. Model. Pract. Theory, vol. 39, pp. 152–171, Dec. 2013.

[9]. Z. Liu, R. Ma, F. Zhou, Y. Yang, Z. Qi, and H. Guan, "Power-aware I/O-Intensive and CPU-Intensive applications hybrid deployment within virtualization environments," in 2010 IEEE International Conference on Progress in Informatics and Computing (PIC), 2010, vol. 1, pp. 509–513.

[10]. A. Lezama, A. Tchernykh, R. Yahyapour, Performance Evaluation of Infrastructure as a Service Clouds with SLA Constraints. Computación y Sistemas 17(3): 401–411 (2013).

[11]. S. B. Matthias Splieth, "Analyzing the Effect of Load Distribution Algorithms on Energy Consumption of Servers in Cloud Data Centers," 2015.

[12]. A. Tchernykh, L. Lozano, U. Schwiegelshohn, P. Bouvry, J. Pecero, S. Nesmachnow: Energy-Aware Online Scheduling: Ensuring Quality of Service for IaaS Clouds. International Conference on High Performance Computing & Simulation (HPCS 2014), pp 911–918, Bologna, Italy (2014).

[13]. A. Tchernykh, U. Schwiegelsohn, R. Yahyapour, N. Kuzjurin: Online Hierarchical Job Scheduling on Grids with Admissible Allocation, Journal of Scheduling 13(5):545–552 (2010)

[14]. A. Tchernykh, J. Ramírez, A. Avetisyan, N. Kuzjurin, D. Grushin, S. Zhuk,: Two Level Job-Scheduling Strategies for a Computational Grid. In R. Wyrzykowski et al. (eds.) Parallel Processing and Applied Mathematics, 6th International Conference on Parallel Processing and Applied Mathematics. Poznan, Poland, 2005, LNCS 3911, pp. 774–781, Springer-Verlag (2006).

[15]. B. Dorronsoro, S. Nesmachnow, J. Taheri, A. Zomaya, E-G. Talbi, P. Bouvry: A hierarchical approach for energy-efficient scheduling of large workloads in multicore distributed systems. Sustainable Computing: Informatics and Systems 4:252–261 (2014).

[16]. A. Tchernykh, J. Pecero, A. Barrondo, E. Schaeffer: Adaptive Energy Efficient Scheduling in Peer-to-Peer Desktop Grids, Future Generation Computer Systems, 36:209–220 (2014).

[17]. J.M. Ramírez, A. Tchernykh, R. Yahyapour, U. Schwiegelshohn, A. Quezada, J. González, A. Hirales: Job Allocation Strategies with User Run Time Estimates for Online Scheduling in Hierarchical Grids. Journal of Grid Computing 9:95–116 (2011).

[18]. S. Iturriaga, S. Nesmachnow, B. Dorronsoro, P. Bouvry: Energy efficient scheduling in heterogeneous systems with a parallel multiobjective local search. Computing and Informatics 32(2):273–294 (2013)

[19]. U. Schwiegelshohn, A. Tchernykh: Online Scheduling for Cloud Computing and Different Service Levels, 26th Int. Parallel and Distributed Processing Symposium Los Alamitos, CA, pp. 1067–1074 (2012).

[20]. A. Tchernykh, L. Lozano, U. Schwiegelshohn, P. Bouvry, J. Pecero, S. Nesmachnow, A. Drozdov: Online Bi-Objective Scheduling for IaaS Clouds with Ensuring Quality of Service. Journal of Grid Computing, Springer-Verlag, DOI 10.1007/s10723-015-9340-0 (2015).

[21]. Parallel Workload Archive [Online, November 2014]. Available at http://www.cs.huji.ac.il/labs/parallel/ workload

[22]. Grid Workloads Archive [Online, November 2014]. Available at http://gwa.ewi.tudelft.nl

[23]. E. Zitzler: Evolutionary algorithms for multiobjective optimization: Methods and applications, PhD thesis, Swiss Federal Institute of Technology. Zurich (1999)

[24]. D. Tsafrir, Y. Etsion, D. Feitelson: Backfilling Using System-Generated Predictions Rather than User Runtime Estimates. *IEEE Transactions on Parallel and Distributed Systems* 18 (6), pp.789–803 (2007)

[25]. F. Armenta-Cano, A. Tchernykh, J. M. Cortés-Mendoza, R. Yahyapour, A. Drozdov, P. Bouvry, D. Kliazovich, A. Avetisyan: Heterogeneous Job Consolidation for Power Aware Scheduling with Quality of Service. Proceedings of the 1st Russian Conference on Supercomputing - Supercomputing Days 2015, Moscow, Russia, September 28-29, 2015. Editors V. Voevodin, S. Sobolev. Published on CEUR-WS: 22-Oct-2015, Vol-1482, p. 687-697. ONLINE: http://ceur-ws.org/Vol-1482/, URN: urn:nbn:de:0074-1482-7

Дерандомизационная криптостойкость гомоморфного шифрования[1]

А.В. Трепачева <alina1989malina@ya.ru>
Южный федеральный университет,
344006, Россия, г. Ростов-на-Дону, ул. Большая Садовая, д. 105/42

Аннотация. В статье освещается проблематика построения и анализа систем криптографической защиты облачных вычислений на основе гомоморфного шифрования. Рассматриваются минимальные требования, которым должна удовлетворять гомоморфная криптосистема, чтобы быть пригодной для практического использования. Для этого вводится новое понятие – шифрование, стойкое к дерандомизации, а также объясняются связи этого понятия с классическими общепринятыми определениями криптостойкости, а также с защищенностью в целом облачной системы. Показываются примеры простых гомоморфных криптосистем, как удовлетворяющие требованию стойкости к дерандомизации, так и не обладающие этим свойством. В заключение делается вывод о применимости данных криптосистем в облачных вычислительных системах.

Ключевые слова: защита информации; вычисления над зашифрованными данными; гомоморфное шифрование; криптоанализ; дерандомизация.

DOI: 10.15514/ISPRAS-2015-27(6)-24

Для цитирования: Трепачева А.В. Дерандомизационная криптостойкость гомоморфного шифрования. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 381-394. DOI: 10.15514/ISPRAS-2015-27(6)-24.

1. Введение

Среди проблем, с которыми сталкиваются облачные вычисления, одними из самых актуальных и в то же время сложных для решения, являются проблемы обеспечения информационной безопасности. «Как организовать сложную обработку данных в облаке и при этом гарантировать конфиденциальность и правильность вычислений?», «Как минимизировать коммуникационную сложность используемых протоколов?», «Как обеспечить надежность конфи-

[1] Работа выполнена при поддержке гранта РФФИ №15-07-00597 А «Разработка и исследование алгоритмов полностью гомоморфного шифрования»

денциальных вычислений?» – все это вопросы, на которые до сих пор нет удовлетворительных ответов.

Причина этого состоит в том, что многие угрозы исходят от злонамеренных администраторов облачных серверов, а также от хакеров которые несмотря на защитные экраны и контрмеры получают доступ к данным, а специализированные криптосистемы, позволяющие организовать вычисления над зашифрованными данными (а также доказательство правильности этих вычислений) без их расшифрования пока что недостаточно изучены и мало применяются на практике. Известные применения включают проекты CryptDB [1], Cipherbase [2], BigQuery [3], Always Encrypted [4], защищенная облачная БД исследователей из Новосибирска [5].

Что же это за шифрование, которое позволяет проводить вычисления над зашифрованными данными без расшифрования? Самым общим классом таких систем является т.н. шифрование, сохраняющее свойства (англ. Property-preserving encryption, PPE), сокращенно ШСС. ШСС позволяет проводить вычисления над зашифрованными данными так, что только владелец секретного ключа может воспользоваться их результатами и получить их так, как если бы все эти вычисления проводились на его локальном компьютере над незашифрованными данными. Простейшим примером такого шифрования является детерминированное шифрование, или шифр простой замены. Криптограммы такого шифра можно сравнивать на равенство без расшифрования. Более сложные примеры включают шифрование, сохраняющее порядок (ШСП) [6], поисковое шифрование (ПШ), гомоморфное шифрование (ГШ) [7]. Среди криптосистем гомоморфного шифрования выделяют т.н. полностью гомоморфное шифрование (ПГШ), которое позволяет *теоретически* проводить *любые* вычисления над зашифрованными данными без их расшифрования.

Например, в архитектуре CryptDB сочетается использование трёх видов шифрования для поддержания всей функциональности: традиционное блочное шифрование, гомоморфное шифрование и шифрование, сохраняющее порядок. Однако поддерживаемая функциональность оставляет желать лучшего: например, невозможно сделать какие-либо более-менее сложные арифметические (например, статистические) вычисления над данными – их можно только просуммировать или умножить на какую-то фиксированную открытую (известную) константу. Это происходит вследствие того, что используемая в CryptDB гомоморфная криптосистема – криптосистема Пэйе [7] имеет такой ограниченный набор функциональных возможностей. Такое решение было принято для гарантирования безопасности, поскольку эта криптосистема считается обладающей достаточным уровнем криптостойкости. Впрочем, в настоящее время существует уже довольно большое количество криптосистем, поддерживающих бóльшие гомоморфные свойства и обладающих какими-либо доказанными в соответствии со стандартными криптографическими предположениям свойствами криптостойкости. Проблема, однако, состоит в

том, чтобы определить, *насколько безопасно применять ту или иную гомоморфную криптосистему в конкретной практической ситуации*?

Таким образом, в данной работе рассматривается следующая задача: предложить критерии, позволяющие *относительно легко* определить пригодность гомоморфной криптосистемы для использования на практике и проверить некоторые известные гомоморфные криптосистемы на удовлетворение этим критериям. В частности, предлагается некоторая новая модель криптостойкости, вводится понятие *дерандомизации (гомоморфной) криптосистемы*, которое является частным случаем понятия дерандомизации в криптографии [8]. В этой модели исследованы криптосистемы MORE и PORE, авторы которых – Эвиад Кипнис (Aviad Kipnis) и Элифас Хибшуш (Eliphas Hibshoosh) – рассмотрели сводимость атаки на них только по шифртекстам к решению квадратного уравнения по модулю труднофакторизуемого числа [9]. Данный выбор обусловлен тем, что для этих криптосистем есть точные сведения относительно стойкости по классическим атакам, а также и то, что в патенте [10] свои криптосистемы авторы назвали именно «методами полностью гомоморфной рандомизации». Также в этой модели исследованы криптосистемы, предложенные исследователями из Новосибирска [11].

Общая структура работы такова: в разделе 2 приведены некоторые необходимые в дальнейшем сведения и обозначения; в разделе 3 вводится формальное описание дерандомизационной стойкости после некоторого интуитивного пояснения. В разделе 4 приведено заключение.

2. Предварительные сведения и обозначения

Формально, криптосистема состоит из тройки алгоритмов $(\mathrm{KeyGen}, \mathrm{Enc}, \mathrm{Dec})$. Вероятностный алгоритм KeyGen принимает на вход параметр уровня криптостойкости λ и выдает в качестве результата пару ключей – ключ зашифрования и ключ расшифрования. Алгоритмы Enc и Dec принимают на вход, соответственно, открытый текст вместе с ключом зашифрования и шифртекст вместе с ключом расшифрования (вычислительная сложность всех этих алгоритмов должна быть полиномиальна от параметра уровня криптостойкости λ). В зависимости от того, является ли алгоритм Enc вероятностным или детерминированным, вся криптосистема в целом будет являться вероятностной или детерминированной. В гомоморфной криптосистеме к этой тройке алгоритмов добавляется еще алгоритм Eval, который принимает на вход шифртексты, и в качестве результата выдает шифртекст, производя вычисления над зашифрованными данными. Как правило, гомоморфные криптосистемы допускают выполнение какого-то ограниченного набора арифметических операций над зашифрованными данными.

Проблема построения полностью гомоморфного шифрования (ПГШ), обеспечивающего гомоморфное сложение и умножение зашифрованных данных при неограниченном числе операций была поставлена впервые в работе [12], и оставалась в основном открытой до появления в свет работы Крейга Джентри в 2009 году (отличительное качество ПГШ состоит в том, что оно позволяет вычислить *любую* формулу над зашифрованными данными, поскольку любая формула выражается через сложения и умножения операндов). Впрочем, предложенное решение проблемы было эффективно лишь теоретически, поэтому вскоре после публикации Крейгом Джентри работы [13] стали появляться другие конструкции полностью гомоморфных криптосистем. Для большинства из них делались попытки обосновать их криптостойкость через сложность задач решения систем уравнений или факторизации чисел при атаке с известным открытым текстом. Эта атака является на сегодняшний день де-факто стандартом, позволяющим оценить качество ПГШ в целом, поскольку в силу гомоморфных свойств и свойства податливости (malleability) из произвольного шифртекста возможно получение шифртекста нуля.

Такая криптостойкость носит также название семантической криптостойкости и проверяется, например, через т.н. «игру» $\mathrm{DSemGame}$.

Определение 2.1. *(«Игра по угадыванию семантики»* $\mathrm{DSemGame}$*) Опишем алгоритм* $\mathrm{DSemGame}^{\mathrm{Adv}}(\lambda)$*, где* λ *– параметр уровня криптостойкости, а* Adv *– криптоаналитик.*

1. *Положить* $\mathbf{k} \leftarrow \mathrm{KeyGen}(1^{\lambda})$
2. *Генерировать* $(m_0, m_1) \leftarrow \mathrm{Adv}^{Enc_{\mathbf{k}}(\cdot)}(1^{\lambda})$ *при* $|m_0| = |m_1|$.
3. *Положить* $b \leftarrow \{0,1\}$ *и шифртекст* $c \leftarrow Enc_{\mathbf{k}}(m_b)$.
4. *Пусть* $b' \leftarrow \mathrm{Adv}^{Enc_{\mathbf{k}}(\cdot)}(c)$.
5. *Выдать 1, если* $b = b'$ *и* Adv *никогда не запрашивал* m_0 *или* m_1 *у оракула зашифрования, иначе выдать 0.*

Определение 2.2. *Криптосистема называется семантически криптостойкой, если для любого полиномиального алгоритма* Adv *при любом выборе достаточно большого параметра уровня криптостойкости* λ *выполняется*

$$\Pr\left[\mathrm{DSemGame}^{\mathrm{Adv}}(1^{\lambda}) = 1\right] \le 1/2 + negl(\lambda),$$

где вероятность случайных величин берется равномерной.

Однако некоторые авторы доказывают криптостойкость в атаке только по шифртекстам, поскольку их криптосистемы не обеспечивают стойкость в атаке с известным открытым текстом, хотя являются достаточно простыми и вычислительно эффективными, что обуславливает желание найти им какое-то применение. Дело в том, что хотя свойство податливости имеет место для всех алгебраически гомоморфных криптосистем, но фактическую опасность представляет в основном для криптосистем с небольшим пространством открытых текстов: например, если это пространство – один бит (как в исходной криптосистеме Джентри), можно произвести следующую операцию – возвести шифртекст в квадрат и прибавить полученное к исходному шифртексту, таким образом, получив шифртекст нуля. В случае же большого пространства открытых текстов (тем более, если это пространство не имеет алгебраической структуры поля) неочевидно как можно воспользоваться податливостью для взлома шифра.

В целом, можно сказать, что есть потребность в критериях, позволяющих отделить «пригодные к использованию» ПГШ криптосистемы от «непригодных» в ряде практических ситуаций. Например, можно ли использовать ПГШ в случае если, например, криптоаналитик без труда может получить некоторые соотношения между открытыми текстами, соответствующими имеющимся у него шифртекстам? В случае, если криптоаналитику известно вероятностное распределение на множестве открытых текстов? Или в ситуации (часто встречающейся в приложениях облачных баз данных), когда один и тот же набор исходных данных оказывается зашифрованным в нескольких экземплярах с использованием разных криптосистем (т.е. данные дублируются но с использованием разного шифрования – это делается для обеспечения всех необходимых операций над данными, поскольку один вид ШСС не сохраняет все необходимые свойства)?

Зададимся вопросом: *какими свойствами криптостойкости должна обладать гомоморфная криптосистема для безопасного использования в этих условиях*? Конечно, в случае использования шифра, криптостойкого против атаки по известным открытым текстам можно не беспокоится о нарушении защиты. Проблема, однако, состоит в том, что полностью гомоморфные криптосистемы с доказанными свойствами криптостойкости против атаки по известным открытым текстам пока что являются слишком вычислительно неэффективными. Вместе с тем есть вычислительно эффективные криптосистемы с доказанными свойствами стойкости против атаки только по шифртекстам, поэтому возникает резонный вопрос о возможности применения их в данной ситуации.

Будем исходить из следующего предположения: *пригодное к использованию полностью гомоморфное шифрование не может быть детерминированным*. Во-первых, оно не обеспечивает семантическую криптостойкость: если существует единственный шифртекст, соответствующий данному открытому тексту, то в игре DSemGame противник может отличить зашифрованное m_0

от m_1 , запуская алгоритм Enc и сравнивая результат со своим шифртекстом. Во-вторых, нейтральный по операции элемент в большинстве криптосистем определен единственным образом, и т.о. фактически не может быть зашифрован при таком подходе.

Исходя из сказанного, «плохой» будем считать детерминированную гомоморфную криптосистему, или *сводящуюся к таковой*. О чем идет речь? Дело в том, что вероятностную криптосистему можно преобразовать в детерминированную путем преобразования её алгоритма Enc: зафиксируем некоторый набор случайных битов, подаваемых на вход этому алгоритму и будем производить шифртексты только с этим набором. Поскольку этот набор произволен, то одной вероятностной криптосистеме можно поставить в соответствие 2^{λ} (где λ – количество случайных битов, подаваемых на вход исходному алгоритму Enc) детерминированных криптосистем, каждая из которых получается при некотором зафиксировании вектора случайных битов. Будем говорить о вышеописанном процессе как о *дерандомизации* вероятностной криптосистемы.

Пусть $\varepsilon = \{\mathrm{KeyGen}, \mathrm{Enc}, \mathrm{Dec}\}$ – вероятностная криптосистема, $c_1,...,c_N$ – шифртексты открытых текстов $m_1,...,m_N$, произведенные алгоритмом Enc криптосистемы ε, т.е. $c_i = \mathrm{Enc}(m_i, \mathbf{k}, r_i), i = 1,...,N$, где $\mathbf{k}$ – ключ зашифрования, r_i – случайные элементы (последовательности случайных битов). Рассмотрим детерминированную криптосистему $\varepsilon' = \{\mathrm{KeyGen}, \mathrm{Enc}', \mathrm{Dec}\}$, полученную в результате дерандомизации криптосистемы ε и обозначим как $c_1',...,c_N'$ шифртексты, производимые её алгоритмом Enc' от тех же открытых текстов $m_1,...,m_N$, т.е. $c_i' = \mathrm{Enc}'(m_i, \mathbf{k}), i = 1,...,N$. Если существует полиномиальный алгоритм, который преобразует каждое c_i в c_i' то будем говорить, что криптосистема является нестойкой к дерандомизации.

По аналогии можно ввести понятие «частичной дернадомизации» когда существует полиномиальный алгоритм преобразования шифртекстов в такие, которые были порождены с использованием *меньшего* набора случайных битов.

Конечно, сама по себе дерандомизация – это еще не полный взлом шифра, однако это показатель того, что его криптостойкость не так высока, как кажется на первый взгляд. Можно ли использовать дерандомизацию как элемент процесса полного взлома шифра? Как конкретно может быть проведена дерандомизация?

В следующем разделе попытаемся предложить модель криптостойкости, в рамках которой можно будет ответить на эти вопросы.

3. Понятие дерандомизационной криптостойкости

В литературе [8] можно встретить понятие дерандомизации алгоритмов, которое означает *эффективное* преобразование вероятностного алгоритма в детерминированный. В данной работе вводится понятие дерандомизации вероятностной криптосистемы или шифра (хотя неформально оно употреблялось, например, в [14]), которое аналогично дерандомизации алгоритмов означает *эффективное* преобразование вероятностного шифра (криптосистемы) в шифр простой замены (детерминированный).

Данное понятие имеет особое значение для полностью гомоморфных криптосистем: пригодная к использованию криптосистема такого типа *должна быть* вероятностной, поскольку в большинстве случаев нейтральные по гомоморфным операциям элементы определены единственным образом, как следствие в случае детерминированной гомоморфной криптосистем даже незначительной дополнительной информации (например, знания о том, что вероятностное распределение на множестве открытых текстов неравномерно) может оказаться достаточно для вскрытия шифрования.

Приведем пример практической ситуации, в которой возникает необходимость подобного анализа защищенности. Допустим, случается частичная утечка информации, например, нескольких бит открытого текста. В случае если уже имеются какие-то соотношения между открытыми текстами, с учетом дополнительной информации шифр может быть полностью взломан даже в случае его стойкости против атаки только по шифртекстам.

В интересующих нас случаях для раскрытия исходных данных вовсе не требуется полной определенности в соотношениях между открытыми текстами, а требуется лишь *некоторая* информация. В соответствии с этим попытаемся расширить наше представление о дерандомизации, для того чтобы это учесть.

Таким образом, приходим к следующему: дерандомизация подразумевает получение некоторой системы уравнений с участием только открытых текстов, однако эти уравнения могут иметь разное количество решений и разную степень сложности решения. Интуитивно, можно сказать, что, например, получение линейных или аффинных соотношений почти полностью разрушает конфиденциальность. Необходимо также различать случаи, когда система имеет одно решение, но, например, сложна для решения от случая, когда она имеет одно решение и легко решается.

Определение 3.1: *Систему из (более чем одного) полиномиальных уравнений*

$$\begin{cases} p_1(x_1,...,x_n,\alpha)=0 \\ ... \\ p_k(x_1,...,x_n,\alpha)=0 \end{cases} \quad (1)$$

от переменных $x_1,...,x_n,\alpha$ *назовем однопараметрической системой уравнений, если при каждом значении параметра (выбранного из переменных)* α *система* (1) *имеет единственное решение*.

Определение 3.2. *Криптосистема называется криптостойкой к дерандомизации, если не существует полиномиального алгоритма построения однопараметрической системы уравнений с участием только открытых текстов, соответствующих данным шифртекстам*.

Очевидно, что для шифра простой замены можно составить однопараметрическую систему уравнений, связывающую открытые тексты имеющихся у криптоаналитика шифртекстов, и других переменных кроме открытых текстов в системе уравнений нет. Более того, при фиксации любого открытого текста все остальные открытые тексты из системы определяются однозначно, таким образом можно любой шифртекст назначить параметром.

Определение 3.3. *Криптосистема называется криптостойкой к полной дерандомизации если не существует полиномиального алгоритма построения однозначно разрешимой системы уравнений с участием только открытых текстов, соответствующих данным шифртекстам*.

Основная цель при построении полностью гомоморфных криптосистем состоит в том, чтобы сделать их вероятностными, поскольку, к примеру, в случае шифра простой замены если для операций выполняются свойства кольца, то нейтральные элементы по умножению (единица) и сложению (ноль) определяются единственным образом и, таким образом при любом секретном ключе при зашифровании переходят сами в себя.

Определение 3.4. *Криптосистема называется криптостойкой к обобщенной дерандомизации, если не существует полиномиального алгоритма построения какой-либо нетривиальной системы уравнений с участием только открытых текстов, соответствующих данным шифртекстам*.

Естественно задаться вопросом: в каких соотношениях состоят введенные данными определениями понятия по отношению к традиционным в криптографии, таким как неразличимость шифртекстов [15], семантическая крипто-

стойкость и т. д.? Классически, от шифрования[2] требуется семантическая криптостойкость против атаки по известным открытым текстам.

Утверждение 1. *Свойство неразличимости шифртекстов криптосистемы (стойкость к атаке по известным открытым текстам) влечет криптостойкость к полной дерандомизации*.

Доказательство. Предположим, криптосистема нестойка к полной дерандомизации. Тогда криптоаналитик составляет систему уравнений, связывающих открытые тексты, соответствующие шифртекстам, а затем запускает игру $\mathrm{DSemGame}$, в которой по очереди подставляет открытые тексты в систему (при этом в одном из двух случаев система будет разрешима) и тем самым находит, какой из шифртекстов что шифрует.

Утверждение 2. *Свойство неразличимости шифртекстов криптосистемы (стойкость к атаке по известным открытым текстам) влечет криптостойкость к дерандомизации*.

Доказательство. Предположим, криптосистема нестойка к дерандомизации. Тогда криптоаналитик составляет систему уравнений, связывающих открытые тексты, соответствующие шифртекстам, а затем запускает игру $\mathrm{DSemGame}$, в которой по очереди подставляет открытые тексты в систему и находит НОД получившихся после подстановки полиномов от одной переменной α (при этом в одном из двух случаев НОД будет нетривиален, т.е.>1) и тем самым находит, какой из шифртекстов что шифрует.

Дерандомизации зачастую позволяет ответить на такие вопросы: есть ли в данной последовательности шифртекстов такие, которые шифруют один и тот же открытый текст и много ли таких шифртекстов?

Можно предложить некоторое альтернативное определение дерандомизации. Представим себе алгоритм шифрования гомоморфной криптосистемы: *в зависимости от параметра уровня криптостойкости этот алгоритм использует разное количество случайных битов при производстве шифртекстов*. Если часть из этих битов стала известна криптоаналитику, то такую ситуацию называют *утечкой битов* или *частичной утечкой информации*. Ситуация же дерандомизации с этой точки зрения представляет собой некоторый массовый

[2] Как общепринято в литературе по криптографии, считаем, что криптосистема (шифр) состоит из тройки алгоритмов генерации ключа $\mathrm{KeyGen}(1^{\lambda})$, зашифрования $\mathrm{Enc}_{\mathbf{k}}(\cdot)$ и расшифрования $\mathrm{Dec}_{\mathbf{k}}(\cdot)$.

вариант утечки битов: *для всех шифртекстов имеющихся у криптоаналитика ему открывается одинаковое число случайных битов*. Таким образом, можно предложить альтернативное определение дерандомизации. Для более формального определения вспомним, что алгоритм зашифрования $Enc_{\mathbf{k}}(\cdot)$ *вероятностной* криптосистемы использует для производства шифртекста кроме секретного ключа еще и набор из λ случайных битов.

Определение 3.2'. *Криптосистема является стойкой к дерандомизации, если для преобразования шифртекстов, произведённых алгоритмом* $Enc_{\mathbf{k}}(\cdot)$ *с использованием* λ *случайных битов, то преобразование этих шифртекстов к таким, которые произведены алгоритмом* $Enc_{\mathbf{k}}(\cdot)$ *с использованием* $\lambda' = 0$ *случайных битов, необходимо решить вычислительно сложную задачу*.

Справедлива следующая теорема.

Теорема 1. Определения 3.2 и 3.2' эквивалентны.
Доказательство будет изложено в расширенной версии статьи.

Определение 3.4'. *Криптосистема является стойкой к обобщённой дерандомизации, если для преобразования шифртекстов, произведённых алгоритмом* $Enc_{\mathbf{sk}}(\cdot)$ *с использованием* λ *случайных битов, то преобразование этих шифртекстов к таким, которые произведены алгоритмом* $Enc_{\mathbf{k}}(\cdot)$ *с использованием* $\lambda' < \lambda$ *случайных битов, необходимо решить NP-сложную вычислительную задачу.*

Теорема 2. Определения 3.4 и 3.4' эквивалентны.
Доказательство будет изложено в расширенной версии статьи.

Рассмотрим теперь примеры стойких и нестойких к дерандомизации криптосистем. Рассмотрим криптосистемы из [9].

Утверждение 3. Криптосистемы MORE и PORE являются стойкими к дерандомизации.

Утверждение 4. Криптосистема из [11] является нестойкой к дерандомизации.

Поскольку в качестве преобразования зашифрования в [11] предлагается использовать такой гомоморфизм полиномиальных колец как подстановку (композицию) полиномов, а вместе с тем известно, что для декомпозиции полиномов (с точностью до линейного члена) существуют полиномиальные алгоритмы, то можно сделать вывод о нестойкости к дерандомизации этой криптосистемы.

Замечание. На первый взгляд может показаться, что нестойкость к дерандомизации полностью гомоморфных криптосистем влечет и их нестойкость к атаке по шифртекстам (в случае нестойкости к атаке с известным открытым текстом), ввиду свойства податливости. Однако это неверно, поскольку получаемые посредством податливости шифртексты, про которые мы знаем, какому исходному тексту они соответствуют, могут оказаться «универсальными», т.е. подходящими для всех секретных ключей и таким образом не хранящими информации о секретном ключе. Таковы, например, в случае криптосистемы MORE из [9] диагональные матрицы или в случае PORE просто числа (любое число для PORE является шифртекстом самого себя на любом ключе).

4. Заключение

Были определены минимальные требования, предъявляемые к используемым на практике гомоморфным шифрам. Были введены новое понятие дерандомизации шифров и определения криптостойкости к дерандомизации. Установлены взаимосвязи введенных определений с классическими определениями стойкости в криптографии при различных атаках в общем случае и в случае полностью гомоморфных криптосистем.

Показаны примеры алгебраически гомоморфных криптосистем, являющиеся стойкими и нестойкими к дерандомизации.

Список литературы

[1]. Popa, R. A., Redfield, C., Zeldovich, N., Balakrishnan, H. CryptDB: protecting confidentiality with encrypted query processing //Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. – ACM, 2011. – С. 85-100.

[2]. A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In CIDR, 2013.

[3]. Google Encrytped Big Query. https://github.com/google/encrypted-bigquery-client.

[4]. Always Encrypted. https://msdn.microsoft.com/en-us/library/mt163865(v=sql.130).aspx.

[5]. Shatilov, K., Boiko, V., Krendelev, S., Anisutina, D., & Sumaneev, A. Solution for secure private data storage in a cloud //Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on. – IEEE, 2014. – С. 885-889.

[6]. Agrawal, R., Kiernan, J., Srikant, R., & Xu, Y. Order preserving encryption for numeric data //Proceedings of the 2004 ACM SIGMOD international conference on Management of data. – ACM, 2004. – С. 563-574.

[7]. Н. П. Варновский, А. В. Шокуров. Гомоморфное шифрование. Труды ИСП РАН, том 12, 2007 г. стр. 27-36

[8]. Barak B., Ong S. J., Vadhan S. Derandomization in cryptography //SIAM Journal on Computing. – 2007. – Т. 37. – №. 2. – С. 380-400.
[9]. Kipnis A., Hibshoosh E. Efficient Methods for Practical Fully Homomorphic Symmetric-key Encrypton, Randomization and Verification //IACR Cryptology ePrint Archive. – 2012. – №. 637.
[10]. Kipnis A., Hibshoosh E. Method and system for homomorphicly randomizing an input : заяв. пат. 14/417,184 США. – 2013. ().
[11]. Жиров А.О., Жирова О.В., Кренделев С.Ф.. Безопасные облачные вычисления с помощью гомоморфной криптографии. Безопасность информационных технологий, 2013, Т. 1, С. 6–12.
[12]. R. L. Rivest, L. Adleman, M. L. Dertouzos. On data banks and privacy homomorphisms //Foundations of secure computation. –Т. 4. – №. 11. – 1978. pp. 169-180.
[13]. C. Gentry. Fully homomorphic encryption using ideal lattices //Proceedings of the 41st annual ACM symposium on Symposium on theory of computing-STOC'09. – ACM Press, 2009. pp. 169-169.
[14]. Hemenway B., Ostrovsky R. Building lossy trapdoor functions from lossy encryption //Advances in Cryptology-ASIACRYPT 2013. – Springer Berlin Heidelberg, 2013. – С. 241-260.
[15]. Goldreich O. Foundations of cryptography: volume 2, basic applications. – Cambridge university press, 2004.

Derandomization Security of Homomorphic Encryption

A. Trepacheva <alina1989malina@ya.ru>
Southern Federal University,
105/42, Bolshaya Sadovaya st., Rostov-on-Don, 344006, Russia

Abstract. The paper considers the problems of developing and analysis of cloud database systems. We determine the minimal requirements for encryption to be usable in practical applications. A new notion – a non-derandomizable encryption – allows to do this and we explain the practical value of this notion as well as links between it and classical notions of cryptosystem's security, practical security of whole cloud computing system.
The derandomizable encryption essentially is equivalent to a simple substitution cipher. In other words, encryption is derandomizable if an effective algorithm exists translating it into a simple substitution cipher.
There are some features of derandomizable encryption allowing to check their properties in a simple way. For this purpose, this paper proposes an alternative definition of derandomizable encryption in terms of systems of equations, drawn up by known plaintext cryptanalysis.
Then the paper proposes definitions of generalized derandomization and full derandomization.
Briefly, the generalized derandomizable encryption allows to reduce efficiently the number of variables in system of equations composed for known plaintext attack; the fully derandomizable encryption allows to compose uniquely solvable system of equations by known plaintext attack effectively.

We show the examples of simple algebraically homomorphic cryptosystems – both derandomizable and not non-derandomizable. The paper finally concludes about usability of considered cryptosystems for practical cloud systems.

Keywords: information security, cloud computing, homomorphic encryption, secure computations, derandomization.

DOI: 10.15514/ISPRAS-2015-27(6)-24

For citation: Trepacheva A. Derandomization Security of Homomorphic Encryption. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 381-394 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-24.

References

[1]. Popa, R. A., Redfield, C., Zeldovich, N., & Balakrishnan, H. CryptDB: protecting confidentiality with encrypted query processing //Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. – ACM, 2011. – С. 85-100.

[2]. A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In CIDR, 2013.

[3]. Google Encrytped Big Query. https://github.com/google/encrypted-bigquery-client.

[4]. Always Encrypted. https://msdn.microsoft.com/en-us/library/mt163865(v=sql.130).aspx.

[5]. Shatilov, K., Boiko, V., Krendelev, S., Anisutina, D., & Sumaneev, A. Solution for secure private data storage in a cloud //Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on. – IEEE, 2014. – С. 885-889.

[6]. Agrawal, R., Kiernan, J., Srikant, R., & Xu, Y. Order preserving encryption for numeric data //Proceedings of the 2004 ACM SIGMOD international conference on Management of data. – ACM, 2004. – С. 563-574.

[7]. N. P. Varnovskij, A. V. Shokurov. Gomomorfnoe shifrovanie [Homomorphic Encryption]. *Trudy ISP RAN* [*The Proceedings of ISP RAS*], 2007, vol. 12, pp. 27-36 (in Russian).

[8]. Barak B., Ong S. J., Vadhan S. Derandomization in cryptography //SIAM Journal on Computing. – 2007. – Т. 37. – №. 2. – С. 380-400.

[9]. Kipnis A., Hibshoosh E. Efficient Methods for Practical Fully Homomorphic Symmetric-key Encrypton, Randomization and Verification //IACR Cryptology ePrint Archive. – 2012. – №. 637.

[10]. Kipnis A., Hibshoosh E. Method and system for homomorphicly randomizing an input : заяв. пат. 14/417,184 США. – 2013. ().

[11]. Zjirov A.O., Zjirova O.V., Krendelev S.Ph. Bezopasnye oblachnye vychislenija s pomoshh'ju gomomorfnoj kriptografii [Secure cloud computing with homomorphic encryption]. Bezopasnost' informacionnyh tehnologij [Security of Information Technologies], 2013, v. 1, pp. 6–12 (in Russian).

[12]. R. L. Rivest, L. Adleman, M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 1978, vol. 4, no. 11. pp. 169-180.

[13]. C. Gentry. Fully homomorphic encryption using ideal lattices. *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing-STOC'09*. – ACM Press, 2009, pp. 169-169.

[14]. Hemenway B., Ostrovsky R. Building lossy trapdoor functions from lossy encryption //Advances in Cryptology-ASIACRYPT 2013. – Springer Berlin Heidelberg, 2013. – C. 241-260.

[15]. Goldreich O. Foundations of cryptography: volume 2, basic applications. – Cambridge university press, 2004.

Автоматизированное оперативное управление техногенными химико-технологическими объектами при возникновении запроектных аварийных ситуаций[1]

Ю.Н. Матвеев <matveev4700@mail.ru>
Н.А. Стукалова <nast77@mail.ru>
Тверской государственный технический университет,
170026, Россия, наб. А. Никитина, 22

Аннотация. В статье обосновывается необходимость применения распределенных вычислительных систем для математического моделирования процесса образования облака зараженного воздуха при возникновении запроектных аварийных ситуаций на техногенных химико-технологических объектах. Проанализирована зависимость скорости изменения концентрации опасной примеси в произвольной точке пространства и приведены математические модели процессов образования облака зараженного воздуха при высокотемпературных выбросах (взрывах, пожарах) и проливах больших количеств токсичных химических веществ на различные поверхности с последующим их испарением. Доказана целесообразность декомпозиции задачи моделирования процесса образования облака зараженного воздуха.

Ключевые слова: аварийная ситуация; химико-технологический объект; облако зараженного воздуха; математическая модель; распределенные вычислительные системы.

DOI: 10.15514/ISPRAS-2015-27(6)-25

Для цитирования: Матвеев Ю.Н., Стукалова Н.А. Автоматизированное оперативное управление техногенными химико-технологическими объектами при возникновении запроектных аварийных ситуаций. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 395-408. DOI: 10.15514/ISPRAS-2015-27(6)-25.

[1] Работа выполнена при финансовой поддержке РФФИ грант №15-29-07970

1.Введение

Автоматизация работы объектов повышенной опасности, в частности, опасных химических производственных комплексов и технологических процессов, является сегодня неотъемлемой частью повышения их безопасности и снижения вероятности возникновения аварийных ситуаций. Цена аварии на подобных объектах, как правило, имеет катастрофический масштаб, поскольку последствия возникновения чрезвычайных ситуаций затрагивают не только и не столько сам объект, но и сопряженные с ним территории. В общем случае, авария для техногенных объектов определяется как несанкционированное высвобождение массы или энергии, которое причиняет или способно причинить ущерб реципиенту риска. При этом масса или энергия этого высвобождающегося опасного вещества образует источник аварийной опасности. Исследование особенностей вредоносных факторов и разработка эффективных мер по их ослаблению или ликвидации возможно только на основе использования математического моделирования таких объектов. Это объясняется тем, что аварийную ситуацию невозможно организовать с благими намерениями или повторить. Математическая модель аварийной ситуации должна адекватно описывать возникновение и развитие источника опасности. Исследование модели позволяет определить условия, при которых происходит эмиссия опасной субстанции, и рассчитать параметры процесса эмиссии.

2. Классификация аварийных ситуаций

Сценарии возникновения и развития аварийных ситуаций представляют собой последовательность возможных характерных событий, ведущих к утечке токсичных, пожаро- и взрывоопасных веществ. Среди них можно выделить две основных группы инициаторов аварийных ситуаций – внутренние события и внешние. К внутренним относятся такие события, источником которых служат технологические операции и технологическое оборудование, например, скрытые дефекты оборудования, разрыв трубопроводов, разрушение емкостей с химически опасными веществами, ошибки операторов-технологов и т.д. К внешним относятся события, первопричина которых находятся вне самого химико-технологического процесса, например, падение самолёта на объект при авиационной катастрофе.

Кроме этого, весь спектр возможных аварий можно разделить на две группы, которые принято называть «проектными» и «запроектными».

К «проектным авариям» относятся такие ситуации, которые в случае их возникновения не приводят к аварии т.к. заранее запланированы дополнительные организационные и технические мероприятия по их нейтрализации. Как правило, это аварийные ситуации, причиной которых являются различного рода отказы оборудования. Для снижения вероятности и возможных масштабов таких аварий в типовую проектную и технологическую

документацию вносят определённые дополнения, предусматривается установка различного рода блокирующих, сигнальных и других систем.

К «запроектным» относятся аварии, не вошедшие в первую группу. Причинами таких аварий служат в основном внешние непредсказуемые заранее события – различного рода стихийные бедствия (землетрясение, тайфуны, ураганы и т.д.) или непрогнозируемые последствия человеческой деятельности (взрывы, крупные пожары и пр.). Возможность снижения вероятности их возникновения учитывается в основном за счёт таких организационных мероприятий, как выбор площадки для размещения объекта, например, в сейсмически неактивной зоне. В проектной и технологической документации возможность возникновения таких аварий не находит своего отражения, поэтому такие аварии принято называть «запроектными».

Вероятность возникновения проектных аварий гораздо выше, чем аварий запроектных. Однако масштабы запроектных аварий в случае их возникновения гораздо больше, чем масштабы проектных, так как при малой частоте их появления они обладают гораздо большей разрушительной силой. Причём, по упомянутым выше причинам, на техногенных объектах не предусматривается введение элементов и систем, предназначенных для противостояния или ограничения этой разрушительной силы.

Анализ возможных сценариев аварийных ситуаций на техногенных объектах химических производств, приводящих к выбросам токсичных химических веществ (ТХВ) в атмосферу с образованием облака зараженного воздуха (ОЗВ), показывает, что основными вариантами сценариев таких аварий являются[1]:

- высокотемпературные выбросы ТХВ, которые по времени протекания могут быть кратковременными и продолжительными (взрывы, пожары);
- пролив больших количеств ТХВ на различные поверхности с последующим их испарением.

3. Процессы образования ОЗВ как объекты управления.

Зависимость скорости изменения концентрации примеси в произвольной точке пространства – $\partial Ca/\partial t$ определяется расположением в пространстве источников примеси и рядом параметров, в число которых входят составляющие скорости ветра – u, v, w вдоль осей Ox, Oy, Oz трехмерного пространства, коэффициенты атмосферной турбулентности – k и др. [2]

В общем виде эта зависимость определяется дифференциальным уравнением баланса примеси или уравнением переноса примеси:

$$\frac{\partial Ca}{\partial t} = -u\frac{\partial Ca}{\partial x} - v\frac{\partial Ca}{\partial y} - w\frac{\partial Ca}{\partial z} + \frac{\partial}{\partial x}(k\frac{\partial Ca}{\partial x}) + \frac{\partial}{\partial y}(k\frac{\partial Ca}{\partial y}) + \\ + \frac{\partial}{\partial z}(k\frac{\partial Ca}{\partial z} + w_a Ca) + F + R - P - W = f(u, v, w, k, w_a, F, R, P, W) \quad (1)$$

где:

w_a – собственная вертикальная скорость примеси,

R и P – скорость образования и уничтожения примеси в результате химических реакций,

W – скорость выпадения примеси на подстилающую поверхность;

Ca -количество примеси a, содержащейся в единице объема воздуха (объемная концентрация примеси);

F – скорость поступления примеси в воздух от того или иного источника (интенсивность источника).

Уравнение (1) показывает, что перенос примеси в движущейся среде обусловлен двумя различными физическими факторами: во-первых, при наличии разности концентраций идет процесс молекулярной диффузии; во-вторых, частицы примеси увлекаются движущейся средой и переносятся вместе с ней.

Кроме параметров, входящих в уравнение (1) должны быть заданы условия, ограничивающие область распространения ОЗВ. Если область распространения ОЗВ ограничена сверху поверхностью z=H, а снизу земной поверхностью z=0, то условия на этих границах обычно задаются в виде:

при z=H Ca=0 или $k\frac{\partial Ca}{\partial z}$ =0, (2)

при z=0 $k\frac{\partial Ca}{\partial z}$ + w_aCa=βCa . (3)

При рассмотрении примеси в слое толщиной H порядка 2-5 км условия (2) определяют исчезновение примеси или ее вертикального потока на этой границе. Условие (3) на земной поверхности имеет смысл равенства вертикального турбулентного потока и потока примеси при ее гравитационном оседании на поверхность со скоростью w_a количеству примеси, поглощаемой поверхностью - βCa, где β - коэффициент аккомодации, зависящий от физических свойств подстилающей поверхности, наличия на ней растительности или застроек.

Тогда задачу оптимизации управления процессом образования ОЗВ в общем виде можно записать:

$$V_0 = F\left(\overline{X}, \overline{U}, Ca, t\right) \to \min \qquad (4)$$

т. е. минимизировать объем ОЗВ при связях и ограничениях:

$$\partial Ca/\partial t = f(u,v,w,k,w_a,F,R,P,W) = f(\overline{X}, \overline{U}), \qquad (5)$$

$$Ca(x,y,z,t;h_0) \le \text{ПДК}, \qquad (6)$$

$$x_{min} \le x \le x_{max}, \qquad (7)$$

$$y_{min} \le y \le y_{max}, \qquad (8)$$

$$0 \le z \le z_{ф}, \qquad (9)$$

$$0 \le t \le t_{дир}, \qquad (10)$$

где

$Ca(x,y,z,t;h_0)$ – концентрация ТВХ в точке с координатами x,y,z в момент времени t от источника, расположенного на высоте h_0;

$\overline{X}, \overline{U}$ – векторы входных и управляющих параметров;

$z_{ф}$ – высота флюгера;

$t_{дир}$ – директивное время локализации аварии.

4. Математические модели процессов образования облака зараженного воздуха

Как уже было отмечено, процессы образования ОЗВ – это процессы перехода ТХВ из начального источника химического заражения в атмосферу в результате взрывного разрушения аппаратов, пожаров и проливов на различные поверхности с последующим испарением.

При взрыве процессом поступления ТХВ в атмосферу будет испарение жидких частиц и капель с поверхности образовавшегося пароаэрозольного облака. Поступление ТХВ в атмосферу определяется производительностью источника [3]:

$$q = \frac{\overline{\Delta_0}}{m_0} E_0 \exp\left(-\frac{\alpha_k E_0 t}{m_0}\right), \qquad (11)$$

где:

q – производительность источника, *кг/с*;

$\overline{\Delta_0}$ – начальная плотность заражения;

m_0 – масса частицы среднемассовым диаметром d_k, *кг;*

E_0 – начальная скорость испарения, *кг*/(*м*^{2}·с):

$$E_0 = K_p 2\pi\left(1+1{,}2^{-2}\,\mathrm{Re}^{2/3}\right)C_0 D d_n \sqrt{\upsilon/D}\,, \qquad (12)$$

где:

K_p – коэффициент, зависящий от типа ТХВ и вида почвы (K_p=0,9 для большинства ТХВ);

Re – число Рейнольдса:

$$\mathrm{Re} = \frac{0{,}25\,u_1 d_n}{va}\,; \qquad (13)$$

C_0 – максимальная концентрация насыщенного пара, % ;

D – коэффициент диффузии вещества;

d_n – диаметр пятна от частицы вещества, *м* :

$$d_n \approx 3\overline{d_k} \qquad (14)$$

υ – коэффициент кинематической вязкости воздуха;

a_k – параметр, зависящий от доли свободной или связанной жидкости;

t – время, прошедшее от начала заражения, с.

При проливе процессом поступления ТХВ в атмосферу будет только испарение с поверхности образовавшейся «лужи»:

$$q = S_{np} E \qquad (15)$$

где:

q – производительность источника, *кг*/с;

E – скорость испарения, *кг*/(*м* ^{2}·с);

S_{np} – площадь зеркала пролива, *м*^{2}.

Испарение с гладкой поверхности пролитого вещества рассчитывается по методу Братсерта [4].

Скорость стационарного испарения жидкого ТХВ со свободной поверхности пролива, когда необходимо оценивать общую массу испарившегося вещества, можно определить из выражения [5]:

$$E = 0{,}0735 \cdot C_m \cdot U_* \cdot \left(\frac{D_M}{\upsilon_B} \right)^{2/3}, \ \frac{кг}{м^2 \cdot с} \qquad (16)$$

где:

C_m – максимальная концентрация паров ТХВ при данной температуре испарения, *кг*/*м* ^{3};

U_* – динамическая скорость воздуха над поверхностью испарения, *м*/с;

D_M – коэффициент молекулярной диффузии паров ТХВ в воздухе, $м^2$/с;

υ_B – кинематическая вязкость воздуха, $м^2$ /с.

Основное уравнение для расчета C_m применительно к ТХВ имеет вид:

$$C_m = 16 \cdot 10^9 \frac{M_{OB} \cdot P_m}{T}, \ кг/м^3 \qquad (17)$$

где:

$M_{ов}$ – молекулярная масса ТХВ, *кг*/моль;

P_m – давление насыщенного пара, мм рт.ст. при температуре Т.

Давление насыщенного пара ТХВ при температурах, характерных для процессов испарения ТХВ при авариях, может быть рассчитано по уравнению

$$Lq_{10}(P_m) = A - \frac{B}{T}, \qquad (18)$$

где:

А, В – коэффициенты уравнения Антуана, рассчитываемые по экспериментальным данным для каждого типа ОВ в определенном интервале температур;

Т – температура жидкого ТХВ, *град.К*.

Динамическая скорость воздуха у поверхности земли может быть рассчитана по формулам:

$$u_*/u_1 = \begin{cases} \left(0,55 - 0,52B^{0,38535}\right)\exp\left(0,52 + 0,283B^{0,269}\right)\ln Z_0, B > 0; \\ 0,55 \cdot Z_0^{0,52}, B = 0 \\ \left[\left(0,55 + 0,52(-B)^{0,38535}\right)\right]\exp\left[\left(0,52 + 0,0313(-B)^{0,155}\right)\right]\ln Z_0, B < 0, \end{cases} \quad (19)$$

где:

B – значение параметра Бызовой (табл.1);

Z_0 – шероховатость подстилающей поверхности, *м* (табл.2)

u_1– скорость ветра на высоте 1*м* над слоем шероховатости, рассчитываемая по формуле:

$$u_1 = u_\phi\left[\left(1+Z_0\right)/Z_\phi\right]^m, \quad (20)$$

где:

Z_0 – высота флюгера, *м*;

u_ϕ – скорость ветра на высоте флюгера, *м*/с;

m – показатель профиля скорости ветра в приземном слое атмосферы, который рассчитывается по формулам:

$$m = \begin{cases} \dfrac{\left(z_0/z_\phi - 1\right)}{\ln\left(z_0/z_\phi\right)}, B = 0; \\ \dfrac{B\left(1-z_0/z_\phi\right)}{\left(1-z_0/z_\phi\right)^B}, B \neq 0. \end{cases} \quad (21)$$

Табл. 1 Значения коэффициента Бызовой

Характеристика устойчивости атомосферы	Очень неустойчивая	Умеренно неустойчивая	Слабо неустойчивая	Безразличная	Слабо неустойчивая	Умеренно неустойчивая	Очень неустойчивая

Класс устойчивости по Паскуилу -Тернеру	1	2	3	4	5	6	7
B	-0,1	-0,01	-0,001	0,0	0,001	0,01	0,05

Табл. 2 Шероховатость подстилающей поверхности Z_0 в зависимости от типа местности, где происходит рассеяние паров ТХВ

Тип поверхности	Z_0, м
Ровная местность, покрытая снегом	0,001
Ровная местность с высотой травы до 0,1	0,001
Ровная местность с высотой травы до 0,15 м	0,01
Ровная местность с высотой травы до 0,6 м	0,05
Неровная местность, покрытая кустарником	0,1
Лес высотой до 10 м	0,4
Городская застройка	1,0

Коэффициент молекулярной диффузии D_M может быть получен из уравнения Андрусова [4]:

$$D_{OM} = \frac{17{,}2\left(1+\sqrt{M_B+M_T}\right)}{Pa\left(3{,}08+V_K^{1/3}\right)^2\cdot\sqrt{M_B\cdot M_T}} \qquad (22)$$

где:

D_{OM} – коэффициент диффузии при 0 °С, *м* ^{2}/с;

M_B, M_T – молекулярная масса воздуха и ТХВ соответственно, *г*/моль;

P_a – давление, атм.

Для температур, отличающихся от 0^0С, используется уравнение вида [4]:

$$D_M(T) \approx D_{OM}\left(\frac{T}{273}\right)^{2,5}, \quad (23)$$

где T – температура.

Расчетное уравнение для оценки величины кинематической вязкости воздуха при атмосферном давлении имеет вид [4]:

$$\upsilon_B = 2{,}84 \cdot 10^{-6} \cdot T \cdot \exp\left(4{,}36 + 2{,}844 \cdot 10^{-3} \cdot T\right) \quad (24)$$

При пожаре образование ОЗВ будет осуществляется следующими процессами:

- выброс паров ТХВ в атмосферу в результате взрывного разрушения емкостей при термическом воздействии (через 20 минут после возникновения пожара) и конвективный подъем в очаге пожара;
- поступление паров ТХВ в атмосферу в результате конвективного подъема в очаге пожара и испарения с поверхности пролива при повышенной температуре.

Методика количественной оценки выброса паров ТХВ при взрывном разрушении емкости под воздействием тепловой нагрузки в [4] отсутствует. Поэтому можно ориентировочно принять массу ТХВ, поступившего в ОЗВ, равной 10 … 20% от общего количества ТХВ в емкости.

$$M_B = (0{,}1...0{,}2)\, M_E, \quad (25)$$

где:

M_B – масса паров ТХВ, поступивших в ОЗВ при термическом взрыве в момент времени $t_в$, *кг*;

M_E – масса ТХВ в емкости, *кг*.

Производительность источника поступления ТХВ в ОЗВ рассчитывается по формулам (15)-(22) .

Высота подъема конвективной колонки может быть определена из соотношения [6]:

$$h_k = \frac{K_r R_n}{U}\left[\frac{2{,}5 + 3{,}3 \cdot g \cdot R_n \cdot \left(T_n - T_B\right)}{T_B \cdot U^2}\right], \quad (26)$$

где:

h_k – высота источника относительно поверхности земли, *м*;

K_r – размерный коэффициент (равный 1,5 *м*/с);

R_n – радиус площади зоны пожара, *м*;

U – скорость ветра на высоте 10 метров, *м*/с;

g – ускорение силы тяжести;

T_n – температура горения в абсолютной шкале, ^{o}К;

T_B – температура окружающего воздуха в абсолютной шкале, ^{o}К.

Интенсивность выделения i-го продукта горения находится по формуле

$$Q_i = m_i \cdot \alpha_i \qquad (27)$$

где

Q_i – интенсивность выделения i-го продукта горения, *кг*/(*м* ^{2}·с);

m_i – массовая скорость выгорания, *кг*/(*м* ^{2}·с);

α_i – весовая доля i-го компонента продукта горения, выделяющегося при горении единицы массы горючей нагрузки (справочные данные).

Таким образом, при пожаре с разрушением емкостей общее количество ТХВ, поступающего в атмосферу, будет складываться из массы M_B мгновенного выброса и массы M_k конвективного подъема; при пожаре по всей площади разлива общее количество ТХВ, поступающего в атмосферу, будет складываться из массы M_k конвективного подъема и массы испаряющегося ТХВ.

На основании проведенного анализа процессов перехода ТХВ из начального источника химического заражения в ОЗВ можно сделать вывод, что переход ТХВ осуществляется в результате следующих процессов:

1. мгновенный выброс паров ТХВ в атмосферный воздух;
2. непрерывное поступление ТХВ в атмосферу – испарение из образовавшейся на поверхности (на неограниченной поверхности, в обваловке, в поддоне) «лужи» с ТХВ и испарение с поверхности выседания жидких частиц и капель;
3. поступление паров ТХВ в результате конвективного подъема из очага горения, а также взаимодействия указанных процессов.

Дифференциация основных вариантов процессов образования ОЗВ позволяет при идентификации процесса образования и распространения ОЗВ провести декомпозицию сложной модели на более простые, что упрощает разработку математического и программного обеспечения для решении задач оперативного управления в условиях чрезвычайной ситуации.

Для принятия решений в условиях чрезвычайных ситуаций необходимо исследовать сам объект как сложный динамический объект, его характеристики и свойства как объекта управления, процесс организации управления в условиях чрезвычайных ситуаций, а также разработать основы

создания систем информационной поддержки при принятии решений в условиях чрезвычайных ситуаций на основе математического моделирования.

5. Заключение

Рассмотренный в статье подход к моделированию процесса образования ОЗВ предполагает введение существенных допущений и ограничений. Например, величины компонент вектора скорости ветра по координатным осям OX,OY приняты постоянными и независящими от высоты.

Однако хорошо известно, что на разных высотах направление и скорость ветра значительно различаются. Это обстоятельство приводит к многократному повышению размерности задачи моделирования процесса образования ОЗВ, которая, к тому же, становится динамической.

Очевидно, что решение этой задачи потребует выделения больших объемов вычислительных ресурсов, которыми обладают распределенные вычислительные системы.

Список литературы

[1]. Методика выявления и оценки химической обстановки при разрушении (аварии) объектов, содержащих СДЯВ.-М: ГШВС.1989-116с.

[2]. Батырев В.В., Минько С.М. и др. Обоснование размеров зоны защитных мероприятий вокруг объектов по хранению и уничтожению химического оружия. Отчет по НИР «Вагонетка-О», этап 2.- Новогорск: АГЗ, 1998.-184с.

[3]. Белов П.Н. Теория расчета распространения атмосферных примесей и ее применение для оценки загрязнения природной среды // Географическое прогнозирование и охрана природы: Сб.науч.тр./ Под ред. Звонковой Т.В., Касимова Н.С. – М.:Изд-во МГУ, 1990. – 176с.

[4]. Братсерт У.Х. Испарение в атмосферу. Теория, история, приложения./ Пер. с англ. Под ред. А.С. Дубова. – Л.: ГИМЗ, 1985. – 350 с.

[5]. Методика прогнозирования развития и последствий аварийных ситуаций на объектах уничтожения химического оружия. Шифр «Система» (первая редакция). М.: ГУП ГосНИИОХТ, 2001. – 123 с.

[6]. Разработка и обоснование перечня мероприятий, осуществляемых при ликвидации последствий чрезвычайных ситуаций, применительно к возможным сценариям их развития. Промежуточный отчет по НИР «Конда», этап 2, № 715 – Редкино: ОАО «РОКБА», 2001 – 142 с.

Computer-Aided Operational Management Technogenic Chemical-Technological Objects at Occurrence of Beyond Design Basis Emergency Situations

Y. Matveev <matveev4700@mail.ru>
N. Stukalova <nast77@mail.ru>
Tver state technical University, 170026, Russia, Tver, A. Nikitin emb., 22

Abstract. Scales of consequences of accidents depend on the sizes of the cloud of the infected air and quality of resource management which arose when corruptings and spreaded in the atmosphere on mitigation of consequences. The purpose of operational management of resources on mitigation of consequences in alert conditions in case of release of ecologically dangerous substances are a minimization of spatial boundaries of manifestation of the striking action of a cloud of the infected air. In environmental monitoring of a status of dangerous chemical production the possibility of tracing in real time of consequences of burst of toxic agent is of great importance. Therefore an opportunity to trace a path of relocation of an aerosol cloud in real time is represented important. In article need of use of the distributed computing systems for mathematical process modeling of formation of a cloud of the infected air in case of origin of beyond design basis alert conditions on technogenic chemical and technological objects is justified. Classification of the considered alert conditions depending on the factors generating them is given. Processes of formation of a cloud of the infected air as control objects are considered. Dependence of speed of change of concentration of dangerous impurity in arbitrary point of space is analyzed and mathematical models of processes of formation of a cloud of the infected air in case of high-temperature bursts (explosions, the fires) and passages of large amounts of toxic chemicals on different surfaces with their subsequent evaporation are given. Feasibility of decomposition of the task of process modeling of formation of a cloud of the infected air is proved.

Keywords: emergency; chemical engineering; cloud contaminated air; mathematical model; distributed computing system.

DOI: 10.15514/ISPRAS-2015-27(6)-25

For citation: Matveev Y., Stukalova N. Computer-Aided Operational Management Technogenic Chemical-Technological Objects at Occurrence of Beyond Design Basis Emergency Situations. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 6, 2015, pp. 395-408 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-25

References

[1]. Metodika vyjavlenija i ocenki himicheskoj obstanovki pri razrushenii (avarii) obektov, soderzhashhih SDJaV [Methods of identification, M: General Staff of the Armed Forces.1989-116 p (in Russian)

[2]. Batyrev V.V., Minko S.M. i dr. Obosnovanie razmerov zony zashhitnyh meroprijatij vokrug obektov po hraneniju i unichtozheniju himicheskogo oruzhija [Rationale for the size of the zone of protective actions around objects on storage and destruction of chemical weapons]. Otchet po NIR «Vagonetka-O», jetap 2. [Scientific research reports "Trolley-On", step 2.] - Novogorsk: AGZ, 1998.-184 p. (in Russian)

[3]. Belov P.N. Teorija rascheta rasprostranenija atmosfernyh primesej i ee primenenie dlja ocenki zagrjaznenija prirodnoj sredy [The theory for calculating the distribution of atmospheric constituents and its application to evaluation of environmental pollution], Geograficheskoe prognozirovanie i ohrana prirody: Sb.nauch.tr. Pod red. Zvonkovoj T.V., Kasimova N.S. – M.:Izd-vo MGU[MSU Publishing house], 1990. – 176 p. (in Russian)

[4]. Bratsert U.H. Isparenie v atmosferu. Teorija, istorija, prilozhenija [Evaporation into the atmosphere. Theory, history, applications]. Per. s angl. Pod red. A.S. Dubova. – L.: GIMZ [GIS], 1985. – 350 p. (in Russian)

[5]. Metodika prognozirovanija razvitija i posledstvij avarijnyh situacij na ob#ektah unichtozhenija himicheskogo oruzhija. Shifr «Sistema» (pervaja redakcija) [Methods for prediction of development and consequences of emergency situations on objects of chemical weapons destruction. The code "System" (first edition)]. M.: GUP GosNIIOHT [sue GosNIIOKhT], 2001. – 123 p. (in Russian)

[6]. Razrabotka i obosnovanie perechnja meroprijatij, osushhestvljaemyh pri likvidacii posledstvij chrezvychajnyh situacij, primenitel'no k vozmozhnym scenarijam ih razvitija. Promezhutochnyj otchet po NIR «Konda», jetap 2 [Development and validation of the list of activities undertaken during the liquidation of consequences of emergency situations, with respect to possible scenarios of their development. Interim report on research "Condat", phase 2], № 715 – Redkino: OAO «ROKBA» [JSC "ROKBA"], 2001 – 142 p. (in Russian)

Облачный сервис для решения многомасштабных задач нанотехнологии на суперкомпьютерных системах

С.В. Поляков <polyakov@imamod.ru>
А.В. Выродов <vyrodov.alexey@gmail.com>
Д.В. Пузырьков <dpuzyrkov@gmail.com>
М.В. Якобовский <lira@imamod.ru>
Институт прикладной математики им.М.В.Келдыша РАН,
125047, Россия, г. Москва, Миусская пл. 4

Аннотация. В работе представлены структура и отдельные компоненты облачного сервиса, предназначенного для решения многомасштабных задач нанотехнологии на суперкомпьютерных системах. Мотивацией к созданию именно облачного сервиса была необходимость интеграции идей и знаний по данной прикладной проблеме, специалистов по решению задач данного класса на суперкомпьютерных системах, различных технологий моделирования и множества пакетов прикладных программ, а также различных вычислительных ресурсов, имеющихся у ИПМ и его партнеров. Итогом работы стал прототип облачной среды, реализованный в виде сервиса Мультилогин и прикладного программного обеспечения доступного из виртуальных машин пользователей. Первым приложением сервиса стала параллельная программа Flow_and_Particles для суперкомпьютерных расчетов многомасштабных задач газовой динамики в микроканалах сложных технических систем и визуализатор результатов расчетов Flow_and_Particles_View.

Ключевые слова: облачные сервисы и технологии; многомасштабные задачи газовой динамики; суперкомпьютерное моделирование.

DOI: 10.15514/ISPRAS-2015-27(6)-26

Для цитирования: Поляков С.В., Выродов А.В., Пузырьков Д.В., Якобовский М.В. Облачный сервис для решения многомасштабных задач нанотехнологии на суперкомпьютерных системах. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 409-420. DOI: 10.15514/ISPRAS-2015-27(6)-26.

1. Введение

Настоящая работа посвящена развитию распределенных облачных вычислений при решении задач нанотехнологий. Конкретная задача состоит в создании облачного сервиса для многомасштабного моделирования нелинейных процессов в полидисперсных многокомпонентных средах с

помощью гетерогенных кластеров и суперкомпьютеров. Фундаментальность и актуальность общей проблемы состоит в том, что в настоящее время в связи с внедрением нанотехнологий во многих отраслях промышленности существует острая необходимость объединения различных математических подходов, информационных и вычислительных ресурсов в единый аппарат суперкомпьютерного моделирования. Наиболее удачным способом объединения является создание соответствующих облачных сред и сервисов, в которых каждый пользователь сможет иметь доступ ко всем возможным информационным материалам, моделирующим программам, вычислительным ресурсам и индустриальным САПР.

В рамках настоящей работы предлагалось создать прототип облачного сервиса на базе кластеров ИПМ им. М.В. Келдыша РАН, ориентированного на суперкомпьютерное моделирование задач нанотехнологий методами механики сплошной среды и молекулярной динамики. Специфика выбранного направления научных исследований связана с распределенной параллельной обработкой на кластерах и суперкомпьютерах больших объемов данных, связанных с моделируемой средой как на макроуровне, так и в микромире.

Примером конкретной задачи в выбранной прикладной области может служить моделирование многомасштабных нелинейных процессов взаимодействия газовых сред со стенками металлических каналов в технических микросистемах [1-3], используемых в нанотехнологиях. Специфика подобного рода задач состоит в проведении множества детальных вычислительных экспериментов, касающихся определения как свойств отдельных веществ (металлов и газовых сред), так и свойств результирующих многокомпонентных и многофазных течений газов в рассматриваемых реальных технических системах. Проведение указанных вычислительных экспериментов невозможно без использования самой современной высокопроизводительной компьютерной и суперкомпьютерной техники. Использование последней без создания соответствующих интегрирующих сред и сервисов (каковыми и являются облачные среды и сервисы) существенно снижает эффективность параллельных вычислений, реализованных в прикладном программном обеспечении ПО.

Конкретными задачами настоящей работы были:

- организация облачной среды и виртуального пространства пользователей;
- реализация удобного и эффективного интерфейса пользователей с доступными им различными удаленными компьютерными и суперкомпьютерными системами;
- разработка и верификация прикладного программного обеспечения, предназначенного для решения многомасштабных задач нанотехнологии на примере расчета течений газовых смесей в металлических микроканалах;

- разработка системы управления и мониторинга расчетными заданиями пользователей на удаленных вычислительных системах;
- разработка системы хранения, пред- и пост- обработки, а также визуализации больших массивов распределенных данных, связанных с проведением вычислительных экспериментов на удаленных кластерах и суперкомпьютерах.

Для решения поставленных задач и достижения общих целей работы к настоящему моменту получены результаты, обсуждаемые в последующих пунктах.

2. Система KIAM Multilogin

В рамках работы разработана система KIAM Multilogin, которая является облачным VDI сервисом (удаленный рабочий стол), предоставляющим доступ пользователям к персональным виртуальным машинам. Доступ для пользователей возможен как из Интранет, так и из Интернет сетей по открытым (HTTP) и шифрованным (HTTPS, SSH, VPN) протоколам. Виртуальные машины управляются двумя основными типами ОС: Linux и Windows. Виртуальные машины используются для:

- работы с прикладными пакетами, системами моделирования и научными базами данных;
- создания и тестирования собственных приложений;
- запуска их на счет на доступных вычислительных ресурсах (внутренних и внешних);
- мониторинга за запущенными приложениями;
- обработки, анализа и визуализации результатов расчётов.

Вся система KIAM Multilogin построена исключительно на программных компонентах с открытым кодом таких как: Centos Linux, Ovirt, OpenStack, Ceph, Apache Directory Server, GlusterFS и пр. Система Мультилогин построена с учетом современных тенденций SDDC (программно-определяемый дата-центр). Все программные компоненты установлены на сервера x86-x64 архитектуры. В серверах установлены стандартные жесткие диски. В качестве ЛВС используются простые коммутаторы Ethernet 1 Гбит/с.

Архитектура системы KIAM Multilogin показана на рис. 1. Она состоит из четырех основных элементов: подсистемы удаленного доступа и маршрутизации, подсистемы виртуализации, хранения виртуальных машин и доступа к их рабочим столам, подсистемы аутентификации и авторизации, подсистемы хранения пользовательских данных.

Инфраструктура удаленного доступа и маршрутизации показана на рис. 2. В настоящий момент она обслуживается одним сервером (imm5), однако в будущем предполагает использование нескольких дублирующих серверов с целью повышения отказоустойчивости. Инфраструктура виртуализации,

хранения виртуальных машин и доступа к рабочим столам виртуальных машин в целом показана на рис. 3. Ее подсистема хранения пользовательских данных показана на рис. 4. Последняя реализована посредством 10 серверов с суммарным объемом дисковой памяти 50 Тб и полезным пространством 20 Тб.

Рис. 1. Архитектура системы Multilogin.

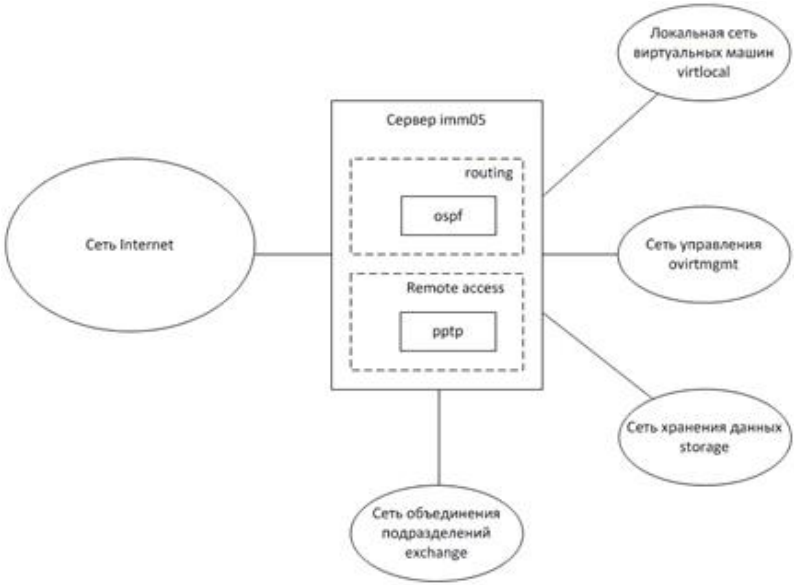


Рис. 2. Инфраструктура удаленного доступа и маршрутизации.

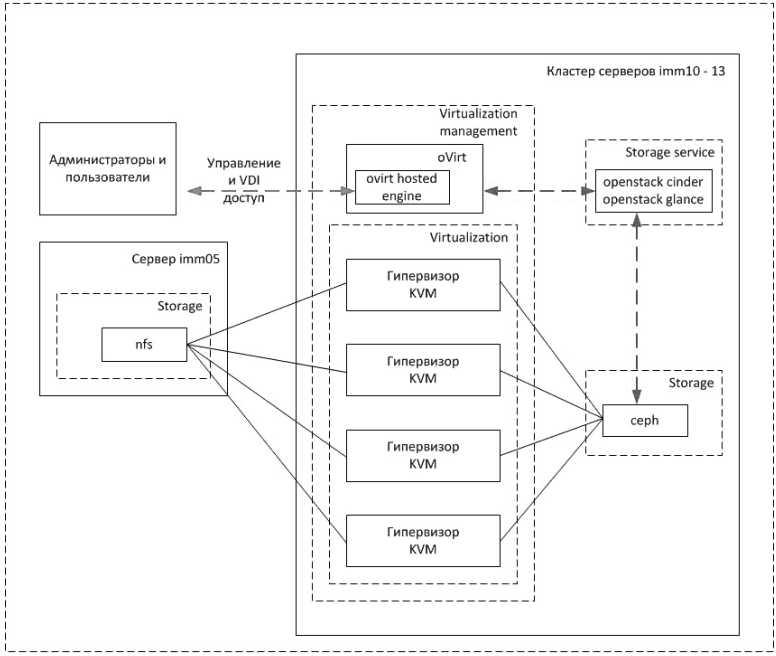


Рис. 3. Инфраструктура виртуализации, хранения виртуальных машин и доступа к рабочим столам виртуальных машин VDI.

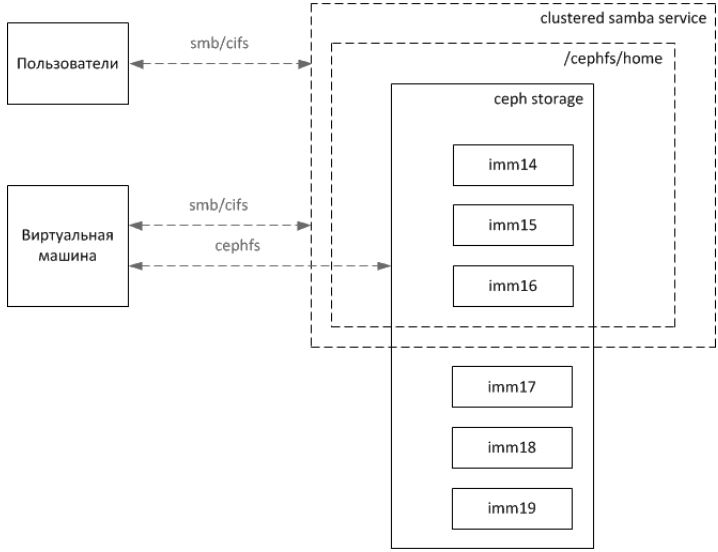


Рис. 4. Инфраструктура хранения пользовательских данных.

Система KIAM Multilogin позволяет быстро и своевременно добиваться требуемых результатов благодаря, в том числе:

- высокому уровню доступности;
- отказоустойчивости от единичных сбоев оборудования;
- эффективному использованию вычислительных ресурсов;
- минимальному времени простоя при регламентных работах.

Система KIAM Multilogin отличается от обычных распределенных Web-сервисов аппаратно-программной независимостью сессий пользователей и повышенной отказоустойчивостью. При этом допускается длительный период жизни одной сессии, подключение к ней с разных компьютеров, а также миграция активной сессии с одного физического сервера на другой.

3. Система KIAM Job_Control

Также в рамках работы создана система управления и мониторинга расчетными заданиями пользователей KIAM Job_Control, выполняющимися на различных кластерах и суперкомпьютерах, расположенных как в локальной сети ИПМ им. М.В.Келдыша РАН, так и за ее пределами. Основной задачей системы является эффективное управление расчетами пользователя в ситуации, когда объем контрольной точки составляет несколько гигабайт и более, а результирующие данные могут превосходить терабайт.

Стратегия системы состоит в том, чтобы спрогнозировать очередной запуск задания на основе данных о его положении в системах очередей доступных кластеров и суперкомпьютеров и обеспечить к моменту запуска наличие актуальной расчетной точки на данном вычислителе. После проведения кванта расчетов система должна обеспечить в фоновом режиме перекачку промежуточных или окончательных результатов расчетов на кластер хранения и обработки данных расчета.

От запущенного параллельного приложения требуется максимальная компактность контрольной точки и результатов расчетов, информация о минимальном размере кванта расчета на соответствующей параллельной конфигурации и времени сохранения данных (контрольной точки и результатов), а также маски сохраняемых файлов. Данная информация может быть записана приложением в специальный файл или введена пользователем при настройке интерфейса мониторинга.

Обращение к системе KIAM Job_Control возможно как напрямую с компьютера пользователя (находящегося либо в локальной сети Института или во внешней сети) в режиме "limited_access", так и из виртуальной машины в режиме "full_access". Таким образом, полный функционал осуществляется только через сервис KIAM Multilogin.

4. Приложение Flow_and_Particles

В качестве первого приложения создаваемого облачного сервиса выбрана задача многомасштабного моделирования течений газов в микроканалах технических систем в условиях многих масштабов расчетной области. В частности, рассмотрена задача о течении азота в никелевом микроканале. Основное внимание в этой задаче уделяется расчету макропараметров газовой среды. Различие в масштабах расчетной области (длина канала, поперечное сечение канала, длина свободного пробега молекул, толщина пограничного слоя) и приповерхностное взаимодействие газа с металлом приводят к необходимости учитывать рельеф и свойства микроканала на молекулярном уровне. В результате математическая модель исследуемого течения не может быть полностью сформулирована в рамках макроскопического подхода.

При реализации математической модели используется мультимасштабный подход, сочетающий решение уравнений квазигазодинамики (КГД) и коррекцию газодинамических параметров методом молекулярной динамики (МД). Общий алгоритм представляет собой расщепление по физическим процессам. КГД система уравнений решается методом конечных объемов. Система уравнений МД используется в качестве подсеточного алгоритма, применяющегося внутри каждого контрольного объема, и решается с помощью схемы Верле. В МД-вычислениях взаимодействие частиц описывается с помощью потенциалов, определяющих основные свойства компонент газовой смеси. Подробно этот подход освящен в работах [1-3].

Параллельная реализация подхода основана на методах расщепления по физическим процессам и разделения областей. Компьютерная реализация выполнена в виде приложения Flow_and_Particles, входящего в программный комплекс GIMM_NANO [4] (разработан в рамках госконтракта № 07.524.12.4019 Минобрнауки РФ), и ориентирована на использование вычислительных систем с центральной и гибридной архитектурами. При ее создании использовались концепция гибридной параллельной вычислительной платформы [5] и такие технологии параллельного программирования как MPI, OpenMP и CUDA.

Тестирование разработанного приложения Flow_and_Particles проводилось на суперкомпьютерах К100 (ИПМ им. М.В. Келдыша РАН), МВС-10П (МСЦ РАН), кластер с сетью Ангара (АО "НИЦЭВТ"). Предварительные расчеты показали, что общий численный алгоритм устойчив к использованию корректирующих течение данных, полученных в результате МД-вычислений. С его помощью методами МД были получены основные коэффициентные зависимости для КГД-системы, проверен переход от МД к КГД и обратно, произведен расчет плоского и полностью трехмерного течений в микроканалах с диаметрами от 10 до 30 мкм и длиной от 60 до 120 мкм. Полученные результаты подтвердили эффективность разработанного подхода.

5. Программа визуализации Flow_and_Particles_View

Для просмотра результатов расчетов программы Flow_and_Particles разработана программа распределенной визуализации Flow_and_Particles_View, предназначенного для сбора, обработки и визуализации распределенных результатов моделирования, расположенных на удаленном вычислительном кластере и хранящихся частями в различных директориях сетевой файловой системы. Реализация программного комплекса выполнена на языке программирования Python с использованием известных пакетов для анализа данных SciPy [6] и средства визуализации Mayavi [7]. В качестве системы управления используются скрипты, которые можно запускать на узле визуализатора по SSH или с помощью IPython notebook [8], предоставляющего веб-интерфейс для пользовательских задач и позволяющего просматривать результаты в браузере.

Разработанный программный код позволяет обрабатывать большие объемы данных в интересующие моменты времени и в выделенных зонах расчетной области. Также он позволяет параллельно с расчетами формировать дискретные кадры различных макрохарактеристик процесса и собирать их в видео-файл. Сбор данных производится по сети по протоколам SSH и SFTP, а так же посредством чтения из локальных директорий (или с использованием NFS или любых других способов). Также рассмотрены некоторые способы ускорения вычислений с помощью пакетов Numpy и Numba [9].

В результате применения программы при исследовании взаимодействия газа с металлической пластиной удалось в деталях наблюдать эффект адсорбции [3], который очень важен для многих практических приложений. В качестве иллюстрации работы программы на рис. 5, показан эффект адсорбции азота на поверхности никеля на выбранном пользователем участке. Для получения данного и других изображений, составляющих видеоролик об эволюции процесса адсорбции, пришлось обработать около 1 Тб распределенных данных, рассчитанных на МВС-10П (МСЦ РАН) и сохраненных на кластерах ИПМ им.М.В.Келдыша РАН и АО "НИЦЭВТ".

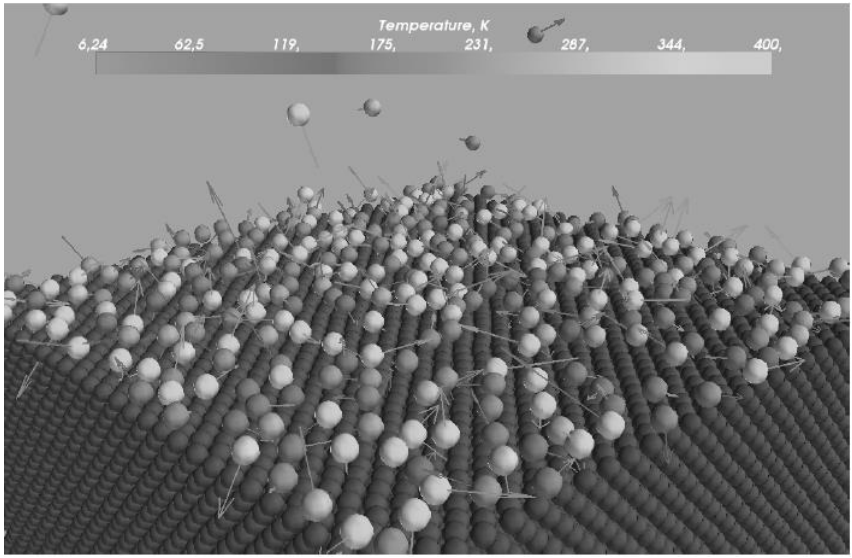


Рис. 5. Результат визуализации эффекта адсорбции азота на поверхности никеля.

Работа выполнена при поддержке Российского фонда фундаментальных исследований (проекты №№ 13-01-12073-офи_м, 15-07-06082-а, 15-29-07090-офи_м).

Список литературы

[1]. Ю.Н. Карамзин, Т.А. Кудряшова, В.О. Подрыга, С.В. Поляков. Многомасштабное моделирование нелинейных процессов в технических микросистемах. Математическое моделирование, 27(7), 2015. С. 65-74.

[2]. В.О. Подрыга, С.В. Поляков, Д.В. Пузырьков. Суперкомпьютерное молекулярное моделирование термодинамического равновесия в микросистемах газ-металл. Вычислительные методы и программирование, 16(1), 2015. С. 123-138.

[3]. В.О. Подрыга, С.В. Поляков, В.В. Жаховский. Атомистический расчет перехода в термодинамическое равновесие азота над поверхностью никеля. Математическое моделирование, 27(7), 2015. С. 91-96.

[4]. А.А. Бондаренко, С.В. Поляков, М.В. Якобовский, О.А. Косолапов, Э.М. Кононов. Программный комплекс GIMM_NANO. Международная суперкомпьютерная конференция "Научный сервис в сети Интернет: все грани параллелизма", 23 - 28 сентября 2013 г., г. Новороссийск, CD-proceedings, 1-5 pp.

[5]. С.В. Поляков, Ю.Н. Карамзин, О.А. Косолапов, Т.А. Кудряшова, С.А. Суков. Гибридная суперкомпьютерная платформа и разработка приложений для решения задач механики сплошной среды сеточными методами. Известия ЮФУ. Технические науки, № 6 (131), 2012. С. 105–115.

[6]. SciPy official site — http://www.scipy.org/

[7]. Mayavi official site — http://code.enthought.com/projects/mayavi/

[8]. IPython official site — http://ipython.org/

[9]. Numba official site — http://numba.pydata.org/

Cloud Service for Decision of Multiscale Nanotechnology Problems on Supercomputer Systems

S. Polyakov <polyakov@imamod.ru>
A. Vyrodov <vyrodov.alexey@gmail.com>
D. Puzyrkov <dpuzyrkov@gmail.com>
M. Yakobovskiy <lira@imamod.ru>
Keldysh Institute of Applied Mathematics of RAS,
4 Miusskaya square, Moscow, 125047, Russian Federation

Abstract. In work the structure and separate components of the cloudy service intended for the solution of multi-scale problems of nanotechnology on supercomputer systems are presented. The need of integration: (a) an ideas and knowledge on this applied problem, (b) a specialists in this scientific field and a programmers for the supercomputer systems, (c) various technologies of modeling and a set of packages of applied programs, (d) various computing resources which are available for the Institute and its partners – was motivation to creation of cloudy service. The prototype of the cloudy environment realized in the form of service Multilogin and the applied software available on virtual machines of users became a results of this work. The first applications of created service are (a) the Flow_and_Particles parallel program for supercomputer calculations of multi-scale gasdynamics processes in micro-channels of technical systems and (b) the Flow_and_Particles_View visualizer of distributed computation results. Use of the developed service allowed to increase efficiency of scientific research in the chosen applied field. In particular, by means of the developed service it was succeeded to calculate and analyze details of interaction of nitrogen stream with a nickel surface. This problem is characterized by large volume of calculated data. For example, representative selection of the control points relating to various time-points borrows in the sum of 3-4 Tb for each variant of calculation. By means of the developed software it was succeeded not only to keep these distributed data, but also to construct according to them necessary graphical diagrams and video.

Keywords: cloud services and technologies; multiscale gasdynamics problems; supercomputer simulations.

DOI: 10.15514/ISPRAS-2015-27(6)-26

For citation: Polyakov S., Vyrodov A., Puzyrkov D., Yakobovskiy M. Cloud Service for Decision of Multiscale Nanotechnology Problems on Supercomputer Systems. *Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015,* pp. 409-420 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-26

References

[1]. Yu.N. Karamzin, T.A. Kudryashova, V.O. Podryga, S.V. Polyakov. Mnogomasshtabnoe modelirovanie nelineynykh processov v tekhnicheskikh mikrosystemakh [Multiscale simulation of non-linear processes in technical micro-systems]. Matematicheskoe modelirovanie [Mathematical Models and Computer Simulations], 2015, vol. 27, no. 7, pp. 65-74 (in Russian).

[2]. V.O. Podryga, S.V. Polyakov, D.V. Puzyrkov. Superkompyuternoe molekulyarnoe modelirovanie termodinamicheskogo ravnovesiya v mikrosystemakh gas-metall [Supercomputer molecular simulation of thermodynamics equilibrium in micro-systems with gas and metal]. Vychislitelnye metody i programmirovanie [Numerical Methods and Programming], vol. 16, no. 1, 2015, pp. 123-138 (in Russian).

[3]. V.O. Podryga, S.V. Polyakov, V.V. Zhakhovsky. Atomisticheskiy raschet perekhoda v termodinamicheskoe ravnovesie azota nad poverkhnostyu nikelya [Atomistic calculation of transition to thermodynamic balance of nitrogen over a nickel surface]. Matematicheskoe modelirovanie [Mathematical Models and Computer Simulations], 2015, vol. 27, no. 7, pp. 91-96 (in Russian).

[4]. A.A. Bondarenko, S.V. Polyakov, M.V. Yakobovskiy, O.A. Kosolapov, E.M. Kononov. Programmnyi kompleks GIMM_NANO [The GIMM_NANO software suite]. *International Supercomputer Conference "Scientific service in Internet network: all sides of parallelism", 2013, September 23-28, Novorossiysk (Russia), CD-proceedings*, pp. 333-337 (in Russian).

[5]. S.V. Polyakov, Yu.N. Karamzin, O.A. Kosolapov, T.A. Kudryashova, S.A. Sukov. Khibridnaya superkompyuternaya platforma i razrabotka prilozheniy dlya resheniya zadach mekhaniki sploshnoy sredy setochnymi metodami [Hybrid supercomputer platform and applications programming for the decision of continuum mechanics problems by grid methods]. Izvestiya YuFU. Technicheskie nauki [News of the Southern Federal University. Technical Sciences], 2012, no. 6 (131), pp. 105–115 (in Russian).

[6]. SciPy official site — http://www.scipy.org/

[7]. Mayavi official site — http://code.enthought.com/projects/mayavi/

[8]. IPython official site — http://ipython.org/

[9]. Numba official site — http://numba.pydata.org/

Облачный фреймворк для интеграции сетевых экспертных и аналитических средств

[1]А.Н. Ермаков <alexei-ermakov@yandex.ru>
[2]С.В. Клименко <stanislav.klimenko@gmail.com>
[1]А.А. Меркулов <merkulov@cnsa.ru>
[1]С.А. Панфилов <serelen@list.ru
[3]А.Н. Райков <alexander.n.raikov@gmail.com>
[1]Аналитическое агентство «Новые стратегии»,
119526, Россия, г. Москва, ул. 26 Бакинских Комиссаров 14-100
[2]Институт физико-технической информатики,
142281, Россия, Московская обл., г. Протвино, Заводской пр., 6
[3]Институт проблем управления РАН,
117997, Россия, г. Москва, ул. Профсоюзная, 65

Аннотация. Информационно-аналитические системы поддержки решений во власти и бизнесе, как правило, носят распределенный характер. В них можно выделить два существенно отличающихся контура информационно-аналитического обеспечения. Первый основывается на поддержке процессов сбора, доставки и обработки информации, представленной в базах данных, а второй - обеспечивает собственно процессы принятия решений с анализом мнений экспертов в реальном времени. Участники принятия решений в системе распределенных ситуационных центров искусственно погружаются в обстановку виртуальной реальности, но технологический мостик между разделенными расстояниями участниками принятия решений препятствует росту их качества, достижению согласия участников относительно целей и путей действий, особенно при работе в гетерогенных средах, когда требуется переключение интерфейсов и программных продуктов. Встает вопрос ускорения процесса согласования группового решения. В работе предлагается подход и практическая реализация проектных решений по использованию методов ситуационной осведомленности, виртуального сотрудничества, облачных вычислений. Особое место занимает новый подход к верификации когнитивных моделей на основе анализа Больших Данных. В основе предложенного алгоритма лежит предположение, что суждение о наличии связи (взаимовлияния) между факторами когнитивной модели может быть обнаружено в текстах документов из массивов больших данных. Предложен также фреймворк менеджмента знаний для интеграции гетерогенных информационно-аналитических средств в облачной среде. Фреймворк поддерживает целенаправленную и устойчивую сходимость процессов принятия решений. В основу построения структуры фреймворка положен один из наиболее гибких подходов к

построению фреймворка менеджмента знаний и авторский метод решения обратных задач в топологических пространствах с применением генетических алгоритмов. Отмечается, что предлагаемый подход сформировался при создании порядка 50 проектов в области стратегического анализа, информационно-аналитических систем.

Ключевые слова: визуализация; верификация; гетерогенность; когнитивная модель; конвергентность; облачная среда; сетевая экспертиза; большие данные.

DOI: 10.15514/ISPRAS-2015-27(6)-27

Для цитирования: Ермаков А.Н., Клименко С.В., Меркулов А.А., Панфилов С.А., Райков А.Н. Облачный фреймворк для интеграции сетевых экспертных и аналитических средств. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 421-440. DOI: 10.15514/ISPRAS-2015-27(6)-27.

1. Введение

Достижение согласия участников принятия решений замедляется при взаимодействии членов команды в распределенной, сетевой, среде. В подобных средах участники принятия решений ограничиваются обменом текстовых и голосовых сообщений. Участники могут видеть друг друга, но чувства общности, присутствия, доверия, ощущения причастности ослабляются.

При работе в системе распределенных ситуационных центров участники искусственно погружаются в обстановку виртуальной реальности, но технологический мостик между разделенными расстояниями участниками совещаний постоянно ощущается, особенно при работе в гетерогенных средах, когда требуется переключение интерфейсов и программных продуктов, с которыми работают различные участники. Встает вопрос ускорения достижения взаимопонимания при реализации процесса согласования группового решения.

Особую сложность в решении проблемы представляет принятие решений в чрезвычайных ситуациях, когда решения не экстраполируются из предшествующего опыта, типовые фреймворки решений отсутствуют, время принятия решений резко ограничено, требуется быстрое достижение согласия участников принятия решений относительно целей действий. При этом стоит фундаментальная задача обеспечения целостности модельного когнитивного представления ситуации.

Задержки в принятии решений особенно недопустимы в условиях нештатных, аварийных ситуаций, когда при резком дефиците времени на принятие решений приходится согласовывать решения в разнохарактерных коллективах людей (руководство, спасательные бригады, эксперты, население и пр.) [1]. Здесь приходится сочетать иерархическую и сетевую организацию работ различных ведомственных образований. При этом сетевые команды быстрее действуют, чем иерархические, в них скорее распространяется информация, однако скорость достижения согласия больше зависит не столько от скорости

коммуникаций, сколько от достижения взаимопонимания относительно вербальных и визуальных представлений различных аспектов проблемной ситуации. Произнесенное слово может быть участниками понято по-разному, а показанный всем участникам визуальный образ – направить их действия в разные стороны. Важен целостный контекст, нужна корректная семантическая интерпретация событий.

Вместе с тем контекст может быть очень сложным. Это связано с необходимостью учета большого множества факторов, характеризующих проблемную ситуацию, а также воздействия субъективных мнений, поскольку в работе участвуют люди. Например, при планировании деятельности крупной корпорации, государственного органа, в состав участников принятия решения может войти несколько десятков человек руководящего состава, а также несколько групп независимых экспертов. При этом число взаимосвязанных факторов создаваемых ими моделей может достигать нескольких сотен. Факторы и взаимосвязи нуждаются в содержательной интерпретации для того, чтобы участники быстрее понимали друг друга. Часто для этого используют инструмент лингвистических средств: словарей, глоссариев, представления знаний. Вместе с тем, как оказалось, этого недостаточно, поскольку участники не имеют времени читать определения огромного числа факторов. Мало того, если даже лингвистический подход и помогает разобраться с улучшением взаимопонимания терминов, то он не обеспечивает должным образом интеграцию различных аналитических инструментов, программ, моделей, которые используются при принятии групповых решений.

В таких случаях необходимо интегрированное решение фундаментальной проблемы наиболее оптимальной интеграции методов и средств обработки и анализа больших данных в сочетании с когнитивным моделированием в среде групп экспертов. При этом требуется погружение сетевых экспертов в виртуальную среду ситуационной осведомленности [1].

В настоящей работе акцентируется внимание на разработке механизма интеграции гетерогенных экспертно-аналитических инструментов для области государственного управления и крупного бизнеса. Механизм строится на основе построения специального фрейма-интегратора с применением методов менеджмента качества, сетевой экспертизы, 3D-визуализации, стандартов по жизненному циклу продукции и предоставлению услуг. При этом осуществляется выбор типового фреймворка менеджмента знаний, а также предусматривается возможность обращения к механизмам краудсорсинга и автоматической верификации концептуальных моделей с помощью анализа Больших Данных.

Для получения должной синергии осуществляется комплексное решение технологических задач: адаптации имеющихся у авторов средств анализа Больших Данных, модернизации облачной среды поддержки сетевой экспертизы и электронных мозговых штурмов, группового когнитивного

моделирования, обеспечения высокого уровня согласованности экспертных оценок за счет решения обратных задач на когнитивных графах.

При этом облачный сервис помогает обеспечить новые виды и качественное изменение процессов принятия решений, а именно: услуга становится более простой и доступной; участники оптимально расходуют свои ресурсы; пользователю не нужно запасаться дорогими и наукоемкими технологиями.

При создании фрейма-интегратора используется авторский подход конвергентного управления, помогающий декомпозировать проблему на большое множество простых частей и потом собрать их в единое целое с получением нового качества [2].

2. Существующие подходы и ограничения

Теоретический анализ и практическая апробация создания существующих методов и подходов показывает, что они не гарантируют эффективной сходимости (конвергентности) процессов проведения совещаний и экспертных процедур к заданным целям, особенно в условиях большой неопределенности и нештатных ситуаций [2].

Экспертные процессы часто носят дивергентный характер, множество высказываний экспертов трудно привести к одному знаменателю, построить решение, удовлетворяющее конечного пользователя. Если таковым пользователем является команда людей, то дивергентные процессы и неустойчивость процедур обсуждения затрудняют приход к согласию относительно целей и путей действий в нештатной ситуации. При этом считается, что процессы голосования не помогают достигать высокого уровня согласия, что особенно важно с мотивационной стороны при стратегическом планировании и в условиях нештатной ситуации.

Устранение имеющихся трудностей авторский коллектив ищет на пути развития конвергентного подхода к управлению, способствующему ускорению сходимости и повышению качества групповых управленческих решений. Этот подход основан на интеграции теоретических основ методов: решения обратных задач на топологических пространствах, управляемой термодинамики, квантовой семантики – создании упомянутого выше фрейма-интегратора.

Для решения вопросов интеграции гетерогенных функциональных сред в настоящее время авторами используются следующие методы и подходы:

- ситуационной осведомленности, неогеографии, виртуального окружения, серьезных игр, облачных вычислительных сред, интеллектуальных информационных технологий, обработки, анализа Больших Данных;
- анализа геопространственной и семантической интерпретации данных с применением когнитивного моделирования. Система виртуального окружения имеет интуитивный интерфейс для доступа к информации.

Интерактивные возможности среды обеспечивают вызов требуемой информации для любого объекта и точки на карте;

- интерпретации выводов теоремы А.Н.Тихонова для нечетких топологических пространств с определением условий обеспечения целенаправленности групповых процессов принятия решений [2], а также условий обеспечения устойчивости поведения термодинамических систем [3];
- модернизация имеющегося облачного сервиса сетевой экспертизы в части уточнения способа интеграции метода когнитивного моделирования и генетического алгоритма системы АрхиДока [4] с методами анализа Больших Данных.

Интеграцию гетерогенных функциональных сред авторы осуществляют путем реализация фрейма-интегратора, При этом учитываются особенности наиболее продвинутого механизма, а скорее институционального образования, для групповой поддержки принятия решений - системы распределенных ситуационных центров. Состав системы распределенных ситуационных центров может быть проиллюстрирован рис. 1. Обязательным функциональным компонентом этой системы является экспертно-аналитическая подсистема, обеспечивающая поддержку процедур принятия решений среди участников, включая внешних экспертов, работающих в удаленном (сетевом) режиме.

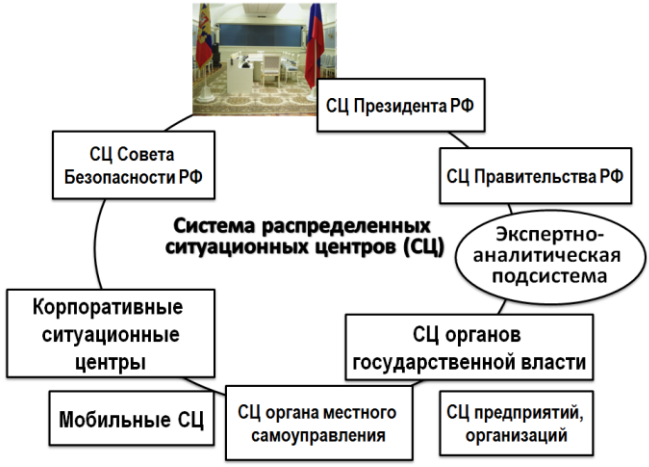


Рис. 1. Система распределенных ситуационных центров

Подобная экспертно-аналитическая подсистема включена в Федеральную государственную информационную систему территориального планирования на всех (федеральном, региональном и муниципальном) уровнях управления Российской Федерации. При ее разработке учитывались аспекты возможной

потребности совместного экспертного и геоинформационного обеспечения процессов принятия решений в нештатных чрезвычайных и аварийных ситуациях.

В нештатных условиях создание формальных количественных моделей, опирающихся на традиционные показатели и индикаторы, далеко не всегда представляется возможным. Для задач подобного типа характерны латентность, хаотичность, риски, квантованность, неопределенность, описание на качественном уровне, неоднозначность последствий решений возникающих проблем. При решении таких проблем нужны интуиция, опыт, ассоциативность мышления, догадки, экспертные оценки.

Для учета подобных характеристик при моделировании авторами используются методы когнитивного (познавательного) моделирования, в основе которых лежит разработка концептуальных моделей развития ситуаций, учитывающих не только реальную ситуацию, но и специфику происходящих в ней процессов [5]. Когнитивные модели создаются с целью оценки и прогноза динамики безопасности, реализации проблемного мониторинга ситуации, прогнозирования развития и комплексной оценки воздействия различных направлений действий.

Экспертно-аналитический модуль, по всей видимости, потребуется и в Федеральной информационной системе стратегического планирования, разработка которой предусмотрена Федеральным законом от 28.06.2014 № 172-ФЗ "О стратегическом планировании в Российской Федерации". Экспертно-аналитическая система предназначена для поддержки проведения сетевой экспертизы путем обеспечения:

- ведения реестров экспертов и оценки рейтинга экспертов;
- сбора комментариев экспертов по вопросам;
- анкетный опрос экспертов с заданием лингвистических шкал;
- организации сетевого экспертного мониторинга ситуации;
- проведения электронного мозгового штурма;
- проведения сетевого стратегического совещания;
- устойчивой сходимости мнений участников к согласованному решению и др.

Достаточно новым в организации сетевых процессов принятия решений может стать именно создание упомянутого выше фрейма-интегратора, который обеспечит «бесшовное» взаимодействие распределенных участников принятия решений, работающих в гетерогенной функциональной среде, создаваемой различными информационно-аналитическими средствами. Этот фрейм-интегратор может быть представлен в виде соответствующего дружественного интерфейса.

3. Фрейм-интегратор аналитических методов и инструментов

Фрейм-интегратор аналитических методов и инструментов зависит от предметной области. Так, если предметной областью является сфера государственного управления или крупного производственного бизнеса, то взаимосвязь и функционирование аналитических методов и инструментов, которые используются при сетевых групповых процессах принятия решений, могут быть обеспечены на основе использования:

- метода структурирования функций качества;
- стандартов, типа ГОСТ Р ИСО 15704 по организации жизненных циклов и истории жизни предприятия;
- продвинутого фреймворка менеджмента знаний.

Хорошо известный метод структурирования функций качества основан на детальном анализе потребностей внешнего окружения, учете бизнес-процессов, регламентированной экспертизе и пр. Он ориентирован на получение высококачественного результата по производству продукции или предоставления услуг. Этот метод реализуется путем декомпозиции большой проблемы на части (до сотен и тысяч частей), экспертной оценки частей, а затем интеграции отдельных частей в целое. При этом пошагово регламентируется процедура этого процесса. Все оценки и сравнения делаются экспертами. Тогда, каждый участник процесса благодаря подключению механизма экспертизы получает максимально достоверное знание связи между выполняемой ими работой и степенью удовлетворенности потребителя продукцией в целом.

Применение метода структурирования функций качества и упомянутого выше стандарта по организации жизненных циклов и истории жизни предприятия позволяет сформировать верхнеуровневую архитектуру искомого фрейм-интегратора. Создание такого интегратора подразумевает разработку соответствующих межкомпонентных интерфейсов, позволяющих обрабатывать информацию бесшовным образом, делать процессы интероперабельными. Место и общая архитектура фрейм-интегратора, обеспечивающего функциональную взаимосвязь различных аналитических компонент, проиллюстрирована рис. 2.

При работе в гетерогенной информационно-аналитической среде обработка информации, логическое формирование выводов, определение уровня доверия к результатам вывода и пр. требует использования механизмов менеджмента знаний. Для типизации фреймворка менеджмента знаний авторами настоящей работы проведен соответствующий анализ существующих методов менеджмента знаний и предложен типовой фреймворк.

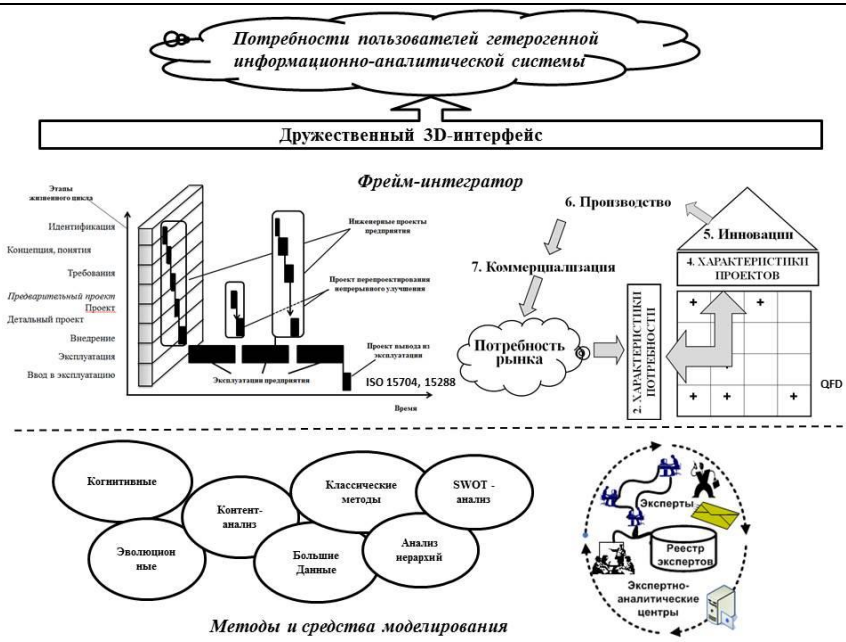


Рис. 2. Архитектура функционального фрейм-интегратора

4. Фреймворк менеджмента знаний

Знания - это фактор совершенствования, облегчающий приспособление к постоянным изменениям рынка, внедрение инноваций. Как известно, способность к оптимальному применению знаний вносит значительный вклад в организационную устойчивость и выживание. Знания и связанные с ними нематериальные активы все чаще расценивают как фундамент организационного успеха, как в частном, так и в государственном секторе. Менеджмент знаний дает наибольший эффект в коммерческом секторе [6]. Однако государственный сектор также может использовать накопленный опыт для:

- адаптации деятельности к существующей внешней среде;
- при создании устойчивой технологической инфраструктуры;
- при максимизации результативности и эффективности;
- для повышения компетентности управления;
- при наличии лояльных и инициативных сотрудников;
- для обеспечения более полного удовлетворения потребностей потребителей услуг.

В настоящее время понятие «знание» трактуется достаточно широко. Это набор данных и информации. Согласно упомянутому стандарту, понятие «знание» включает различные комбинации новой технологии, производственного опыта, и, что важно отметить - эмоции, верования, значения величин, идеи, интуицию, любопытства, мотивации, стили обучения, отношения, способность доверять, способность решать сложные проблемы, открытость, умение работать в компьютерной сети, коммуникабельность, отношение к риску, наличие духа предпринимательства. Выделяются формализованного и неформализованные знания. Знание также может быть индивидуальным и коллективным. В настоящее время идентифицируются и применяются пять основных видов деятельности в отношении знаний: идентификация, создание, хранение, обмен знаниями и их применение [7]. При этом культура организации (корпоративная культура) является важнейшим фактором в успешном продвижении менеджмента знаний.

При создании информационных систем основное внимание традиционно уделяется формализованным знаниям. Данные знания кодируют объектами, словами, номерами, представляют в графической форме, в форме рисунков, спецификаций, учебников, процедур и т.п. Вместе с тем в процессе вхождения в 6-й и 7-й технологические уклады формирование неклассической и постнеклассических парадигм управления, когда на место объекта управления встает субъект или саморазвивающаяся полисубъектная среда [8], все большее значение приобретают неформализованные знания.

В этом контексте понятие менеджмента знаний - оксюморон, поскольку управлять понятиями и концептами, интерпретирующими знания, которые содержатся вне компьютера, в эмоциях и мыслях руководителей, сотрудников, экспертов, потребителей в явном виде невозможно. Поэтому здесь можно, с одной стороны, ограничить понятие «знания» формализованной структурой (логикой), что явно оторвет менеджмент знаний от реальной практики, а с другой - в состав знаний включить неформализуемые феномены.

В последнем случае требуется обеспечить целенаправленное решение задач и реализацию функций органов государственной власти или бизнес-компанией путем создания необходимых формализованных условий для такого решения. Для углубленного ознакомления с таким подходом можно обратиться к работе [1]. Этот подход создает необходимые условия для целенаправленного и устойчивого решения задач и исполнения функций в условиях большой неопределенности и необходимости отвечать на вопросы типа «Что делать?».

Такие задачи носят обратный характер, они некорректны – незначительные изменения исходных данных могут привести к существенному изменению результата решения. Вместе с тем можно найти необходимые условия, при которых задача становится устойчивой. Для этого информация и данные определенным образом структурируются. Предполагается, что сотрудник, лицо, принимающее решений, привносит в процесс решения свою, качественную, информацию. Обстоятельства могут быть неявными, а человек

(команда, организация и пр.) характеризоваться скрытыми возможностями и характеристиками [9]. В таких условиях на основе методов решения обратных задач [10] могут быть даны определенные рекомендации по структурированию информации, созданию соответствующего фреймворка, при принятии решений, например:

- сформируй в любом виде (качественном, количественном) главную цель;
- как можно более четко представление проблемы в виде цели, ресурсов и действий (или цели, функции, ресурсы);
- построй дерево целей, как минимум – трехуровневое, где первый уровень охватывает целое (главная цель, назначение, миссия, видение руководителя), второй – внешние цели (потребитель результата), а третий – внутренние (снижение издержек, внутренние целевые критерии действий по функциям);
- множество ресурсов и возможностей (функций и ресурсов) раздели на конечное и обозримое число частей (до 25);
- при решении задачи добейся позитивного результата продвижения к цели на каждом очередном шаге ее решения.

Это необходимые условия для обеспечения устойчивости решения задач в условиях большой неопределенности и неформализованном представлении исходных данных, особенно - цели. Для обеспечения достаточных условий стоит использовать специальные алгоритмы решения обратных задач в условиях, когда к решению подключены группы экспертов. Такие алгоритмы могут быть реализованы только с помощью компьютера и специальных программных средств, построенных на основе рассматриваемых в настоящем разделе методов искусственного интеллекта, прежде всего, когнитивного моделирования, анализа иерархий, анализа больших данных, современного сетевого интеллекта [11], типового фреймворка управления знаниями.

Методов менеджмента известно более 100, например, в их ряд можно поставить такие методы как:

- абдуктивного вывода (факты, гипотезы, объяснения);
- Диких карт, Черных лебедей;
- управляемого хаоса, теории катастроф;
- природных вычислений;
- обучающихся организаций;
- когнитивного диссонанса и др.

Анализ имеющегося множества методов показывает, что за основу построения искомого можно выбрать сравнительно новый и достаточно известный фреймворк Коллинса и Парсела, отличающийся своей полнотой и

динамичностью. Он делает акцент на окружении, в котором знание может быть создано, открыто, идентифицировано, передано другим, очищено, оценено, преобразовано, адаптировано и применено. Для создания окружения, в котором знание может развиваться, необходимо сформировать правильные:

- Условия. Надежную инфраструктуру и организационное обеспечение;
- Средства. Обобщенные модели, инструменты и процессы для обучения;
- Действия (активности). Люди инстинктивно ищут, распространяют и используют знания;
- Лидерство. Обучение и распространение знаний обусловлено ролями.

В результате применения этого фреймворка происходит синергия, успех мультиплицируется, особенно с учетом сетевого фактора распространения и обновления знаний.

Модернизированный фреймворк, отражающий также аспекты сетевой экспертизы и анализа Больших Данных приведены на рис. 3.

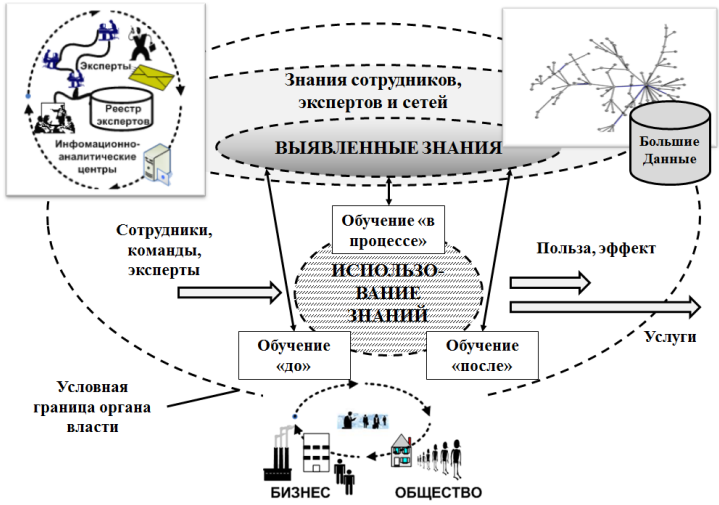


Рис. 3. Фреймворк менеджмента знаний

Рис. 3. иллюстрирует требуемый фреймворк менеджмента знаний, показывающий, что создавая и развивая методы и инструментарий менеджмента знаний следует комплексно учитывать различные направления формирования и извлечения знаний, прежде всего:

- знаний из внешнего окружения для понимания потребностей рынка и обеспечения устойчивости функционирования;
- знаний о внутренних процессах, что обеспечивает постоянное

снижение издержек функционирования и предоставления государственных и муниципальных услуг.

При этом необходимыми условиями успешного развития менеджмента знаний является:

- наличие согласия руководителей и сотрудников подразделений различного уровня управления относительно целей и путей действий (стратегия) с формированием соответствующих критериев ранжирования знаний;
- понимание менеджеров знаний (когнитологов) фундаментальных закономерностей поведения знаний, которые вытекают из результатов теоретических и практических исследований в различных дисциплинах, включая философию, психологию, политологию, эконометрику, кибернетику, физику, биологию и др.

5. Верификация когнитивной модели на основе Больших Данных

Важным вопросом использования аналитических инструментов является семантическая верификация (проверка) построенного вручную прототипа когнитивной модели. Например, прототип модели может выступать как результат достижения согласия экспертов по видению проблемной ситуации и требует проверки следующих положений:

- все ли факторы исследуемой проблемной области учтены экспертами в модели;
- выявлены все ли связи между факторами в модели;
- для каждой связи правильно ли установлен характер влияния одного фактора на другой.

В качестве инструментов верификации может выступать отображение данных прототипа модели на существующие открытые данные из Интернет, мнения экспертов, населения или сообществ, включая:

Под верификацией подразумевается анализ содержания релевантного информационного сообщения, содержащего суждение о структурных свойствах прототипа когнитивной модели. Рассмотрим алгоритм верификации прототипа когнитивной модели в части проверки наличия связи между отдельными факторами с применением данных из Интернет.

В основе алгоритма лежит предположение, что суждение/мнение о наличии связи (взаимовлияния) может быть обнаружено в текстах анализируемых документов. Данная задача относится к области анализа текстов и компьютерной лингвистики с использованием технологий Big Data (NoSQL, MapReduce, Hadoop и др.). Источниками информации (документов) выступают тематические ресурсы сети Интернет (новостные порталы, форумы экспертных сообществ и др.).

Алгоритм проверки наличия связи между двумя факторами когнитивной модели предусматривает несколько вариантов решения:

- «Связь присутствует с высокой вероятностью» – найден релевантный документ или несколько документов, в которых выполнены условия наличия в тексте ключевых слов обоих факторов и удовлетворяется критерий их близости;
- «Связь присутствует со средней вероятностью» – найден релевантный документ или несколько документов, в которых выполнены условия наличия в тексте ключевых слов обоих факторов, но не удовлетворяется критерий их близости;
- «Связь присутствует с низкой вероятностью» – не найден релевантный документ или несколько документов, в которых выполнены условия наличия в тексте ключевых слов обоих факторов. Для каждого фактора найдена своя совокупность релевантных документов и удовлетворяется критерий близости их словарных спектров;
- «Связь отсутствует» – для каждого фактора найдена своя совокупность релевантных документов, но не удовлетворяется критерий близости их словарных спектров, либо вообще не найдены релевантные документы.

Результаты проверки наличия связи могут быть различными для одних и тех же факторов когнитивной модели при разных временных «срезах» данных из окружающей информационной среды (изменились границы временного интервала поиска, дополнился/сократился список источников и др.). Алгоритм проверки наличия связи между двумя факторами представлен следующими этапами:

Этап 1. Предварительный анализ прототипа когнитивной модели:

- Морфологический анализ формулировок факторов;
- Формирование перечня ключевых слов для каждого фактора (приведение отдельных слов к нормальной форме, канонизация; исключение союзов, предлогов и др.);
- Дополнение перечня ключевых слов для требуемого фактора с использованием словарей терминов и синонимов предметной области. По завершению данного этапа для двух факторов формируются отдельные списки ключевых слов. Например, фактор 1 – "уровень", "заработная", "плата"; фактор 2 – "удовлетворенность", "население", "социальная", "услуга", "пенсионер".

Этап 2. Формирование запросов к поисковой системе:

- Конструирование запросов к поисковой системе: комбинированных – содержащих ключевые слова обоих факторов; индивидуальных – содержащих ключевые слова только одного из факторов. Определение

параметров поиска: глубина по времени, источники данных, предварительная фильтрация поисковых результатов;

- Получение результатов поисковых запросов: найдены документы как по комбинированным, так и по индивидуальным запросам; найдены документы только по индивидуальным запросам; документы не найдены. Если по завершению данного этапа документы не найдены, то пользователь соглашается с фактом отсутствия связи, либо уточняет перечень ключевых слов или условия поискового запроса.

Этап 3. Анализ найденных документов, подтверждение гипотезы о наличии связи между факторами.

- Анализ документов, содержащих ключевые слова обоих факторов. Проверка критерия близости ключевых слов обоих факторов в тексте документа. В случае положительного результата проверки – уведомление пользователя о наличии связи между факторами с высокой вероятностью. В случае отрицательного результата – уведомление о наличии связи между факторами со средней вероятностью.
- Анализ отдельных совокупностей релевантных документов для каждого фактора. Построение словарных спектров для групп документов и проверка критерия их близости. В случае положительного результата проверки – уведомление пользователя о наличии связи между факторами с низкой вероятностью. В случае отрицательного результата – уведомление об отсутствии связи между факторами.
- В случае установления факта отсутствия связи пользователь соглашается с достигнутым результатом, либо уточняет перечень ключевых слов или условия поискового запроса.

Порядок использования для верификации прототипа когнитивной модели мнений экспертов, профессиональных сообществ и населения проиллюстрирован схемой, приведенной рис. 4.

Схема демонстрирует, что в существующие технологии краудсорсинга следует встроить механизмы сетевой экспертизы, для того чтобы придать этому процессу конвергентный, сходящийся, характер.

Рис. 4. Интеграция механизмов краудсорсинга и сетевой экспертизы

6. Дружественный 3D-интерфейс

Показанная на рис. 2 архитектура функционального фрейм-интегратора требует создания дружественного 3D-интерфейса с пользователями. Экспоненциальным ростом объема информации требует ее визуализации. Все эти данные набираются со все возрастающими точностями, сохраняются в цифровом формате и анализируются с использованием компьютерных инструментов визуализации. При возрастающей сложности данных инструменты визуализации становятся особенно важны как для исследователей, так и для сотрудников органов государственной власти и бизнес-сектора. В частности, системы с полным эффектом присутствия (иммерсивные) являются привлекательной возможностью для изучения пространственных 3D данных не только молекул и астрофизические явлений, но и наборы геопространственных данных для территориального управления [12].

Новый тип пространств визуализации представляют собой среды гибридной реальности. Они стирают границу между виртуальным окружением и стеной из множества расположенных бок о бок мониторов высокого разрешения. Среда гибридной реальности является первой в мире практически сплошной системой окружения с эффектом погружения на основе плоских LCD панелей. Уникальным представляется возможность пользователя одновременно видеть 2D и 3D информацию, что обеспечивает большую гибкость в приложениях со смешанным представлением. При этом 2D представление разумно использовать для работы дежурного персонала, так как такие системы позволяют длительное использование без серьезного утомления оператора. Тогда как 3D представление следует использовать в задачах мозгового штурма, где существенным фактором является эффект погружения.

Десятилетиями системы погружения создавались на основе шлемов виртуальной реальности (Head Mounted Displays), которые обладали ограниченными разрешением и областью обзора. В 1992 году появилась новая парадигма в области виртуальной реальности – она была представлена кубом с длиной ребра 10 футов, с отображаемой проекторами графикой на пяти его гранях, что позволяло пользователям наблюдать 3D видеоизображение, пользуясь сравнительно легкими очками с активным затвором. Система была достаточно велика, чтобы вместить в себя несколько человек, которые одновременно могли наблюдать визуальные образы. Вдобавок трекинг головы позволял ученым исследовать сложные наборы данных, используя реализованное в системе взаимодействие.

Сравнительно недавно размещенные вплотную друг к другу LCD дисплеи появились как вариант практичной платформы для масштабной визуализации объёмов данных. Сконструированные из расположенных рядом мониторов, которые вместе формируют непрерывную поверхность для отображения (видеостену) проблемных ситуаций, такие системы часто занимают все пространство окружающих стен. По сравнению с вариантом на проекторах, стены LCD дисплеев обеспечивают изображения отличного качества и разрешения, зачастую достигающего от 100 до 300 мегапикселей, а также требуют меньше затрат на установку.

Для эффективного использования видеостен был разработан ряд программных продуктов. Одним из наиболее успешных среди них является SAGE (*Scalable Adaptive Graphics Environment*). Основной инновационной чертой этого продукта можно считать возможность запуска, отображения и управления взаимным расположением множества процессов визуализации. Эти продвижения в технологиях позволили объединить лучшие свойства иммерсивных систем виртуальной реальности с лучшими возможностями систем на основе видеостен ультравысокого разрешения, что привело к созданию концептуально новых иммерсивных систем окружения, которые мы назвали системами окружения гибридной реальности (*Hybrid Reality Environments* -- сокращенно HRE).

Представленные выше возможности являются основой, полностью соответствующей окружению гибридной реальности. Эти возможности используются для формирования соотвествующего дружественного 3D интерфейса.

7. Практический задел

По приведенным выше проектным решениям у авторов настоящей работы есть уникальный задел, особенно в части создания облачного сервиса сетевой экспертизы, когнитивного моделирования, стратегического планирования, визуализации, анализа Больших Данных, создания систем поддержки принятия решений в виде функционала ситуационных центров. Этот задел сложился в результате реализации с 2000 года порядка 50 проектов в области

стратегического анализа, информационно-аналитических систем, интеллектуальных информационных технологий и систем поддержки решений.

В частности, экспертно-аналитическая система внедрена в работу нескольких ситуационных центров органов государственной власти. Например, по заказу Минобрнауки России проведена разработка и апробация порядка формирования экспертного сообщества по вопросам государственной службы и кадров. Создан облачный сервис сетевой экспертизы по контракту с «Фондом содействия развитию малых форм предприятий в научно-технической сфере». Указанный сервис обеспечивает:

- ведение реестра экспертов с построением рейтинга;
- процесс сбора шкальных оценок и комментариев экспертов для последующей аналитической обработки и когнитивного моделирования;
- проведение сетевого экспертного стратегического совещания с применением метода анализа иерархий и SWOT-анализа;
- проведение сетевого мозгового штурма в режиме телеконференции;
- решение обратной задачи на когнитивном графе с применением генетического алгоритма;

Разработан прототип оригинальной компоненты обеспечения виртуального сотрудничества, реализующий, например, формирование дескриптивных оценок «характера обсуждения темы совещания». Он позволяет ускорить процесс достижения согласия участников за счет предоставления информации, характеризующей отношение количества слов в активных и предметных предложениях, к общему количеству слов в сообщении.

Создан и экспериментально опробован макет системы верификации когнитивных моделей на основе анализа Больших Данных.

8. Заключение

В работе поставлена комплексная задача создания гетерогенной информационно-аналитической системы, имеющей развитые возможности анализа геопространственной и релевантной информации в сетевой экспертной среде. Для этого на методическом уровне сформирован специальный фрейм-интегратор, позволяющий обеспечить взаимодействия разнохарактерных информационно-аналитических методов и средств, включая сетевые экспертизы, когнитивное моделирование, виртуальное сотрудничество, визуализацию, анализ больших данных. Методологической основой интеграции является авторский подход конвергентного управления, создающий необходимые условия для целенаправленной сходимости групповых процессов принятия решений к нечетким целям.

При этом система виртуальной реальности обеспечивает «погружение» участников принятия решений в обстановку ситуации и интуитивный интерфейс для доступа к информации. Интеллектуальные способности системы обеспечивают ее адекватность для выполнения крупномасштабных облачных вычислений. Благодаря облачному интеграционному подходу, а также использованию гетерогенных вычислительных ресурсов становится достижимым решение актуальных задач, связанных с совершенствованием методов принятия групповых управленческих решений в чрезвычайных ситуациях, облегчением доступа к большим данных, возможностью быстрого исследования множественных альтернативных сценариев с вовлечением большего числа участников в процесс принятия решений, обеспечением эффективной коммуникации между ситуационным центром, населением.

К настоящему времени реализованы в виде специального программного обеспечения отдельные компоненты этом контексте авторам предстоит создать интерфейсы фрейм-интегратора между отдельными информационно-аналитическими средствами.

Работы по проекту проводятся в контексте государственных решений по импортозамещению и конкурентоспособности, акцентирования на критических инфраструктурах.

Список литературы

[1]. Ермаков А.Н., Меркулов А.А., Панфилов С.А., Райков А.Н. Поддержка решений в аварийных ситуациях на железной дороге с применением техник ситуационной осведомленности и виртуального экспертного сотрудничества // Сб. материалов Четвертой научно-практической конференции «Интеллектуальные системы на транспорте», 3-4 апреля 2014 г., Санкт-Петербург, С. 48-55.

[2]. Райков А.Н. Конвергентное управление и поддержка решений. -М.: Издательство ИКАР, 2009. – 245 с.

[3]. S.V.Ulyanov, A.N.Raikov. Chaotic factor in Intelligent Information Decision Support Systems. Edited by R.Aliev end ets.// Third International Conference on Application of Fuzzy Systems and Soft Computing (ICAFS'98). - Wiesbaden, Germany, October 5-7, - 1998. - P. 240 - 245

[4]. «Специальное программное обеспечение «Сетевая экспертно-аналитическая система «Архидока». Свидетельство о государственной регистрации программ № 2011613934 по заявке 2011612011 от 29 марта 2011 г. -М.: Роспатент. – 2011

[5]. Avdeeva Z., Kovriga S. Cognitive Approach in Simulation and Control Proceedings of the 17th World Congress The International Federation of Automatic Control, Seoul, Korea, July 6-11, 2008. pp. 1613-1620

[6]. Менеджмент знаний. Термины и определения. ГОСТ Р 53894-2010.

[7]. Менеджмент знаний. Руководство по обеспечению взаимосвязи менеджмента знаний с культурой организации и другими организационными процессами. ГОСТ Р 54876-2011.

[8]. Лепский В.Е. Эволюция представлений об управлении (методологический и философский анализ). – М.: Когито-Центр, 2015. – 107 с.

[9]. Бугаев А.С., Логинов Е.Л., Райков А.Н., Сараев В.Н. Латентный синтез решений // Экономические стратегии. – 2007. № 1, - С. 52 - 60.
[10]. Иванов В.К. Некорректные задачи в топологических пространствах// Сибирский математический журнал, X, № 5 (1969), - С. 1065 -1074.
[11]. Gubanov, D., Korgin, N., Novikov, D., Raikov, A. E-Expertise: Modern Collective Intelligence, Springer. Series: Studies in Computational Intelligence, Vol. 558, 2014, XVIII, 112 p.
[12]. CAVE2: http://www.mechdyne.com/cave2.aspx.

Cloud Framework for the Networked Expert and Analytical Tools Integration

[1]A. Ermakov <alexey-ermakov@yandex.ru>
[2]S. Klimenko <stanislav.klimenko@gmail.com>
[1]A. Merkulov <merkulov@cnsa.ru>
[1]S. Panfilov <serelen@list.ru>
[3]A.N. Raikov <alexander.n.raikov@gmail.com>
[1]Analytical Agency "New Strategies",
119526, Russia, Moscow, 26 Bakinskih Commissarov st., 14-100
[2]Institute of Computing for Physics and Technology,
142281, Russia, Moscow region, Protvino, Zavodski ave., 6
[3]Institute of Control Sciences RAS,
117997, Russia, Moscow, Profsoyuznaya st., 65

Abstract. Analytic decision support systems for public and for business applications usually have a distributed nature. We separate these systems into two distinct analytics support frames. The first frame is based on the processes of data consolidation, delivery and preprocessing. The second frame realizes decision support through the real time analysis of the expertize. The participants in the distributed decision-making system are immersed in the atmosphere of virtual reality. But the technological bridge between the distance separated participants of the decision-making process prevents the growth of the decisions quality, makes it difficult to achieve the participants agreement, especially when they are working in heterogeneous environments that require switching interfaces and software. The question is to accelerate the process of agreement achievement. It is proposed an approach that is based on the use of situational awareness, virtual collaboration and cloud computing techniques. It is proposed the new approach to make the cognitive model verification by using the analysis of Big Data. The algorithm is based on the assumption that the judgment of an interference between the factors of the cognitive model can be found in the texts of documents from Big Data. It is also provided the framework of knowledge management for the integration analytical tools in the cloud. The framework supports the targeted and sustained convergence of the decision-making processes. The basis for the structure of the framework is based on the author's method of inverse problems solving in topological spaces with the use of genetic algorithms. It is noted that the proposed approach was formed during the creating about 50 projects in the field of strategic analysis, information-analytical systems.

Keywords: visualization; verification; cognitive model; convergence; cloud; networked expertise; big data
DOI: 10.15514/ISPRAS-2015-27(6)-27
For citation: Ermakov A., Klimenko S., Merkulov A., Panfilov S., Raikov A.N. Cloud Framework for the Networked Expert and Analytical Tools Integration. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 421-440 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-27

References

[1]. Ermakov A.N., Mirkulov A.A., Panfilov S.A., Raikov A.N. Podderzhgka reshenij v avarijnikh situaciyakh na zhgeleznoi doroge s primeneniem tekhnik situacionnoy osvedomlennosti i virtualnogo expertnogo sotrudnichestva [Decision support in emergency situations on the railway using the techniques of situational awareness and virtual expert collaboration]. Sbornik materialov 4-th nauchno-prakticheskoi konferencii [Proceedings of the Fourth scientific and practical conference "Intelligent Transportation Systems"], April, 3-4, 2014, St. Petersburg, p. 48-55. (in Russian)

[2]. Raikov A.N. Konvergentnoe upravlenie i podderjka reshenij [Convergent management and decision support] M.: Izdatelstvo IKAR [publishing house IKAR], 2009. – 245 p. (in Russian).

[3]. S.V.Ulyanov, A.N.Raikov. Chaotic factor in Intelligent Information Decision Support Systems. Edited by R.Aliev end ets. Third International Conference on Application of Fuzzy Systems and Soft Computing (ICAFS'98). - Wiesbaden, Germany, October 5-7, - 1998. - P. 240 - 245

[4]. Specialnoe programmnoe obespechenie "Setevaya expertno-analiticheskaya sistema "ArkhiDoka" [Special software "Network expert-analytical system" Arhidoka]. Svidetelstvo o gosudarstvennoi registracii program [State registration certificate programs] № 2011613934, 29.03.2011. M.: Rospatent (in Russian).

[5]. Avdeeva Z., Kovriga S. Cognitive Approach in Simulation and Control Proceedings of the 17th World Congress The International Federation of Automatic Control, Seoul, Korea, July 6-11, 2008. pp. 1613-1620

[6]. Management znanij. Termini I opredeleniya [Management znanij. Terms and definitions]. GOST R 53894-2010. (in Russian).

[7]. Management znanij. Rukovodstvo po obespecheniju vzaimosvazi managementa znanij s kulturoj organizacii I drugimi organizacionnimi processami. [Knowledge management. Guide to securing correlation between knowledge management and the standard of organization and other organizational processes.]. GOST R 54876-2011 (in Russian).

[8]. Lepskiy V.E. Avoluciya predstavleniy ob upravlenii (metodologicheskiy I filosofskiy analiz) [The evolution of the ideas about the management (the methodological and philosophical analysis)]. M.: Kogito-Centr, 2015. – 107 p. (in Russian).

[9]. Bugaev A.S., Loginov E.L., Raikov A.N., Saraev V.N. The Semantics of Network contacts //Scientific and Technical Information Processing, 36(1), 2009. pp. 68-72..

[10]. Ivanov V.K. Nekorrektniye zadachi v topologicheskikh prostranstvah. Sibirskiy matematicheskiy zhgurnal [Ill-posed problems in topological spaces. Siberian Mathematical Journal, X, № 5 (1969), - C. 1065 -1074. (in Russian).

[11]. Gubanov, D., Korgin, N., Novikov, D., Raikov, A. E-Expertise: Modern Collective Intelligence, Springer. Series: Studies in Computational Intelligence, Vol. 558, 2014, XVIII, 112 p.

[12]. CAVE2: http://www.mechdyne.com/cave2.aspx.

Проверяющие эксперименты с ненаблюдаемым древовидными автоматами

Н.Г. Кушик <ngkushik@gmail.com>
Томский государственный университет,
634050, Россия, г. Томск, пр. Ленина, дом 36.
Телеком Южный Париж,
91000, Франция, г. Еври, ул. Ш. Фурье, дом 9.

Аннотация. В работе исследуются особенности синтеза безусловных проверяющих экспериментов для ненаблюдаемых автоматов. Данная задача активно используется при проверке функциональных и нефункциональных требований для различных дискретных и гибридных систем. В этом случае модель конечного автомата является подходящей, поскольку она позволяет адекватно описывать поведение систем с конечным непустым множеством состояний, конечными алфавитами входных и выходных символов, которые переходят из состояния в состояние при подаче входных воздействий и производят при этом выходные реакции. Проверяющие эксперименты, в свою очередь, позволяют проверить, находится ли предъявленный автомат в заданном отношении с другим автоматом. При синтезе проверяющих экспериментов для недетерминированных автоматов возможно исследование различных отношений конформности и различных способов задания множества неисправных автоматов, относительно которых собственно и строится эксперимент. В данной статье рассматривается модель исправности, в которой отношением конформности выступает отношение неразделимости, в качестве неисправностей исследуются (одиночные) ошибки переходов и выходов, и все неисправные автоматы перечисляются явно (модель белого ящика). Отношение неразделимости предполагает, что два инициальных автомата имеют хотя бы одну общую реакцию на каждую входную последовательность. В статье определяется специальный класс моделей неисправности, для которого проверяющий эксперимент, обнаруживающий любую неконформную реализацию, имеет полиномиальную длину. В частности, в работе исследуется специальный случай, когда эталонный недетерминированный (возможно, ненаблюдаемый) автомат имеет древовидную структуру, и показывается, что в этом случае можно построить кратный проверяющий эксперимент полиномиальной длины (относительно числа состояний автомата-спецификации).

Ключевые слова: конечный автомат, недетерминированный (ненаблюдаемый) автомат, древовидный автомат, проверяющий эксперимент

DOI: 10.15514/ISPRAS-2015-27(6)-28

Для цитирования: Кушик Н.Г. Проверяющие эксперименты с ненаблюдаемым древовидными автоматами. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 441-450. DOI: 10.15514/ISPRAS-2015-27(6)-28.

1. Введение

Одним из распространенных приложений различных задач анализа конечных автоматов является проверка функциональных и нефункциональных требований для различных дискретных и гибридных систем [1, 2]. В этом случае модель конечного автомата является подходящей, поскольку она позволяет адекватно описывать поведение систем с конечным непустым множеством состояний, конечными алфавитами входных и выходных символов, которые переходят из состояния в состояние при подаче входных воздействий и производят при этом выходные реакции. Проверяющие эксперименты позволяют проверить, находится ли предъявленный автомат в заданном отношении с другим автоматом, и при построении таких экспериментов активно используются «умозрительные» эксперименты [3], позволяющие идентифицировать состояние автомата до или после эксперимента с эти автоматом. Известные точные (не эвристические) методы решения синтеза «умозрительных» экспериментов с автоматами, как правило, имеют очень высокую сложность даже для полностью определенных детерминированных автоматов [4, 5]. Вместе с тем, современные технические системы, для которых строятся проверяющие эксперименты, как правило, описываются недетерминированными автоматами, и поэтому сложность синтеза таких экспериментов только повышается.

В данной работе исследуется синтез проверяющих экспериментов для ненаблюдаемых недетерминированных автоматов. Рассматривается модель исправности, в которой отношением конформности выступает отношение неразделимости, в качестве неисправностей исследуются (одиночные) ошибки переходов и выходов, и все неисправные автоматы перечисляются явно (модель белого ящика). Отношение неразделимости предполагает, что два инициальных автомата имеют хотя бы одну общую реакцию на каждую входную последовательность. Исследуется специальный случай, когда эталонный ненаблюдаемый недетерминированный автомат имеет древовидную структуру, и показывается, что в этом случае можно построить кратный проверяющий эксперимент полиномиальной длины (относительно числа состояний автомата-спецификации).

Структура работы следующая. В втором разделе вводятся основные определения и обозначения, используемые в статье. Третий раздел посвящен исследованию синтеза проверяющих экспериментов для ненаблюдаемых древовидных спецификаций. Четвертый раздел включает иллюстративный пример, и последний – пятый – содержит заключение работы.

2. Основные определения и обозначения

Под *конечным автоматом* (или просто *автоматом*) [6] понимается пятерка $\mathbf{S} = (S, I, O, h_S, S_{in})$, где S – конечное непустое множество состояний с выделенным непустым множеством начальных состояний S_{in}, I и O – конечные непустые входной и выходной алфавиты соответственно, такие, что $I \cap O = \varnothing$ и $h_S \subseteq S \times I \times O \times S$ – *отношение* или *множество переходов*. Автомат называется *инициальным*, если S_{in} – одноэлементное множество, т.е. $|S_{in}| = 1$, в противном случае автомат называется слабо *инициальным*. Автомат называется *неинициальным*, если $S_{in} = S$. Четверка $(s, i, o, s') \in h_S$ называется *переходом* в автомате в состоянии s или переходом из состояния s в состояние s'. Если в автомате **S** для любой пары $(s, i) \in S \times I$ существует, по крайней мере, одна пара $(o, s') \in O \times S$ такая, что $(s, i, o, s') \in h_S$, то автомат называется *полностью определенным*, в противном случае автомат называется *частично определенным* или *частичным*. В полностью определенном автомате из каждого состояния существует переход под действием любого входного символа. В частичном автомате из некоторого состояния может не быть ни одного перехода под действием некоторого входного символа. Автомат **S** называется *детерминированным*, если для любой пары $(s, i) \in S \times I$ существует не более одной пары $(o, s') \in O \times S$ такой, что $(s, i, o, s') \in h_S$. В противном случае автомат называется *недетерминированным*. Если в недетерминированном автомате **S** для любой тройки $(s, i, o) \in S \times I \times O$ существует не более одного состояния $s' \in S$ такого, что $(s, i, o, s') \in h_S$, то автомат называется *наблюдаемым*, в противном случае автомат называется *ненаблюдаемым*. В данной статье рассматриваются недетерминированные, возможно, ненаблюдаемые автоматы, диаграммы переходов которых имеют форму дерева (далее – *древовидные* автоматы).

Вместе с отношением переходов в автомате часто используют две функции: функцию *переходов* (функцию $next\_state_S$) и функцию *выходов* (функцию out_S), определенных следующим образом $next\_state_S(s, i) \ni s'$, если $\exists\, o \in O, (s, i, o, s') \in h_S$; $out_S(s, i) \ni o$, если $\exists\, s' \in S, (s, i, o, s') \in h_S$.

Для детерминированных полностью определенных автоматов функции $next\_state_S$ и out_S однозначно определяют поведение (отношение переходов) автомата. Недетерминированные автоматы с различным поведением могут иметь одни и те же функции выходов и переходов $next\_state_S$ и out_S [6].

Отношение переходов обычным образом распространяется на последовательности (слова) в алфавитах I и O. Обозначим через I^* множество всех последовательностей конечной длины в алфавите I, включая пустую последовательность ε. Как обычно, через I^m мы обозначаем множество всех последовательностей из I^* длины m. Соответственно, функции переходов $next\_state_S$ и выходов out_S можно распространить на последовательности входных и выходных символов. В этом случае множество $next\_state_S(s, \alpha)$ включает те и только те состояния, которые достижимы в автомате S из

состояния s по входной последовательности α, т.е. $next\_state_S(s, \alpha)$ есть α-преемник состояния s. Соответственно, множество $out_S(s, \alpha)$ включает все возможные выходные последовательности (реакции) автомата S в состоянии s на входную последовательность α.

Под *экспериментом* с автоматом [3] понимается подача некоторой входной (-ых) последовательности (-ей) на вход автомата, предъявленного к эксперименту, наблюдение выходной (-ых) реакции (-ий) на эту (эти) последовательность (-ти) и формирование заключения о некоторых свойствах автомата. В работе исследуется возможность синтеза проверяющих экспериментов для недетерминированных, возможно, ненаблюдаемых автоматов. При проведении *проверяющего* эксперимента рассматривается некоторый класс автоматов, т.е. предполагается, что имеется некоторое множество автоматов, и один из этих автоматов предъявлен к эксперименту (*тестируемый* автомат или *реализация*). Требуется установить, находится ли тестируемый автомат в определенном отношении с некоторым другим автоматом, который называется *эталонным* автоматом или *автоматом-спецификацией*. Отношение соответствия в данном случае называют также отношением *конформности*.

Синтез проверяющих экспериментов проводится на основе построения различающих экспериментов или различающих / диагностических (*разделяющих* для недетерминированных автоматов [7]) последовательностей, позволяющих идентифицировать начальное состояние автомата до эксперимента. Проверка существования различающего эксперимента и его последующий синтез для недетерминированного (полностью определенного) автомата сводится к проверке того факта, что множество начальных состояний этого автомата является *разделимым*. Иными словами, необходимо проверить, что существует последовательность α, которая является *разделяющей последовательностью* для любых двух различных состояний s_1 и s_2 множества S_{in}, т.е. $\forall\ s_1, s_2 \in S_{in}$ имеет место $out(s_1, \alpha) \cap out(s_2, \alpha) = \varnothing$.

Таким образом, к проверяющему эксперименту предъявлен автомат-спецификация **S**, класс автоматов с теми же входным и выходным алфавитами, как автомат **S**, который обозначается *FD* (*Fault domain*) и называется множеством «тестируемых» автоматов (мутантов) или областью неисправности, и отношение конформности $\approx$. Ставится задача синтезировать такой эксперимент (такую последовательность или конечное множество входо-входных последовательностей), что всякий автомат M из множества *FD*, который находится в отношении $\sim$ со спецификацией **S** выдает «ожидаемые» (-ую) выходные (-ую) реакции (-ию) на входные (-ую) последовательности (-ность) эксперимент. С другой стороны, всякий автомат M', который не является конформным спецификации **S** в соответствии с введенным отношением $\approx$, должен быть отличим от **S**, т.е. выходные (-ая) реакции (-ия) M' не должны (-а) ожидаемыми (-ой), по крайней мере, для одной входной последовательности эксперимента. В этом случае говорят о *полном*

проверяющем эксперименте (тесте) относительно модели неисправности <**S**, $\approx$, *FD*>. В данной работе автомат **S** является недетерминированным и, возможно, ненаблюдаемым, однако с древовидной диаграммой переходов. Отношение $\approx$ есть отношение неразделимости. Все автоматы-мутанты перечисляются явно, каждый из которых есть результат внесения в автомат **S** ошибки перехода или выхода. Под *ошибкой выхода* понимают изменение перехода вида $(s, i, o, s') \in h_S$ на переход вида $(s, i, o', s') \in h_S$ для $o \neq o'$. Под *ошибкой перехода*, в свою очередь, понимают изменение состояния-преемника по некоторой входо-выходной паре, т.е. замену перехода вида $(s, i, o, s') \in h_S$ на переход вида $(s, i, o, s'') \in h_S$ для $s' \neq s''$. В данной работе мы рассматриваем одиночные ошибки переходов и выходов, однако отмечаем, что приведенные результаты естественным образом распространяются на случай кратных неисправностей.

3. Синтез проверяющих экспериментов для древовидных спецификаций

Пусть **S** недетерминированный (возможно, ненаблюдаемый) инициальный древовидный автомат $\mathbf{S} = (S, I, O, h_S, \{s_0\})$, в котором в каждом нетупиковом состоянии определены переходы по каждому входному символу. Построим кратный проверяющий эксперимент *TS* относительно модели неисправности <**S**, $\approx$, *FD*> и оценим максимальную длину каждой разделяющей последовательности, входящей в эксперимент *TS*. Отметим, что для диаграммы переходов автомата **S** все ветви имеют одинаковую длину. Корнем соответствующего дерева выступает узел, помеченный состоянием s_0. Рассмотрим некоторый мутант $M \in FD$ автомата **S**, который получается внесением одиночной ошибки перехода или выхода. Следует отметить, что если *M* был получен внесением ошибки перехода, то диаграмма переходов данного автомата не может иметь форму дерева, поскольку внесенная ошибка перехода обеспечивает присутствие двух входящих дуг в некотором узле дерева.

Как отмечалось выше, синтез проверяющего эксперимента для данной модели неисправности может проводиться на основе синтеза соответствующих различающих последовательностей для эталонного автомата **S** и всякого его мутанта *M*. В предположении, что и эталонный автомат, и его реализация является инициальными автоматами, синтез соответствующей разделяющей последовательности для пары автоматов **S** и *M* сводится к синтезу таковой для прямой суммы этих автоматов $M \oplus \mathbf{S}$ [8].

Таким образом, последовательная генерация автоматов-мутантов *M* для автомата-спецификации **S** и синтез соответствующих разделяющих последовательностей позволяют построить проверяющий тест относительно модели неисправности <**S**, $\approx$, *FD*>, в которой автомат **S** является инициальным полностью определенным недетерминированным, возможно,

ненаблюдаемым автоматом. Отметим еще раз, что в этом случае рассматривается модель белого ящика, т.е. все мутанты $M \in FD$ перечисляются явно. С другой стороны, не всякий мутант M может оказаться отличимым от спецификации **S**, в этом случае данный мутант не будет обнаружен кратным проверяющим экспериментом.

Поскольку эталонный автомат является недетерминированным и, возможно, ненаблюдаемым, длина разделяющей последовательности для прямой суммы $M \oplus \mathbf{S}$ экспоненциально зависит от числа состояний этой прямой суммы. С другой стороны, если длина каждой входо-выходной последовательности в автомате **S** полиномиально зависит от числа n состояний данного автомата, то и максимальная длина разделяющей последовательности для прямой суммы $M \oplus \mathbf{S}$ также полиномиально зависит от числа n состояний автомата-спецификации, $|S| = n$.

Имеет место следующее

Утверждение Для недетерминированного (возможно, ненаблюдаемого) древовидного автомата $S = (S, I, O, h_S, \{s_0\})$, $|S| = n$, $|I| > 1$, у которого в каждом нетупиковом состоянии определены переходы по всем входным символам, длина каждой непустой трассы не превосходит величины $\log n$.

Доказательство данного утверждения опирается на тот факт, что дерево, представляющее граф переходов автомата **S**, является полным d-арным для $|I| \leq d$ с числом узлов, равным n. Соответственно, высота данного дерева не превосходит логарифма вида $\log_{|I|} n$. Поскольку входной алфавит древовидного автомата **S** является двух-буквенным (или выше), длина каждой непустой входо-выходной последовательности автомата **S** ограничена величиной $\log n$.

Отметим, что чем выше степень каждой из вершин дерева, представляющего диаграмму переходов автомата **S**, тем меньше его высота. Следовательно, длина входо-выходной последовательности в данном автомате тем короче, чем выше степень недетерминизма и ненаблюдаемости данного автомата, поскольку увеличение степени недетерминизма/ненаблюдаемости в некотором состоянии автомата повышает степень соответствующего узла в его графе переходов. Таким образом, трассы древовидного автомата тем короче, чем сложнее его структура.

Как отмечалось ранее, при синтезе проверяющего эксперимента относительно модели неисправности $<\mathbf{S}, \approx, FD>$ строятся разделяющие последовательности для прямой суммы вида $M \oplus \mathbf{S}$. Вместе с тем, максимальная длина трассы в автомате-спецификации ограничена величиной $\log n$. Таким образом, если для древовидного недетерминированного (возможно, ненаблюдаемого) автомата $\mathbf{S} = (S, I, O, h_S, \{s_0\})$, $|S| = n$, в котором в каждом нетупиковом состоянии определены переходы по всем входным символам, и его мутанта $M \in FD$ существует разделяющая последовательность α, то $|\alpha| \leq \log n$.

Следовательно, если число мутантов, включаемых в множество FD, полиномиально зависит от числа состояний n эталонного автомата **S**, то общая длина кратного проверяющего эксперимента относительно отношения неразделимости $\approx$ также зависит полиномиально относительно данной величины $n = |S|$.

Отметим также, что в случае древовидных автоматов генерация мутантов и синтез соответствующих разделяющих последовательностей могут производиться одновременно. Иными словами, два этих действия могут быть совмещены в одно, за счет одновременного синтеза автомата-мутанта и входо-выходной последовательности, которая покрывает неисправный переход автомата, предъявленного к эксперименту. Это может быть сделано, например, следующим образом. После внесения ошибки перехода или выхода в автомат **S** строится входо-выходная последовательность γ, которая переводит автомат S из начального состояния в состояние s', в случае внесения ошибки выхода (замена перехода вида $(s, i, o, s') \in h_S$ на переход вида $(s, i, o', s') \in h_S$ для $o \neq o'$) или ошибки перехода (замена перехода вида $(s, i, o, s') \in h_S$ на переход вида $(s, i, o, s'') \in h_S$ для $s' \neq s''$). Договоримся, что далее для обозначения состояний автомата-мутанта M будем использовать алфавит M, и состояние s_i автомата **S** соответствует состоянию m_i автомата M. Пусть последовательность γ имеет вид $\gamma = \alpha/\beta$, тогда последовательность α может выступать преамбулой синтезируемой разделяющей последовательности для прямой суммы $M \oplus \mathbf{S}$. Далее эта последовательность α продляется (если это возможно) постамбулой δ, различающей множества состояний S' и M', являющихся α-преемниками начальных состояний s_0 и m_0 автоматов **S** и M, соответственно. Заметим, что в случае, если была внесена ошибка выхода, то между множествами состояний S' и M' существует взаимно однозначное соответствие. В случае, если была внесена ошибка перехода, множество M' содержит «ошибочное» состояние m'', и соответствующая постамбула должна различать состояния s' и m'' автоматов S и M. Синтез такой последовательности можно произвести явным моделированием поведения обоих автоматов в состояниях из множеств S' и M' или же с использованием методов синтеза разделяющих последовательностей для (ациклических) недетерминированных автоматов. Можно рассчитывать, что чем дальше от корня находится ошибочный переход, тем короче постамбула δ (если существует). Мы отмечаем, что эффективность предлагаемого упрощения (эвристики) синтеза разделяющей последовательности для эталонного автомата **S** и его мутанта-реализации M необходимо оценивать экспериментально, однако в данной работе такие эксперименты не проводились – они представляют интерес для дальнейших научных исследований.

4. Пример

В качестве примера рассмотрим инициальный автомат **S** = ({s_0, s_1, …, s_{25}}, {i_1, i_2}, {o_1, o_2}, h_S, {s_0}), приведенный на рисунке 1. Отметим, что в каждом состоянии автомата **S** определены переходы по входным символам i_1 и i_2.

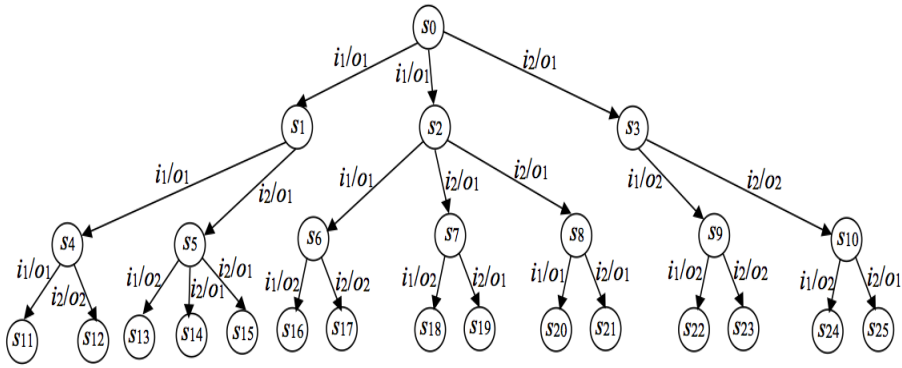


*Рис. 1. Полностью определенный древовидный автомат **S**.*

На следующем рисунке (рис. 2) приведен мутант M автомата **S**, который получается посредством внесения одиночной ошибки перехода на первом уровне дерева, представляющего диаграмму переходов автомата **S**. В частности, мутант M, который получается заменой перехода вида (s_0, i_2, o_1, s_3) $\in h_S$ на переход вида (s_0, i_2, o_1, s_1) $\in h_S$. Отметим еще раз, что состояния автомата-мутанта M переобозначены как элементы множества M = {m_0, m_1, …, m_{25}}.

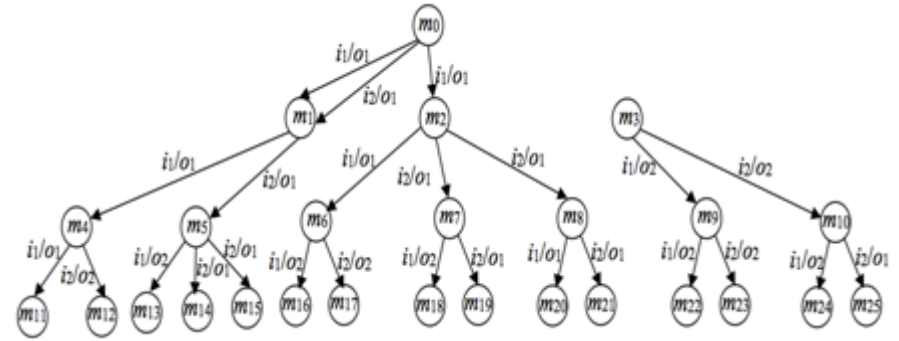


*Рис. 2. Мутант M автомата **S**.*

Поскольку ошибка перехода внесена на первом ярусе дерева, представляющего диаграмму переходов эталонного автомата **S**, преамбула (входная последовательность) α, покрывающая неисправный переход, имеет длину один, а именно, $\alpha = i_2$. Далее эту преамбулу можно продлить последовательностью δ такой же длины, в частности, можно выбрать $\delta = i_2$. Действительно, после подачи последовательности α необходимо различить одноэлементные множества состояний $S' = \{s_3\}$ и $M' = \{m_1\}$. Нетрудно видеть, что для состояний s_3 и m_1 входной символ i_2 представляет собой разделяющую последовательность. Таким образом, входная последовательность вида i_2i_2 есть различающая последовательность для ненаблюдаемого связного древовидного автомата **S** и его мутанта M. Если при проведении проверяющего эксперимента при подаче входной последовательности i_2i_2 предъявленный автомат выдает выходную реакцию o_1o_1, то предъявленный автомат представляет собой неисправную реализацию M.

Следует отметить, однако, что в данном примере разделяющая последовательность оказалась короче любой трассы, определенной в автомате **S**, несмотря на то, что неисправный переход в автомате M' располагается достаточно близко к корню. Этот факт еще раз подтверждает необходимость проведения дальнейших экспериментальных исследований по установлению максимально достижимой/средней длины разделяющих последовательностей, формирующих кратных проверяющий эксперимент для модели неисправности <**S**, $\approx$, *FD*>, где **S** – инициальный древовидный недетерминированный (возможно, ненаблюдаемый автомат), и множество *FD* включает автоматы-мутанты первого порядка, перечисленные явно.

5. Заключение

В статье рассмотрена задача синтеза кратных проверяющих экспериментов для недетерминированных (возможно, ненаблюдаемых) автоматов. Показано, что если эталонный автомат **S** является древовидным, таким, что в каждом его нетупиковом состоянии определены переходы по всем входным символам, то можно построить проверяющий тест относительно модели неисправности <**S**, $\approx$, *FD*>, который имеет полиномиальную длину и обнаруживает всякую неконформную реализацию.

В статье также предложена эвристика для синтеза соответствующих разделяющих последовательностей для эталонного автомата и его мутантов, формирующих проверяющий эксперимент. Одним из будущих направлений работы является оценка эффективности предложенного упрощения метода синтеза проверяющего эксперимента для реальных технических систем.

С другой стороны, интерес представляет исследование других «хороших» классов автоматов с полиномиальными оценками длин проверяющих экспериментов. Перечисленные задачи формируют перспективы для дальнейших научных исследований.

Список литературы

[1]. М.П. Василевский. О распознавании неисправности автоматов. Кибернетика, № 4, 1973 г. стр. 98-108.
[2]. F. Hennie. Fault-Detecting Experiments for Sequential Circuits. Proc. Fifth Ann. Symp. Switching Circuit Theory and Logical Design, 1964. P. 95-110.
[3]. E. Moore. Gedanken-experiments on sequential machines Automata Studies, Annals of Mathematical Studies, No.1, 1956. P. 129-153.
[4]. D. Lee M. Yannakakis. Testing Finite-State Machines: State Identification and Verification. IEEE Transactions on Computers, 1994, Volume 43, Issue 3. P. 306-320.
[5]. S. Sandberg. Homing and Synchronization Sequences. Lecture Notes in Computer Science, № 3472, 2005. P. 5-33.
[6]. Н.В. Евтушенко, А.Ф. Петренко, М.В. Ветрова. Недетерминированные автоматы: анализ и синтез Ч. 1: Отношения и операции : учеб. пособие. Томск : Том. гос. ун-т, 2006. 142 с.
[7]. N. Spitsyna, K. El-Fakih, N. Yevtushenko. Studying the separability relation between finite state machines. Softw. Test., Verif. Reliab., 2007, Volume 17, Issue 4. P. 227-241.
[8]. N. Kushik, N. Yevtushenko, A. Cavalli. On Testing against Partial Non-observable Specifications. Proceedings of the 9th International Conference on the Quality of Information and Communications Technology, 2014. P. 230-233.

Checking Experiments with Non-Observable Tree FSMs

N. Kushik <ngkushik@gmail.com>
Tomsk State University,
36 Lenin ave., Tomsk, 634050, Russian Federation
Telecom SudParis,
9 rue Charles Fourier, Evry, 91000, France

Abstract. The paper addresses the problem of deriving preset checking experiments for non-observable Finite State Machines (FSMs). This problem often appears when checking functional and non-functional requirements of various discrete and hybrid systems. In this case, the model of an FSM is adequate as it allows to formally describe systems with finite non-empty sets of states and finite sets of inputs and outputs that move from one state to another when an input is applied and an output is produced. Checking experiments, on the other hand, allow to verify if a given FSM conforms to another one or not. When deriving checking experiments for nondeterministic FSMs, different conformance relations can be considered as well as different ways to define the set of faulty implementations which the experiment aims to detect. In this paper, we consider a non-separability relation, and the fault model is considered to be a 'white box' where all possible implementations are explicitly enumerated. Faulty implementations can contain first order transfer and output faults. Non-separability relation requires that two initialized machines have at least one common output reaction to any input sequence. In the paper, we define a specific class of fault models such that the checking experiment that detects each implementation not conforming to the specification has polynomial length. In particular, we show that if the specification FSM has

a tree structure then it is possible to derive a complete checking experiment that has polynomial length.

Keywords: finite state machine, FSM, non-deterministic (non-observable) FSM, tree FSM experiment, checking experiment

DOI: 10.15514/ISPRAS-2015-27(6)-28

For citation: Kushik N. Checking Experiments with Non-Observable Tree FSMs. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 441-450 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-28.

References

[1]. M.P. Vasilevskij. O raspoznavanii neispravnosti avtomatov [Failure diagnosis of automata]. Kibernetika [Cybernetics], № 4, 1973, pp. 98-108 (in Russian).
[2]. F. Hennie. Fault-Detecting Experiments for Sequential Circuits. Proc. Fifth Ann. Symp. Switching Circuit Theory and Logical Design, 1964. P. 95-110.
[3]. E. Moore. Gedanken-experiments on sequential machines Automata Studies, Annals of Mathematical Studies, No.1, 1956. P. 129-153.
[4]. D. Lee M. Yannakakis. Testing Finite-State Machines: State Identification and Verification. IEEE Transactions on Computers, 1994, Volume 43, Issue 3. P. 306-320.
[5]. S. Sandberg. Homing and Synchronization Sequences. Lecture Notes in Computer Science, № 3472, 2005. P. 5-33.
[6]. N.V. Evtushenko, A.F. Petrenko, M.V. Vetrova. Nedeterminirovannye avtomaty: analiz i sintez Ch. 1: Otnoshenija i operacii : ucheb. posobie [Nondeterministic FSMs : analysis and synthesis, Chapter 1 : relations and operations : course support book]. Tomsk : Tom. gos. un-t, 2006. 142 p. (in Russian)
[7]. N. Spitsyna, K. El-Fakih, N. Yevtushenko. Studying the separability relation between finite state machines. Softw. Test., Verif. Reliab., 2007, Volume 17, Issue 4. P. 227-241.
[8]. N. Kushik, N. Yevtushenko, A. Cavalli. On Testing against Partial Non-observable Specifications. Proceedings of the 9th International Conference on the Quality of Information and Communications Technology, 2014. P. 230-233.

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.
Телефон: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Сайт: http://www.ispras.ru/proceedings/

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Tel: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Web: http://www.ispras.ru/en/proceedings/