МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М. В. ЛОМОНОСОВА

Факультет вычислительной математики и кибернетики

ПРОГРАММНЫЕ СИСТЕМЫ И ИНСТРУМЕНТЫ

Тематический сборник

№ 21

Под общей редакцией
чл.-корр. РАН, профессора Р. Л. Смелянского

МАКС ПРЕСС

МОСКВА — 2021

УДК 519.6+517.958
ББК 22.19
П75

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики Московского государственного университета имени М. В. Ломоносова

Программные системы и инструменты : Тематический сборник / П75 Под ред. Р. Л. Смелянского.— Москва : Издательский отдел факультета ВМК МГУ имени М. В. Ломоносова (лицензия ИД № 05899 от 24.09. 2001 г.); МАКС Пресс, 2021.— № 21.— 164 с.

ISBN 978-5-89407-625-5 (ВМК МГУ имени М. В. Ломоносова)
ISBN 978-5-317-06733-5 (МАКС Пресс)

Данный выпуск сборника составлен по материалам работ студентов и аспирантов, получивших рекомендации научных семинаров кафедры автоматизации систем вычислительных комплексов и кафедры алгоритмических языков. Редколлегия сборника продолжает традицию его издания в память о первом руководителе кафедры АСВК Л. Н. Королёве. В предлагаемом читателю тематическом сборнике публикуются статьи, посвященные проблемам современных компьютерных сетей, методам управления облачными вычислениями, инструментальным средствам сбора медицинской телеметрии, архитектуре сетевого процессорного устройства, обучению и использованию нейросетей и транспиляции программ.

Статьи сборника будут интересны студентам, аспирантам и специалистам в области разработки прикладных программных систем.

Ключевые слова: информационно-телекоммуникационные технологии, программно-конфигурируемые сети, построение расписаний, облачные вычисления, облачная среда, генетические алгоритмы, эволюционные алгоритмы, платформа для медицинских исследований, Интернет вещей, виртуальные ресурсы, виртуализация сетевых функций, облачная платформа, транспиляция программ, функциональные языки программирования, сетевое процессорное устройство, синхронизация состояния, нейронные сети, OpenFlow-контроллер, прогнозирование нагрузки, таблицы классификации, сжатие данных, отказ в обслуживании, балансировка нагрузки, оценка времени отклика, интегрированная модульная авионика, федеративное обучение, нейронные сети, простое голосование, классификатор.

УДК 519.6+517.958
ББК 22.19

ISBN 978-5-89407-625-5
ISBN 978-5-317-06733-5

Оглавление

СБОРНИК
«Программные системы и инструменты»

От редколлегии:

Тематический сборник “Программные системы и инструменты” был инициирован Львом Николаевичем Королевым в 2000 году, как площадка где студенты и молодые ученые могли бы публиковать свои достижения. Все эти годы Лев Николаевич неустанно вел его, отбирал публикации, редактировал. Редколлегия сборника продолжает эту традицию в память об этом выдающимся человеке.

Этот выпуск сборника составлен по материалам работ студентов, получивших рекомендации научных семинаров кафедры Автоматизации Систем Вычислительных Комплексов, которую Л.Н. Королев создал и бессменно руководил, а также кафедры Алгоритмических Языков.

Редколлегия

Абрамов А. В., Чупахин А. А.

Построение расписания выполнения работ в гетерогенной облачной вычислительной среде

Введение

Эффективность использования вычислительных систем высокой производительности, таких как вычислительные кластеры или распределенные гетерогенные вычислительные системы (РГВС), существенно зависит от методов планирования выполнения работ. Планирование вычислительного процесса заключается в распределении времени процессоров между работами пользователей, с целью уменьшения общего времени их выполнения. Частью указанного процесса является решение задачи составления расписаний. В общем случае задачи составления расписаний являются NP-трудными. Это означает, что для решения сложных задач составления расписаний быстрый (за полиномиальное время) и точный алгоритм не может быть найден, однако могут быть разработаны эффективные алгоритмы нахождения решения с ослаблением некоторых требований в постановке задачи.

В данной работе рассматривается проблема планирования вычислений в ГВС, в которой каждый вычислитель обладает определенным ресурсом, а для каждой поступающей в очередь планирования работы задана оценка времени выполнения и необходимый для ее выполнения ресурс. В качестве ресурса рассматривается количество ядер на каждом из вычислителей. Работы, попадая в очередь, планируются не сразу, а собираются в пакеты [1] размера d, после чего такие пакеты посылаются на вход планировщику, который строит для них расписание выполнения работ. Суммарное время пребывания работы в ВС определяется как разность между временем конца выполнения работы и временем ее попадания в очередь. Необходимо минимизировать суммарное время пребывания всех работ в ВС.

1. Задача планирования вычислений в ГВС

Пусть дано N вычислителей $C_i, 1 \leq i \leq N$, представляющих из себя тройки $C_i = \{H_i, T\_lim_i, Q_i\}$, где

1. H_i – ресурс, которым обладает вычислитель i (количество ядер);
2. T_lim_i – жёсткий лимит, при превышении которого работа снимается с выполнения на вычислителе C_i;
3. Q_i – очередь работ, назначенных на вычислитель C_i.

Множество независимых работ $J = \{J_1, J_2, \ldots, J_K\}$, где каждой работе однозначно ставится в соответствие четвёрка $\{Name_i, Arg_i, R_i, submit_i\}$, где

1. $Name_i$ – имя работы;
2. Arg_i – аргументы, указываемые при запуске работы на исполнение;
3. R_i – количество потребляемого работой ресурса (количество ядер);
4. $submit_i$ – время поступления работы в вычислительную среду.

Пусть задана функция $predtime(Name_j, Arg_j, R_j, C_i)$, которая на вход получает имя работы $Name_j$, аргументы Arg_j, количество потребляемого работой ресурса R_j и вычислитель C_i, на котором работа будет выполняться, а на выход выдает верхнюю оценку времени выполнения t_m этой работы на данном вычислителе. Предполагается, что все работы из множества J уже поступали в вычислительную среду, поэтому для каждой из них имеется оценка времени выполнения. Также предполагается, что каждая работа из множества J остаётся неизменной в течение истории её запусков и обрабатывается на вычислителе без прерываний.

Назначение работы $J_m \ \forall m, 1 \leq m \leq K$, на вычислитель $C_i \forall i, 1 \leq i \leq N$, определяется следующим образом: работе $J_m \forall m, 1 \leq m \leq K$, сопоставляем вычислитель $C_i \forall i, 1 \leq i \leq N$, на котором данная работа будет выполняться таким образом, чтобы количество ресурса, потребляемого работами, назначенными на данный вычислитель, в каждый момент времени не превышало

количество ресурса данного вычислителя H_i, то есть если положить $\{J_{m,i}\} = \{J_{m_1}, \ldots, J_{m_p}\}$ – множеством работ, выполняющихся в данный момент на вычислителе C_i, то

$$\sum_{k:J_{m_k} \in J_{m,i}} R_{m_k} \leq H_i \tag{1}$$

Все работы поступают в общую очередь, назовём её буфером. Попав в буфер, работа планируется не сразу: поступившие работы объединяются в пакеты размера d (количество работ), после чего планировщик распределяет работы из пакета между вычислителями. Сразу после планирования работ начинает формироваться следующий пакет работ из общей очереди. На случай, если работ в буфер поступило меньше, чем d, существует лимит времени пакетизации – *packlimit*. Если в течение времени пакетизации на центральный вычислитель не поступило d работ, то пакет формируется из имеющихся.

Расписание выполнения работ из множества $\{J\}$ определено, если заданы:

1. множество вычислителей $\{C\}$;
2. привязка работ к вычислителям.

Привязка – всюду определенная на множестве работ функция, которая задает распределение работ по вычислителям. Назовём расписание корректным, если оно удовлетворяет условию (1).

Удобно рассматривать пятёрку $\{J_m, t\_real_m, q_m, S_m, E_m\}, 1 \leq m \leq K$, где J_m – работа, t_real_m – реальное время выполнения данной работы на вычислителе, q_m – время, проведённое работой в очереди вычислителя, на который она была назначена, S_m – время начала выполнения, E_m – время конца выполнения, причём $E_m = \min(S_m + t_m, S_m + t\_real_m))$. Обозначим такую пятёрку G_m. Отметим, что при построении расписания используется верхняя оценка времени выполнения, которую предоставляет функция *predtime*, поэтому время конца выполнения рассматривается также оценочное, то есть $S_m + t_m$, где $t_m = predtime(Name_m, Arg_m, R_m, C_i)$. Если по каким-либо причинам время выполнения работы t_real_m на вычислителе C_i превысило время t_m, то она продолжает выполняться до исчерпания жесткого лимита T_lim_i, заданного для каждого вычислителя индивидуально. Если превышено время T_lim_i, то система снимает работу с выполнения.

Введем функцию $Opt(C, G)$, определяющую общее время нахождения всех работ в вычислительной среде, следующим образом:

$$Opt(C, G) = \sum_{m=1}^{K} (t_m + (S_m - submit_m)) \quad (2)$$

Необходимо построить расписание, удовлетворяющее условию 1 и минимизирующее функцию $Opt(C, G)$.

2. Исследование предметной области

При выборе алгоритма в первую очередь стоит обратить внимание на то, что решаемая задача является NP-трудной. Каждая работа потребляет определенное количество ресурсов. Также стоит отметить, что имеется оценка времени выполнения каждой работы на каждом из вычислителей, причём работы выполняются без прерываний. Миграция работ с одного вычислителя на другой невозможна. Работы независимы. Вычислительная среда неоднородна и каждый вычислитель обладает своим объёмом ресурсов. Наша задача минимизировать суммарное время пребывания всех работ в вычислительной среде. Рассмотрим теперь работы с похожей тематикой.

В статье [2] рассматривается алгоритм ветвей и границ и предлагается новый метод оценки частичного решения, который может применяться с любым генератором допустимых расписаний, и ряд новых способов сокращения перебора. В [3, 4, 5] рассматривается генетический алгоритм в задаче построения расписаний для распределённых систем, представлен способ учёта ресурсных ограничений. В работе [6] рассматривается задача построение ВС, использующих наименьшее число аппаратных ресурсов для выполнения прикладной программы за время, не превышающее заданного. В [7] представлен гибридный алгоритм, составленный на основе генетического алгоритма и алгоритма имитации отжига, для решения задач построения расписаний в системах с ограниченными ресурсами.

После обзора методов построения расписаний выполнения работ было решено выбрать для поставленной задачи генетический алгоритм (ГА), т.к. он обладает следующими преимуществами по сравнению с другими методами:

1. Скорость сходимости ГА, в отличие от скорости сходимости алгоритмов случайного поиска и имитации отжига, не зависит

от сложности ландшафта целевой функции (функция может иметь множество оптимумов) и размерности пространства решений.

2. Используя ГА при планировании объёмных пакетов, можно избежать трудоемких вычислений (при планировании объёмных пакетов), чего не всегда можно сделать, используя жадные алгоритмы.

3. Описание разработанного алгоритма

3.1 Основные понятия

В ГА (эволюционных алгоритмах, ЭА) каждое расписание должно быть представлено в закодированной форме, в связи с чем вводятся понятия гена, хромосомы и популяции. Обычно гены представляют собой сами работы, хромосома – одно из возможным расписаний, а популяция – множество хромосом, т.е. множество расписаний. Определим данные понятия для нашей задачи.

Пусть геномом будет пятерка:

$$\{C_i, a_k(C_i), b_k(C_i), pid_k, cores_k\} \tag{3}$$

где C_i – вычислитель, $1 \leq i \leq N$, $a_k(C_i), b_k(C_i)$ – время начала и конца выполнения работы J_k на вычислителе C_i, pid_k – идентификационный номер работы J_k, $cores_k$ – количество ядер, необходимое для выполнения работы J_k на вычислителе C_i.

Тогда хромосомой может служить множество групп указанных пятерок. Каждая группа относится к одному из вычислителей, то есть работы, соответствующие генам, расположенным в группе k, будут выполняться на вычислителе C_k. В каждой группе гены расположены последовательно в порядке выполнения соответствующих работ. При таком подходе гены, относящиеся к отдельным вычислителям, хранятся непрерывно. Сохранение непрерывного расположения генов дает возможность генетическому оператору скрещивания копировать в дочернюю хромосому за раз непрерывные участки оперативной памяти родительских хромосом, что позволяет существенно ускорить процесс скрещивания [3.4].

Популяцией назовём множество хромосом, состоящих из одного и того же набора генов. Функция выживаемости (целевая функция) – функция оптимизации, заданная формулой (2).

В нашей задаче работы поступают в общую очередь, в которой собираются в пакеты размера d, после чего для пакета строится расписание. Начальная популяция генерируется при помощи алгоритма $Backfill$ [8].

3.2 Генетический алгоритм

Входными данными ГА являются пакет работ $\{J_k\}, 1 \leq k \leq d$, среда $compEnv$, предназначенная для выполнения всех работ, вероятности мутации p_m и скрещивания p_c, и число итераций генетического алгоритма.

Алгоритм состоит из следующих шагов:

1. С помощью алгоритма $Backfill$ сгенерировать начальную популяцию из множества работ в пакете $X = X_a : a \in A$, где A – множество номеров расписаний, кодируемых хромосомами в популяции генетического алгоритма, X_a – хромосома.

2. Вычислить значения функции пригодности f_a для всех хромосом из множества X.

3. Проверить критерий останова (достигнута заданная точность, превышено максимальное число итераций).

4. Сохранить элитные (с наименьшим значением целевой функции) хромосомы. Из множества X сформировать множество Y, состоящее из E хромосом с наименьшими значениями f_a.

5. *Селекция.* Создается подмножество $X^{'} \subseteq X$ хромосом, выбранных методом турнирной селекции.

6. *Скрещивание.* Для каждой хромосомы из подмножества $X^{'}$ случайным образом выбирается число на отрезке $[0, 1]$, и если это число меньше или равно p_c, то к данной хромосоме применяется оператор скрещивания. Новые хромосомы, полученные в результате действия на них оператора скрещивания, заменяют в $X^{'}$ хромосомы с максимальными значениями f_a.

7. *Мутация.* Для каждой хромосомы из подмножества $X^{'}$ случайным образом выбирается число на отрезке $[0, 1]$, и если это число меньше или равно p_m, то к данной хромосоме применяется оператор мутации. Каждая хромосома, полученная в результате действия оператора мутации, заменяет в $X^{'}$ исходную хромосому.

8. *Восстановление элитных хромосом.* Хромосомы с максимальными значениями f_a в подмножестве $X^{'}$ заменяются на хромосомы из множества Y.

9. Текущей популяцией становится $X^{'} : X = X^{'}$. Осуществляется переход на шаг 2.

Как уже отмечалось ранее хромосома состоит из последовательности генов $\{C_i, a_k(C_i), b_k(C_i), pid_k, cores_k\}$, где C_i – вычислитель, $1 \leq i \leq N, a_k(C_i), b_k(C_i)$ – время начала и конца выполнения работы J_k на вычислителе C_i, pid_k – идентификационный номер работы J_k, $cores_k$ – количество ядер, необходимое для выполнения работы J_k на вычислителе C_i.

Необходимыми условиями при построении расписания в нашей работе являются его целостность и корректность. Под целостностью понимается неизменность количества работ в расписании, определённых на этапе постановки работ в очередь. Классические варианты операторов ГА могут нарушать целостность расписания, изменяя число тех или иных генов. Так, классический вариант операции мутации может заменить в хромосоме один ген другим, не учитывая, что полученный ген может уже содержаться в хромосоме. Таким образом, в расписании образуются копии работ. Подобная ситуация может возникнуть и при использовании классического оператора скрещивания. Поэтому в данной работе используются специальные операторы мутации и скрещивания, описанные ниже.

Под корректностью понимается корректное распределение ресурсов вычислителей, то есть такое распределение, при котором не нарушаются ресурсные ограничения (1).

3.3 Операция селекции

Селекция – это выбор тех хромосом, которые будут участвовать в создании потомков для следующей популяции, т.е. для очередного поколения. Существуют различные способы селекции. В данной работе используется детерминированная турнирная селекция [3].

3.4 Операция скрещивания

Для сохранения целостности расписания при операции скрещивания необходимо удалять добавленные в дочернюю хромосому гены из обеих родительских хромосом, благодаря чему можно избежать копий генов в хромосоме. На вход оператору скрещивания подаются две родительские хромосомы $X_{p_1}, X_{p_2} \in X$, где X – множество хромосом в популяции. На выходе получается дочерняя хромосома X_{ch}.

Напомним, что хромосома – это множество групп, содержащих последовательности генов, расположенных в порядке выполнения соответствующих им работ. Каждая группа относится к отдельному вычислителю. Пусть N – количество таких групп (количество вычислителей). Тогда родительские хромосомы можно представить в следующем виде: $X_{p_r} = \cup_{n=1}^{N} X_{p_r,n}, r = 1, 2$, $X_{p_r,i}$ – i-ая группа в хромосоме p_r.

Операция скрещивания состоит в следующем:

1. В группу n хромосомы X_{ch} добавляются гены из множества $X_{p_1,n} \setminus X_{ch}$, если n – четное, либо из множества $X_{p_2,n} \setminus X_{ch}$, если n – нечетное.

2. Добавленные гены удаляются из обеих родительских хромосом.

3. Если после прохода по всем группам в родительских хромосомах ещё остались гены, то каждый оставшийся ген из хромосомы X_{p_1} добавляется в случайную группу гена X_{ch}.

4. После выполнения операции скрещивания могла нарушиться корректность расписания, поэтому для каждого гена необходимо пересчитать время начала и конца выполнения соответствующей ему работы.

3.5 Операция мутации

Оператор мутации реализует обмен генов в хромосоме. При однократной мутации выбираются два случайных гена, которые впоследствии меняются местами. Очевидно, что при таком обмене не нарушается целостность расписания, но возможно нарушение корректности, так как после обмена гены могут быть сопоставлены вычислителям с отличными от прежних характеристиками. Поэтому после обмена необходимо восстанавливать корректность расписания, то есть назначать генам другие интервалы выполнения.

Для равномерного распределения работ по вычислителям при выборе первого и второго генов используются различные вероятности их выбора, то есть расчет вероятностей при выборе первого и второго генов для обмена различен. В результате работам чаще сопоставляются наименее загруженные вычислители [4].

4. Экспериментальное исследование свойств алгоритма

Для подбора параметров d (максимальный размер пакетов) и *packlimit* (лимит времени пакетизации) был произведен запуск на сгенерированных данных. Сгенерированный набор состоял из 100 работ. Все параметры были сгенерированы при помощи равномерного распределения, максимальное количество потребляемых работами ядер - 10, минимальное - 1, время их выполнения не превышает 25000 с. Результаты приведены на графиках (рис. 1, рис. 2).

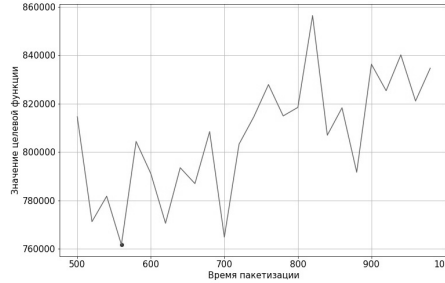


Рис. 1: Зависимость целевой функции (2) от лимита времени пакетизации

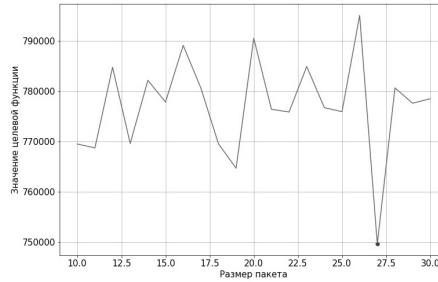


Рис. 2: Зависимость целевой функции (2) от максимального размера пакета

Для *packlimit* наиболее оптимальным значением оказалось значение 560. Для d наиболее оптимальным значением оказалось значение 27.

Для подбора параметров ГА было проведено несколько запусков на сгенерированных данных (рис. 3). На графике изображена зависимость целевой функции от значения вероятности мутации при заданных значениях вероятности скрещивания. Видно, что при увеличении вероятности скрещивания, оптимальное значение вероятности мутации уменьшается. Оптимумы изображены на графиках жирными точками. Оптимальным оказались следующие значения: $cros_{prob} = 0.4$, $mut_{prob} = 0.52$.

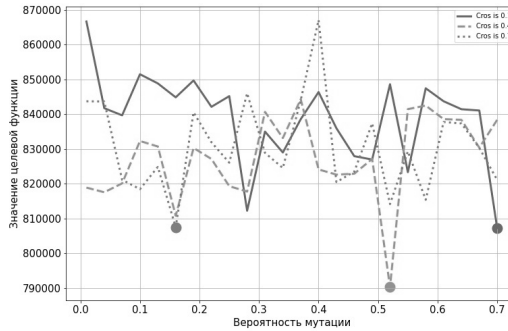


Рис. 3

Для оценки описанного ГА был разработан переборный алгоритм с отсечением наименее оптимальных вариантов (класс *Greedy*). Алгоритм строит расписания для пакета работ. Каждый новый пакет он планирует таким образом, чтобы целевая функция оказалась минимальной, то есть на каждом шаге планирования работы из пакета назначаются на вычислители так, чтобы время их пребывания в ВС оказалось минимальным. Таким образом, алгоритм старается оптимально спланировать каждый новый пакет при условии, что предыдущие пакеты уже спланированы и уже построенное расписание не подлежит изменению. Данный алгоритм не находит оптимум для всей задачи, ведь полный перебор всех существующих вариантов, даже с отсечением заведомо неоптимальных, вычислительно трудоемок для больших задач, однако алгоритм находит оптимум для каждого пакета.

Отсечение происходит следующим образом. После планирования очередной работы предсказывается время пребывания в ВС остальных, ещё не спланированных работ, это возможно сделать, так как нам доступна оценка времени выполнения всех работ на каждом из вычислителей (функция *predtime*) и время поступления всех работ в ВС. Предсказанное время суммируется со значением целевой функции для уже спланированных работ из текущего пакета. Если полученная сумма превосходит текущее наилучшее значение, то данная ветка планирования обрывается.

Так как даже такой перебор занимает приличное время, потому что для поиска лучшего решения необходимо для начала получить какое-то решение, с которым все остальные будут сравниваться, а

первое полученное решение может лежать далеко от оптимума, алгоритм запускался на пакетах небольшого размера $(4 - 6)$.

Было проведено несколько экспериментов. Их результаты приведены на графиках (рис. 4 - рис. 5) и в таблицах (таблица 1 – таблица 3). Для пакетов максимального размера 6 алгоритм *Greedy* работал больше 4-х часов и смог спланировать лишь 6 пакетов, т.к. перебирались все возможные варианты распределения работ по вычислителям, т.е. при числе вычислителей 5, верхняя оценка числа вариантов распределения работ – $5^6 * 720$ с учётом порядка планирования (каждую из 6-ти работ пытаемся распределить на каждый вычислитель, учитывая порядок планирования работ). В качестве входных данных алгоритмам подавались сгенерированные данные, содержащие 100 работ.

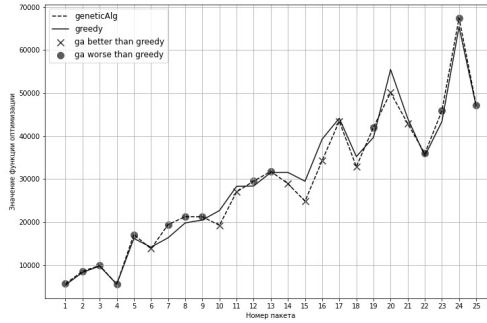


Рис. 4: Значения целевой функции для пакетов от 1 до 25 для алгоритмов *Greedy* и *GeneticAlgorithm*; размер пакета – 4.

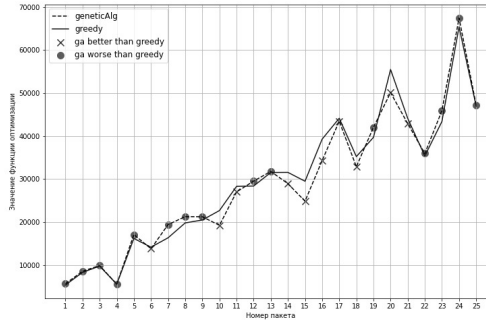


Рис. 5: Значения целевой функции для пакетов от 1 до 20 для алгоритмов *Greedy* и *GeneticAlgorithm*; размер пакета – 5.

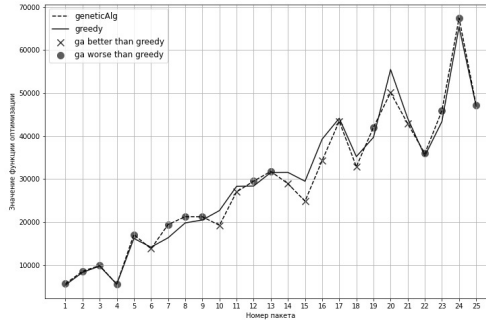


Рис. 6: Значения целевой функции для пакетов от 1 до 17 для алгоритмов *Greedy* и *GeneticAlgorithm*; размер пакета – 6.

Алгоритм	Целевая функция	Время работы
Greedy	800711	15 минут 25 секунд
GeneticAlgorithm	808101	47,5 секунд

Таблица 1: Размер пакета – 4.

Алгоритм	Целевая функция	Время работы
Greedy	797908	1 час 18 минут
GeneticAlgorithm	819988	2 мин. 14 секунд

Таблица 2: Размер пакета – 5.

Алгоритм	Целевая функция	Время работы
Greedy	—	Больше 4-х часов
GeneticAlgorithm	812939	1 мин. 32 секунды

Таблица 3: Размер пакета – 6.

Из графиков видно, что при планировании некоторых пакетов ГА показывает результаты лучше (жирные точки на графиках), нежели переборный алгоритм для планирования задач из одного пакета. Это вызвано неоднородностью ВС: каждый вычислитель обладает своими параметрами, поэтому на одном вычислителе работа может выполняться быстрее, чем на другом. На одном из предыдущих шагов могло выйти так, что ГА спланировал больше работ на более слабый вычислитель, поэтому на следующем шаге планирования вычислитель сильнее окажется свободнее, тогда как переборный алгоритм в попытке достичь наилучшего решения на каждом шаге явно не оставит наиболее сильный вычислитель свободным. А значит, уже на следующем шаге планирования у ГА будет преимущество перед переборным алгоритмом. Но в конечном итоге ГА всё равно получает значения целевой функции больше, чем переборный алгоритм (таблица 1 - таблица 3). В среднем значение целевой функции, полученное при помощи ГА, отклоняется от значения целевой функции, полученного при помощи алгоритма перебора с отсечением, не более чем на 3%.

Заключение

В работе сформулирована задача построения расписаний в гетерогенных ВС при условии ограниченности ресурсов (ядер). Для решения поставленной задачи был выбран класс генетических алгоритмов.

На основе выбранного подхода был разработан и реализован генетический алгоритм со специальными генетическими операторами [9], так как классические генетические операторы могут нарушать целостность расписания. Важным этапом в поиске

решения при помощи генетического алгоритма оказалось поддержание корректности при помощи пересчета параметров слотов на вычислителях после применения к хромосоме операторов мутации и скрещивания. Также была реализована модель ВС для исследования разработанного ГА.

При исследовании свойств разработанного ГА подобраны оптимальные значения его параметров, а также параметров ВС.

Для оценки качества разработанного ГА был приведен переборный алгоритм планирования с отсечением заведомо не оптимальных вариантов.

Оказалось, что при планировании некоторых пакетов ГА может получить значение целевой функции, меньшее, чем при планировании при помощи переборного алгоритма с отсечением, что вызвано неоднородностью ВС, но в конечном итоге переборный алгоритм всё равно получает меньшее значение целевой функции, нежели ГА. Значения целевой функции, полученные при помощи ГА, отклоняются от значений, полученных переборным алгоритмом, в среднем на 3%, но время работы переборного алгоритма с отсечением во много раз (таблицы 1 - 3) превосходит время работы ГА, из-за чего ГА оказывается наиболее перспективным. Таким образом, было показано, что разработанный ГА хорошо подходит для планирования пакетов работ в гетерогенных вычислительных средах при условии ограниченности ресурсов.

Литература

1. Меликян А. Л. *Методы построения расписаний для реализации пакета задач в многопроцессорной вычислительной системе типа ПС-3000.* / А. Л. Меликян // Телемеханика. – 1983. – №12. – с. 148-160.

2. Григорьева Н. С., *Алгоритм ветвей и границ для задачи составления расписания на параллельных процессорах* / Н. С. Григорьева // Вестник Санкт-Петербургского университета. Серия 10: прикладная математика. – 2009. – №1. – с. 44-55.

3. Смагин С. И., *Генетический алгоритм составления расписания выполнения параллельных заданий в распределенной вычислительной системе* / С. И. Смагин,

Т. С. Шаповалов // Вычислительные технологии. Том 15. – 2010. – №5. – с. 107-122. – с. 44-55.

4. Шаповалов Т. С., *Генетический алгоритм составления расписаний для распределенных гетерогенных вычислительных систем* / Т. С. Шаповалов, В. В. Пересветов // Вычислительные методы и программирование. Том 10. – 2009. – №2. – с. 13-21.

5. Костенко В. А., , 2015, № 4, С. 24-38 *Генетический алгоритм с самообучением* /В. А. Костенко, А. В. Фролов// Известия РАН. Теория и системы управления. – 2015. – №4. – с. 24-38.

6. D. A. Zorin, *Algorithm to Simulate Annealing in Problems of Multiprocessor Scheduling* / D. A. Zorin, V. A. Kostenko // Automation and Remote Control, 2014, Vol. 75, No. 10, pp. 1790–1801. DOI: 10.1134/S0005117914100063.

7. Po-Han Chen, *Hybrid of genetic algorithm and simulated annealing for multiple project scheduling with multiple resource constraints* / Po-Han Chen, Seyed Mohsen Shahandashti // Automation in Construction. – 2009. – №18. – PP. 434-443.

8. Мазалов В. В., *Оценка характеристик алгоритма Backfill при управлении потоком задач на вычислительном кластере* / В. В. Мазалов, Н. Н. Никитина // Вычислительные технологии. Том 17. – 2012. – №5. – с. 71-79.

9. *Алгоритмы построения расписания выполнения работ в ГВС* https://git.cs.msu.ru/s02180360/scheduling_problem

Бахмуров А.Г., Бодров А.О.

Построение масштабируемой платформы сбора медицинской телеметрии

Введение

В современном мире IT с каждым годом всё большее распространение получает такая сфера, как IoT или Internet of Things (Интернет вещей). Данное понятие представляет собой компьютерную сеть физических предметов («вещей»), которые могут взаимодействовать друг с другом и внешней средой. Широкое распространение IoT получил в медицине, в современных публикациях упоминаемый как Интернет медицинских вещей (IoMT, Internet of Medical Things). IoMT используется для мониторинга состояния человеческого организма в реальном времени, позволяет улучшить процессы лечения, восстановления пациента и их мониторинг, а также усовершенствовать процесс физических тренировок.

Проблема мониторинга состояния человека актуальна не только в медицинским учреждениях, но и вне них. Сбор и обработка информации о текущем функционировании организма человека в реальном времени позволяют быстро анализировать медицинские показатели и выявлять патологии, что может быть определяющим для здоровья конкретного человека при критическом состоянии пациента. Помимо выгоды для самого пациента, мониторинг может быть чрезвычайно актуальным для специалистов в области медицины, которым данная информация будет полезна, как дополнительный источник для исследования поведения организма при различных заболеваниях. При тренировках мониторинг состояния необходим для грамотного распределения нагрузки на организм.

Ограничение на число клиентов платформы заключит её возможности в жёсткие рамки, поэтому для функционирования платформы с различным числом клиентов следует обеспечить её масштабируемость, то есть способность увеличивать производительность системы пропорционально дополнительным ресурсам (оборудованию), что может обеспечить применение современных технологий виртуализации. Таким образом мы сможем избавиться от ограничения на число клиентов платформы мониторинга, что позволит осуществлять мониторинг состояния

значительного количества пациентов одновременно.

1. Обзор технологий виртуализации

В настоящее время осуществлять виртуализацию можно несколькими способами. Для этого можно использовать:

1. Виртуальные машины (VM)
2. Контейнеризацию [2]

1.1 Виртуальные машины (VM)

Виртуальная машина – это программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой платформы (называемой гостевой платформой) и исполняющая программы для гостевой платформы на платформе-хозяине, изолирующая от платформы-хозяина приложения, бинарные файлы и операционные системы. Виртуальная машина обладает всеми виртуальными устройствами и виртуальным жёстким диском, на который устанавливается гостевая операционная система, драйверы устройств и прочие компоненты. Виртуальное оборудование функционирует, как реальное, при этом виртуальная машина изолирована от реального компьютера, однако может иметь общие каталоги файловой системы с ним.

Использование VM вызывает дополнительные расходы на виртуальное оборудование, запуск гостевой ОС и поддержку необходимого окружения для работы приложений.

1.2 Контейнеры

Контейнеризация – это метод легковесной виртуализации, при котором вместе упаковываются и изолируются от домашней ОС приложение, бинарные файлы и библиотеки. В отличие от виртуальных машин контейнеры обеспечивают виртуализацию на уровне операционной системых [3], а не аппаратного обеспечения [Рис.1]. Благодаря виртуализации на уровне ОС, то есть изоляции только приложения, бинарных файлов и библиотек, возможно быстрое развёртывание, простое масштабирование. Также уменьшается объём занимаемого места на жёстком диске по сравнению с виртуализацией с помощью виртуальных машин.

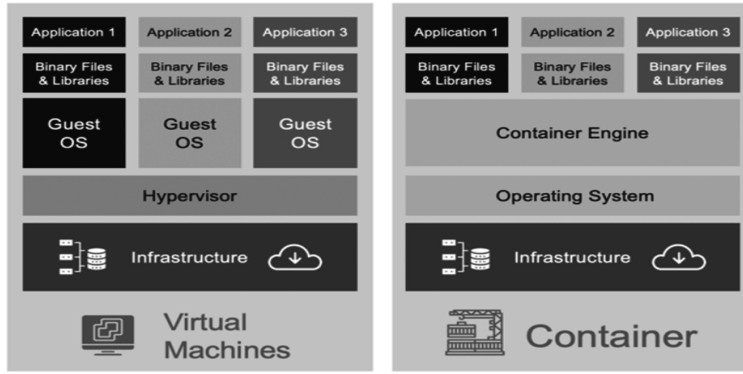


Рис. 1: Сравнение технологий виртуализации виртуальных машин и контейнеров

2. Описание реализации

2.1 Исходные данные

Схема работы приложения

На рис.2 показана упрощенная схема работы платформы, для серверной части которой необходимо реализовать масштабируемость [4].

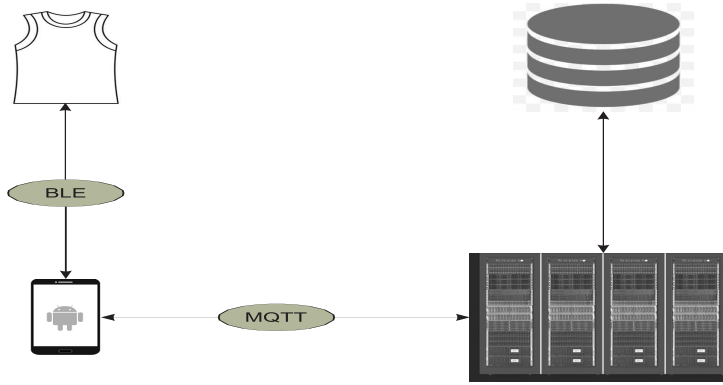


Рис. 2: Схема работы платформы по сбору медицинской телеметрии

В работе сервиса можно выделить следующие этапы:

1. Показатели физиологического состояния пользователя считываются с помощью датчиков умной одежды Hexoskin [1], затем отправляются на мобильный телефон на операционной системе Android по протоколу Bluetooth Low Energy (BLE).

2. Считанные показатели принимаются мобильным телефоном и отправляются на сервер по протоколу MQTT.

3. Сервер записывает полученные данные в базу данных, откуда их можно выгружать для медицинской обработки.

MQTT

Передача данных между клиентом и сервером выполняется по протоколу MQTT [5].

MQTT (message queuing telemetry transport) – сетевой протокол, работающий поверх TCP/IP. MQTT ориентирован на обмен сообщениями между устройствами по принципу издатель-подписчик. Клиент может быть, как издателем или подписчиком, так и иметь обе роли одновременно. Обмен происходит через центральный сервер, который называется брокером (Рис. 5). Клиенты взаимодействуют с брокером через TCP-соединение. Клиенты публикуют топики по TCP-соединению. Топиком назовём механизм, позволяющий давать имена сущностям. Каждое публикуемое сообщение содержит поле для имени топика, которое идентифицирует публикуемые данные. Аналогичным образом, клиент имеет возможность подписаться на определенные топики, если заинтересован в получении конкретных данных. Топики имеют иерархическую структуру в формате “дерева”, что упрощает их организацию и доступ к данным.

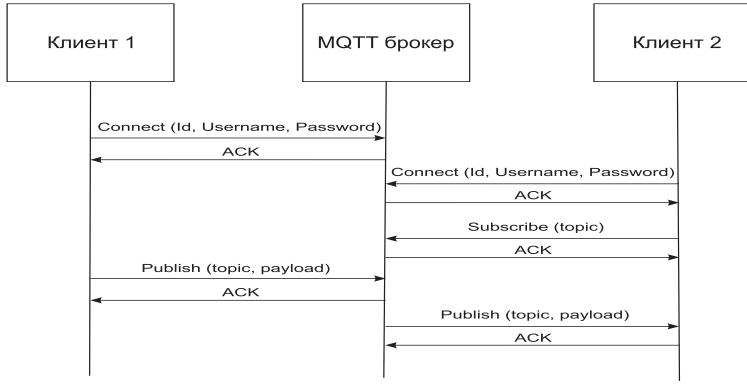


Рис. 3: Схема работы протокола MQTT

Для протокола MQTT выделяется порт 1883. При использовании TLS (криптографического протокола, обеспечивающего защищённую передачу данных) для защиты соединения между клиентом и сервером вместо 1883 используется 8883 порт.

Исходными данными для данной работы и являются сервис записи данных в БД и MQTT-брокер, вместе составляющие серверную часть платформы. Стоит задача масштабирования этой платформы.

2.2 Контейнеризация и масштабирование приложения

Основные понятия

Под масштабируемостью следует понимать способность системы справляться с увеличением рабочей нагрузки путём добавления ресурсов.

Масштабируемость можно разделить на горизонтальную и вертикальную. Под вертикальной масштабируемостью имеется в виду увеличение производительности каждого компонента системы с целью повышения общей производительности. Масштабируемость в этом контексте означает возможность заменять в существующей вычислительной системе компоненты более мощными и быстрыми по мере роста требований и развития технологий.

Горизонтальной масштабируемостью называют разбиение системы на более мелкие компоненты и разнесение их по отдельным машинам или увеличение количества серверов, параллельно выполняющих одну и ту же функцию. Масштабируемость в этом контексте означает возможность добавлять к системе новые узлы, серверы для увеличения общей производительности. Этот способ масштабирования может требовать внесения изменений в программы, чтобы последние могли в полной мере пользоваться возросшим количеством ресурсов. В данной работе рассмотрена именно горизонтальная масштабируемость.

Для оптимизации использования ресурсов, горизонтального масштабирования и обеспечения отказоустойчивости используется балансировка нагрузки – процесс распределения задач между несколькими серверами.

Балансировка нагрузки при помощи HAproxy

В качестве балансировщика нагрузки был выбран HAProxy [6].

HAProxy — серверное программное обеспечение для обеспечения высокой доступности и балансировки нагрузки посредством распределения входящих запросов на несколько обслуживающих серверов.

HAProxy может быть запущено в двух различных режимах – TCP и HTTP. Режимы различаются доступной функциональности. В режиме TCP каждая публикация клиента и каждый приход топика в рамках одного TCP-соединения происходит к одной конкретной копией брокера. В режиме HTTP каждый новый топик приходит на новый брокер. В настоящей работе стоит задача балансировки самого TCP-соединения, а протокол MQTT работает поверх TCP, поэтому представлено использование HAProxy в режиме TCP.

Клиент отправляет запросы HAProxy, выступающему в качестве балансировщика, и получает ответ на каждый. HAProxy распространяет запросы по узлам, взаимодействуя в каждом случае с одним MQTT-брокером (Рис. 4) [7]. Узел может либо принять запрос, если он не перегружен, либо вернуть null или ошибку, тогда HAproxy передаст запрос следующему узлу. Если ни один из узлов не сможет предоставить услугу, то HAproxy вернёт null-значение или ошибку клиенту [8].

Обоснование схемы балансировки и масштабирования приложения

Каждую копию приложения можно подключить к одному брокеру или ко всем сразу. В настоящей работе при распределении нагрузки необходимо избежать дублирования данных, иначе в базе данных Clickhouse пришедшие данные будут повторяться, а избавляться от дублей непосредственно в базе данных сложнее. При этом нам также важно избежать потери данных.

При подключении одного брокера ко всем копиям серверной части платформы возникает проблема «узкого места» при взаимодействии клиентов с брокером. Также возникает дублирование попадающих с базу данных топиков. Если избежать этого и распределить клиентов по копиям сервисов, то возникает проблема синхронизации. При выходе из строя одной из копий серверной части приложения необходимо сообщить об этом остальным (иначе произойдёт потеря данных), а это само по себе является нетривиальной задачей.

В случае разъединения MQTT-брокера и сервиса возникает проблема установки соединения между ними.

Если каждое приложение подписывается на каждый брокер, то также возникает проблема дублирования.

Поэтому каждая копия приложения соединена с одним брокером, и каждая такая пара упакована в Docker-контейнер (Рис. 4). Клиент устанавливает TCP-соединение через балансировщик с одним из брокеров, которые не перегружены. Балансировщик при этом так же упакован в контейнер. При выходе из строя одного из контейнеров, приложений или брокеров, TCP-соединение для соответствующим MQTT-брокером рвётся, и переустанавливается на другой работающий и свободный брокер. Docker-контейнеризация обеспечит изоляцию, благодаря которой получится избежать конфликтов библиотек.

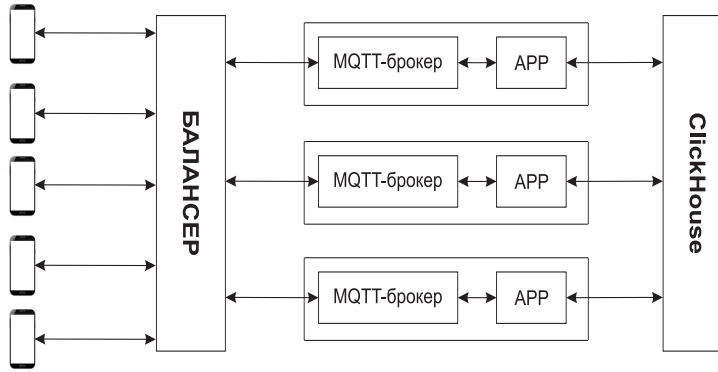


Рис. 4: Балансировка нагрузки между копиями серверной части приложения

Аутентификация

Аутентификация осуществляется так же через один MQTT-брокер, который свободен в этот момент.

Протокол MQTT предоставляет поля для имени и пароля пользователя при запросе соединения с брокером. Эти поля используются для аутентификации, как показано на рис. 5.

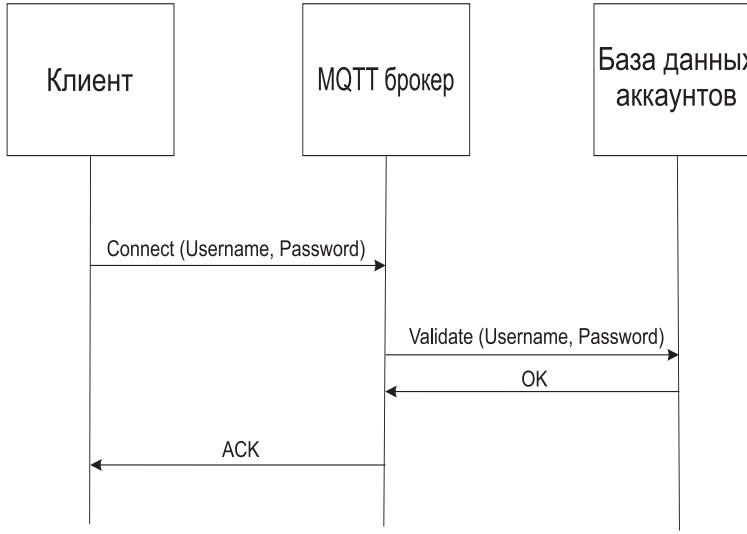


Рис. 5: Процесс аутентификации

При подключении к брокеру клиент предоставляет введенные пользователем логин и пароль в полях сообщения Connect, брокер сверяет полученные данные с учётными записями, находящимися в базе данных акканутов. В настоящей реализации используется брокер Eclipse Mosquitto [9]. Eclipse Mosquitto — брокер с открытым исходным кодом (лицензии EPL/EDL), который реализует протоколы MQTT различных версий.

3. Благодарности

Авторы благодарят сотрудника компании "Яндекс"А.А.Любицкого за техническую помощь и консультации по используемым в проекте программным средствам

Заключение

Разработанная в рамках данной работы система позволяет увеличить нагрузку на серверную часть приложения за счёт горизонтального масштабирования платформы с помощью технологий балансировки нагрузки и контейнеризации. За счёт проделанных исследований повышена отказоустойчивость приложения.

Литература

1. Hexoskin [Электронный ресурс]. – режим доступа: https://www.hexoskin.com/

2. A.M.Potdar, D.G.Narayan, S.Kengond, and M.M.Mulla, “Performance Evaluation of Docker Container and Virtual Machine,” Procedia Comput. Sci., vol. 171, no. 2019, pp. 1419–1428, 2020, doi: 10.1016/j.procs.2020.04.152.

3. Akf partners. [Электронный ресурс]. – режим доступа: https://akfpartners.com/growth-blog/vms-vs-containers

4. Аникевич Ю. В., Бахмуров А. Г., Чайчиц Д. А. Разработка и реализация платформы для сбора и обработки физиологических данных о человеке // Программные системы и инструменты. Тематический сборник № 20 / Под ред. В. А. Антоненко, В. В. Балашов, В. Г. Баула и др. — Т. 20. — Москва: 2020. — С. 36–48. https://drive.google.com/file/d/1L4YhNNTOrhu27o775aA2RqR_1dbgoB6v/view

5. MQTT Version 3.1.1 documentation. [Электронный ресурс]. - режим доступа: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf

6. HAproxy Documentation. [Электронный ресурс]. - режим доступа: https://www.haproxy.com/

7. “How to build an high availability MQTT cluster for the Internet of Things” [Электронный ресурс] – режим доступа: https://medium.com/@lelylan/how-to-build-an-high-avai

lability-mqtt-cluster-for-the-internet-of-things-8011a06bd000

8. Marischa Elveny, Ari Winata, Baihaqi Siregar and Rahmad Syah "A tutorial: Load balancers in a container technology system using docker swarms on a single board computer cluster", EEO, vol. 19, issue 4, no. 2020, pp.744-751, doi: 10.17051/ilkonline.2020.04.178

9. Eclipse Mosquitto [Электронный ресурс] – режим доступа: https://mosquitto.org/

Галкина Е. В.

Транспиляция программ на функциональных языках программирования: подходы и решения

Введение

Качество программного продукта и эффективность работы программиста во многом зависят от выбранного языка программирования, а именно от того, насколько адекватны изобразительные средства языка по отношению к конкретной решаемой задаче. При решении частных подзадач в рамках крупного программного проекта может оказаться удобным использование языковых средств разных парадигм программирования (императивной, функциональной, логической), что позволяет в целом повысить качество программы и упростить процесс её создания. Однако сочетание нескольких языков может быть невозможным при выбранном языке проекта или повлечь за собой слишком высокие накладные расходы, но это осуществимо, когда доступны средства транспиляции.

Транспиляцией [1] называется процесс конвертации программы на одном языке программирования в программу на другом языке. В отличие от компиляции, когда целевой язык является более низкоуровневым, чем исходный, при транспиляции оба языка находятся примерно на одном уровне абстракции. В общем случае можно выделить две основные *цели транспиляции*:

- Миграция между разными версиями одного языка. Конвертация может происходить как и в более новую версию языка, так и в одну из предыдущих, для сохранения совместимости. Например, инструмент `2to3`[1] позволяет автоматически конвертировать код на языке Python 2 в код на Python 3.

- Перевод в другой язык программирования. Исходный и целевой языки могут относиться к различным парадигмам и иметь радикально отличающиеся вычислительные модели. Например, конвертация кода с функциональных языков

[1] https://docs.python.org/3/library/2to3.html

программирования в код на объектно-ориентированных языках.

В данной работе рассматривается задача транспиляции программ на функциональных языках программирования в программы на императивных или объектно-ориентированных языках с целью их использования в одном проекте. Будем называть *базовым* основной язык проекта, на котором реализована бóльшая часть его функционала. Альтернативный язык, используемый для решения частных подзадач, будем называть *дополнительным* языком проекта. Заметим, что таких языков может быть несколько. Транспиляция производится из дополнительного языка в базовый язык проекта, и с этой точки зрения, дополнительный язык является *исходным языком транспиляции*, а базовый — *целевым языком*.

Желание использовать определённый дополнительный язык и конвертировать код на нём в код на базовом языке проекта может быть продиктовано несколькими причинами. Во-первых, это могут быть предпочтения команды разработчиков или требования заказчика. Во-вторых, применение только одного базового языка обычно сокращает количество необходимых инструментов разработки (компиляторы, отладчики, пакетные менеджеры, линтеры и т. д.). В третьих, конвертация даёт возможность использовать уникальные механизмы разных языков и бóльшее количество библиотек. Последнее наиболее актуально, когда решаемая в проекте задача включает в себя несколько предметных областей, для которых более подходящими являются языки с разными выразительными средствами, и количество разработанных на каждом из них библиотек радикально отличается [2, 3]. Например, для символьной обработки текста наиболее удобны функциональные языки, для решения переборных задач — логические, для численных вычислений — императивные.

В качестве целевого языка транспиляции обычно служит тот, который обладает бóльшей переносимостью или лучше поддерживается в целевой среде выполнения [4, 5, 6]. Например, языки C и Java переносимы на большое число платформ, а JavaScript является самым используемым языком для исполнения в браузере. Мотивацией также может быть бóльшая эффективность или уровень развития компилятора для целевого языка [7].

Наравне с Java и JavaScript, язык C++ [8] широко используется в промышленной разработке и на данный момент является одним из самых популярных языков программирования, регулярно занимая

высокие места в рейтингах, таких как TIOBE[2] или ежегодный опрос пользователей Stack Overflow[3].

Язык Haskell [9] является чисто функциональным и обладает большой выразительностью, что позволяет использовать все функциональные средства максимально удобным способом. Кроме того, Haskell поддерживает алгебраические типы данных (АТД), которые удобны для моделирования предметной области, описания объектов и их свойств, и всё чаще в последнее время используется для разработки крупных индустриальных проектов.

В данной статье рассматриваются два возможных подхода к транспиляции [2]: *с сохранением семантики вычислительной модели исходного языка* и *с сохранением структуры программы.* Даётся обзор решений для конвертации программ на функциональных языках в императивные или объектно-ориентированные, описываются ключевые принципы транспиляции кода с языка Haskell на C++ в рамках второго подхода.

1. Подходы к транспиляции

Обозначим за L_1 — исходный язык, L_2 — целевой язык.

Одной из подзадач при транспиляции является разработка правил сопоставления встроенных типов данных языков L_1 и L_2, а также правила преобразования определений пользовательских типов, если они присутствуют в исходном языке. Другой задачей является конвертация вычислительных конструкций языка, которая может осуществляться по-разному [2].

1.1 Конвертация с сохранением семантики

Идея подхода состоит в том, что на целевом языке L_2 частично или полностью эмулируется вычислитель исходного языка L_1, а программа на исходном языке конвертируется в команды для этого вычислителя. Эмулироваться может лишь та часть механизмов, аналоги которых отсутствуют в целевом языке. Остальные вычисления по возможности реализуются с использованием встроенных средств целевого языка.

Более формально процесс можно описать следующим образом:

[2] https://www.tiobe.com/tiobe-index/
[3] https://insights.stackoverflow.com/survey

1. На целевом языке L_2 реализуется вычислительная машина M для интерпретации языка L_1. Для неё определяется набор команд $Cmd(M)$ и правила перевода конструкций на языке L_1 в команды для этой машины. Этот этап достаточно проделать один раз, однако, если необходимо добавить поддержку средств для более новых версий языка L_1, набор команд может расширяться, а эмулятор модифицироваться.

2. Исходный код программы на L_1 преобразуется в набор команд $Cmd(M)$. Эта стадия в свою очередь может состоять из нескольких этапов, такие как синтаксический разбор, перевод в промежуточное представление и синтез команд.

3. Результат транспиляции представляет собой реализацию вычислительной машины M и модуль с командами для этой машины. Таким образом, $Cmd(M)$ играет роль промежуточного языка при транспиляции, а реализация эмулятора становится неотъемлемой частью результирующей программы проекта, как правило, в виде отдельного модуля или библиотеки.

Основное преимущество такого подхода состоит в том, что конвертированы могут быть любые конструкции исходного языка, а поведение результирующей программы идентично поведению исходной (при условии корректной реализации эмулятора). С другой стороны, данный подход имеет ряд недостатков:

- Применение вычислителя исходного языка влечёт за собой низкую эффективность и более высокое потребление памяти по сравнению с программой, которая могла бы быть изначально написана на целевом языке.

- Результатом генерации является нечитабельный код большого размера, который сложно соотнести с исходным, и этот недостаток серьёзно усложняет отладку.

- Ошибки в реализации эмулятора исходного языка приводят к некорректной работе программы, при этом их также трудно локализовать.

1.2 Конвертация с сохранением структуры программы

При данном подходе предварительно разрабатываются правила сопоставления конструкций на языках L_1 и L_2, причём поддерживаться может только определённое подмножество исходного языка, которое можно отобразить, используя аналогичные конструкции в целевом. В результате генерируется более компактный, эффективный и читабельный код, по производительности сопоставимый с «нативными» программами на целевом языке.

Поскольку синтаксис исходного и целевого языка может сильно отличаться, преобразование кода напрямую может представлять определённые трудности. Поэтому для конвертации может использоваться одно или несколько внутренних представлений, например, в виде *абстрактного синтаксического дерева (AST)* транспилируемых языковых конструкций или в виде внутреннего представления, используемого в компиляторе исходного или целевого языка. В случае использования двух различных внутренних представлений предварительно разрабатываются правила для их преобразования.

Обозначим Rep_1 – внутреннее представление для L_1, Rep_2 — внутреннее представление для языка L_2. Упрощённо, процесс конвертации в общем случае включает следующие стадии:

1. Синтаксический разбор исходного кода на L_1 и перевод во внутреннее представление Rep_1.

2. Конвертация внутреннего представления Rep_1 в более подходящее для целевого языка Rep_2.

3. Генерация кода на L_2 по Rep_2.

Если в языке используются макросы, стадия синтаксического разбора может предваряться вызовом макропроцессора.

Rep_1 и Rep_2 могут совпадать, стадия 1 может осуществляться компилятором языка L_1, тогда Rep_1 будет совпадать с внутренним представлением, используемым в компиляторе.

Очевидным недостатком данного подхода является то, что в случае существенных отличий вычислительных средств L_1 и L_2 поддерживается лишь подмножество исходного языка, что накладывает рамки при программировании и ограничивает в использовании всю мощь дополнительного языка, программы на нём должны разрабатываться с учётом дальнейшей транспиляции.

2. Обзор существующих решений

Рассмотрим работы, в которых дополнительный язык относится к функциональной парадигме, «неклассической» для индустриальной разработки.

В работе [2] процесс конвертации кода на Racket, современном потомке языка Lisp, в код на JavaScript реализован с сохранением структуры программы и представлен всеми упомянутыми выше стадиями: макрорасширение, синтаксический разбор во внутреннее представление в виде AST, перевод AST в промежуточный язык (абстрактный синтаксис для JavaScript), оптимизация и генерация результирующего кода на JavaScript.

В рамках подхода с сохранением структуры программы в качестве промежуточного языка опционально может использоваться третий язык, если из него уже реализована конвертация в целевой. Так, в работе [4] производится конвертация программ с Haskell в код на JavaScript через представление STG (Spineless Tagless G-machine — одно из промежуточных представлений, используемых в компиляторе Haskell GHC [10]) и язык SAPL (Simple Application Programming Language — язык внутреннего представления для языка Clean [11]). Тем самым авторы [4] получили возможность конвертировать код на Haskell в эффективный код на JavaScript, используя существующее преобразование SAPL в JavaScript [12] и реализованный ими транслятор STG в SAPL.

В работах [13] и [14] авторы реализуют транспиляцию Haskell в набор команд для JVM (Java Virtual Machine) в рамках подхода с сохранением семантики, используя $< \nu, G >$- и G-машину соответственно [15,16]. Эти абстрактные вычислители были разработаны для имплементации функциональных языков с ленивой стратегией вычисления. Отличие состоит в том, что G-машина используется для вычислений на последовательных процессорах, $< \nu, G >$-машина — на параллельных. Конвертация осуществляется сначала путём перевода программы с Haskell во внутренний язык для G- или $< \nu, G >$-машины, затем выполняется перевод выражений этого языка в низкоуровневые инструкции для этих машин, а они, в свою очередь, преобразуются в инструкции для Java-машины.

В работе [17] также производится конвертация кода на Haskell в байт-код на JVM в рамках подхода с сохранением семантики, но в качестве промежуточных представлений используются STG (промежуточное представление получается с использованием

компилятора GHC) и код на Java, который не предназначен для чтения и модификации человеком, а является промежуточной ступенью для генерации байт-кода для JVM.

Авторы работы [3], напротив, фокусируются на конвертации кода на Haskell в хорошо читаемый код на Java в рамках подхода с сохранением структуры программы, а получение байт-кода JVM не является главной целью. Процесс конвертации состоит из нескольких шагов: разбор кода на Haskell и перевод во внутреннее представление с использованием синтаксического анализатора в составе интерпретатора Hugs[4]; преобразование конструкций из полученного представления во внутреннее представление программы на Java (оно задаётся как тип данных на Haskell внутри системы конвертации); генерация результирующего кода на Java. Разработанная авторами система поддерживает подмножество Haskell и накладывает ограничения на исходный код (например, имена конструкторов типов не должны совпадать с именами конструкторов значений).

3. Транспиляция Haskell в C++

3.1 Использование Haskell и C++ в одном проекте

На данный момент попытки совместного использования Haskell и C++ в основном состоят из применения интерфейса внешних функций (FFI, Foreign Function Interface) и конвертации подмножества кода на Haskell в шаблоны на C++, причём в обучающих или иллюстративных целях[5] [6]. Хотя шаблоны C++ являются мощным механизмом и могут быть полезны при транспиляции кода на Haskell в C++ (например, для имитации полиморфных типов данных), их чрезмерное использование имеет ряд недостатков: код с использованием шаблонов и ошибки компиляции менее понятны, время компиляции и размер результирующего файла увеличиваются.

Современный C++ обладает многими средствами и механизмами, заимствованными из функционального программирования, включая лямбда-выражения, функции высшего порядка, каррирование (они встроены в язык или являются частью стандартной библиотеки, начиная с C++11). Однако по сравнению с

[4] https://www.haskell.org/hugs

[5] https://www.vandenoever.info/blog/2015/07/12/translating-haskell-to-c++.html

[6] https://bartoszmilewski.com/2009/10/21/what-does-haskell-have-to-do-with-c/

языком Haskell, C++ обладает более громоздким синтаксисом и код, включающий большое количество средств, изначально пришедших из функциональной парадигмы, является трудночитаемым и требует больше времени на разработку и поддержку. В то же время, в C++ не включены такие полезные средства функциональных языков, как полиморфные функции, сопоставление с образцом, алгебраические типы данных, классы типов. Классы типов могут быть выражены в языке C++ с использованием *концепций* [18] (с C++20).

Таким образом, целесообразно использовать Haskell как дополнительный язык проекта. Подзадачи, которые более удобно решать на основе Haskell, включают в себя, например, описание сущностей предметной области в виде АТД и определение чистых (без побочных эффектов) функций для описания логики предметной области.

Наличие в C++ средств, пришедших из функциональных языков, даёт возможность генерировать читаемый код на C++ из подмножества кода на Haskell, используя транспиляцию с сохранением структуры программы. Применение именно этого подхода позволит избежать накладных расходов, возникающих при реализации эмулятора исходного языка, и уменьшить трудности при отладке и дальнейшей поддержке кода на C++.

3.2 Поддерживаемое подмножество языка Haskell

Для удобной и полноценной реализации на Haskell решений частных подзадач в рамках проекта на C++ необходимо определить исходное подмножество Haskell для транспиляции. Оно должно быть достаточным для полноценного определения пользовательских типов данных в виде АТД и описания логики предметной области в виде чистых функций. Выбранное нами для решения задачи транспиляции подмножество включает:

1. Базовые атомарные и составные типы данных, АТД, полиморфные типы и функции.
2. Определения пользовательских классов типов и экземпляров классов типов.
3. Средства определения функций:
    - сопоставление с образцом;
    - условные конструкции, включая охранные выражения;
    - конструкции для локального связывания;

- функции высшего порядка и анонимные функции.

4. Базовые операции для ввода-вывода и `do`-нотация для монады `IO` [9].

3.3 Конвертация пользовательских типов

Для иллюстрации рассмотрим транспиляцию определения АТД на Haskell в код на C++. Типу-произведению АТД соответствует описание класса с теми же полями. Конвертация типа-суммы требует введения перечислимого типа, обозначающего конструкторы значений в Haskell-программе, и типа-объединения для хранения самих значений. Кроме того, необходимо исключить возможность создания объектов полученного на C++ класса с неконсистентными данными, для чего генерируются специального вида конструкторы классов на C++. Помимо этих конструкторов необходимо генерировать вспомогательные методы для сопоставления значений с образцом, которые дают возможность определить, какой именно вариант значения типа-суммы используется.

В качестве примера рассмотрим транспиляцию описанного на Haskell типа-суммы `IntOrChar`, значения которого содержат либо целое число, либо символ, но не одновременно:

```
data IntOrChar = IntVal Int | CharVal Char
```

Результат транспиляции этого определения в C++ следующий:

```
typedef enum { // Перечислимый тип для обозначения конструктора данных
  IntOrChar_IntVal,
  IntOrChar_CharVal
} IntOrChar_ConstrTag;

class IntOrChar {
  IntOrChar_ConstrTag constr_tag;
  union { // Объединение для хранения одного из значений типа-суммы
    int intVal;
    char charVal;
  } value;
private:
  IntOrChar() {}; // Конструктор класса по умолчанию объявлен приватным
public:
  IntOrChar(int n) { // Построение значения для IntVal
    constr_tag = IntOrChar_IntVal;
    value.intVal = n;
  }
  IntOrChar(char c) { // Построение значения для CharVal
    constr_tag = IntOrChar_CharVal;
    value.charVal = c;
```

```
  }
  bool IntOrChar_getIntVal(int *res) { // Метод для сопоставления с образцом
    switch (constr_tag) {
      case IntOrChar_IntVal:
        *res = value.intVal;
        return true;
      default: return false;
    }
  }
  bool IntOrChar_getCharVal(char *res) { // Метод для сопоставления с образцом
    switch (constr_tag) {
      case IntOrChar_CharVal:
        *res = value.charVal;
        return true;
      default: return false;
    }
  }
};
```

Помимо специальных конструкторов `IntOrChar(int n)` и `IntOrChar(char c)` генерируются вспомогательные методы для сопоставления значений с образцом: `IntOrChar_getIntVal` и `IntOrChar_getCharVal`.

Определения полиморфных типов конвертируются аналогичным образом в шаблонные классы.

Заключение

В данной работе рассмотрен один из путей совместного использования языков различных парадигм в рамках одного программного проекта — транспиляция кода на дополнительном языке в код на базовом языке проекта. Описаны возможные подходы к транспиляции, выделены их преимущества и недостатки.

Для задачи транспиляции кода с языка Haskell в C++, обоснован подход с сохранением структуры программы, описано подмножество языка Haskell, для которого реализуется транспиляция. Также приведено описание способа транспиляции для объявлений пользовательских типов данных языка Haskell.

Литература

1. ARC-Softwaresystems (June 1988). — Turn BASIC into C: B→C Transpiler. — Amiga-Magazin, Vol. 1988 №6. Esslingen, Germany: Markt & Technik Verlag Aktiengesellschaft. p. 101. ISSN 0933-8713

2. Yaddav V. RacketScript: a Racket to JavaScript compiler. — Northeastern University Boston, MA, 2006.

3. Ricky Barefield. A Haskell-To-Java Translation Tool. [Электронный ресурс]. — Электрон. дан. — URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.5251&rep=rep1&type=pdf (дата обращения 15.08.2021).

4. Domoszlai L, Plasmeijer R. Compiling Haskell to JavaScript through Clean's core. — In Selected papers of 9th Joint Conference on Mathematics and Computer Science (February 2012). — pp. 117–142.

5. Loitsch F, Serrano M. Compiling Scheme to JavaScript. — Inria Sophia Antipolis, 2006.

6. Feeley M, Miller J, Rozas G, Wilson J. Compiling higher-order languages into fully tail-recursive portable C. — Rapport technique. — 1997 Aug 18.

7. Marc Feeley and Martin Larose. Etos: an Erlang to Scheme compiler. — Universite de Montreal, 1997.

8. ISO/IEC 14882:2020(E) — Programming Language C++. — International Organization for Standardization, Geneva, Switzerland. — 2020.

9. Simon Peyton Jones, ed. Haskell 98 Language and Libraries: The Revised Report. — Cambridge University Press, 2003.

10. GHC User's Guide [Электронный ресурс]. — Электрон. дан. — URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide (дата обращения 15.08.2021).

11. Clean Wiki [Электронный ресурс]. — Электрон. дан. — URL: https://clean.cs.ru.nl (дата обращения 15.08.2021).

12. L. Domoszlai, E. Bruël and J. M. Jansen. Implementing a non-strict functional language in JavaScript. — Acta Universitatis Sapientiae 3(1):76– 98, 2011, Sapienta University, Scienta Publishing House

13. Wakeling D. Mobile Haskell: Compiling lazy functional programs for the Java virtual machine. — In Principles of Declarative Programming. — pp. 335–352. — Springer, Berlin, Heidelberg, 1998.

14. D. Wakeling. A Haskell to Java Virtual Machine Code Compiler. In Proceedings of the 1997 Workshop on the Implementation of Functional Languages. — Springer-Verlag, September 1997.

15. L. Augustsson and T. Johnsson. Parallel Graph Reduction with the $< \nu, G >$-machine. — In Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture. — pp. 202–213. — ACM Press, 1989.

16. T. Johnsson. Ecient Compilation of Lazy Evaluation. — In Proceedings of the SIGPLAN'84 Symposium on Compiler Construction, pages 58-69. — June 1984.

17. Tullsen, M. Compiling Haskell to Java. — Yale University, 1996.

18. Bernardy J.,P. et al. A comparison of C++ concepts and Haskell type classes. — Proceedings of the ACM SIGPLAN workshop on Generic programming, pages 37-48. — Sep 2008.

Кузьмин Я. К., Волканов Д. Ю.

Разработка метода синхронизации состояния алгоритма обработки пакетов в сетевом процессорном устройстве[1]

Введение

Программно конфигурируемые сети (ПКС) в настоящее время являются активно развивающейся технологией [1]. Основой технологии ПКС является перенос управляющих функций сети из сетевых устройств на отдельное устройство, называемое контроллером. При таком подходе контроллер выполняет централизованное управление сетью путём отправки команд коммутаторам. В свою очередь, коммутаторы выполняют обработку пакетов на основе правил, полученных от контроллера.

Сетевое процессорное устройство (СПУ) — это специализированная интегральная схема, предназначенная для обработки пакетов в сетевых устройствах, например, в коммутаторах. Программируемые СПУ — это тип СПУ, позволяющих загружать новые алгоритмы обработки пакетов и определять новые сетевые протоколы в процессе работы при помощи обновления записанной в СПУ программы [2].

В настоящее время активно развиваются многопоточные транспортные протоколы, такие как MPTCP [3]. Использование многопоточных транспортных протоколов позволяет достичь большей скорости передачи данных благодаря более полному использованию ресурсов сети. Для наиболее полного использования ресурсов сети необходимо использовать разные маршруты для разных транспортных потоков и балансировать потоки между маршрутами. Но в ПКС алгоритмы балансировки транспортных потоков создают высокую нагрузку на контроллер, так как являются алгоритмами обработки пакетов, требующими хранения состояния [4]. Решением данной проблемы является перенос состояния алгоритма балансировки транспортных потоков с контроллера на коммутатор.

В данной работе предлагаются модификации архитектуры программируемого СПУ, позволяющие синхронизировать состояние алгоритма обработки пакетов между портами СПУ.

[1]Работа выполнена при частичной поддержке РФФИ, грант № 19-07-01076.

Работа имеет следующую структуру: в первой части представлена постановка задачи, во второй части описана исходная архитектура СПУ. В третьей части рассмотрены два предлагаемых подхода к хранению состояния алгоритма обработки пакетов и соответствующие этим подходам модификации архитектуры СПУ. В четвёртой части представлена имитационная модель СПУ и описаны её модификации. В пятой части описано проведённое экспериментальное исследование для двух предлагаемых подходов.

1. Постановка задачи

Дана исходная архитектура СПУ, имеющая следующие особенности:

- СПУ работает потактово;
- каждый порт СПУ имеет отдельный вычислительный конвейер;
- вычислительные конвейеры не связаны друг с другом и работают параллельно, синхронно;
- вычислительные конвейеры имеют одинаковое количество стадий;
- время обработки пакета стадией вычислительного конвейера имеет ограничение в 250 тактов;
- таблица классификации распределена по устройствам памяти, находящемся на стадии конвейера.

Состояние алгоритма обработки пакетов представлено в виде набора переменных.

Требуется разработать метод хранения состояния алгоритма обработки пакетов с его синхронизацией между вычислительными конвейерами СПУ, не нарушающий ограничение на время обработки пакета стадией вычислительного конвейера СПУ. При разработке модификаций требуется учесть особенности архитектуры СПУ.

2. Исходная архитектура СПУ

В ходе обработки в вычислительном конвейере заголовок пакета последовательно проходит по его стадиям (Рис. 1). Для обработки заголовка на каждой стадии отводится 250 тактов. Стадия вычислительного конвейера состоит из устройства памяти и вычислительного ядра. Устройство памяти используется для хранения части программы обработки пакетов и заголовка и метаданных обрабатываемого пакета. Вычислительное ядро выполняет программу, записанную в устройстве памяти. В ходе выпонения программы область памяти, содержащая заголовок и метаданные пакета, доступна на чтение и запись для вычислительного ядра.

Обработка заголовка пакета стадией вычислительного конвейера происходит согласно следующим этапам:

1. загрузка заголовка и метаданных в область устройства памяти стадии, выделенной для их хранения;
2. выполнение программы, записанной в устройство памяти стадии;
3. передача заголовка и метаданных на следующую стадию.

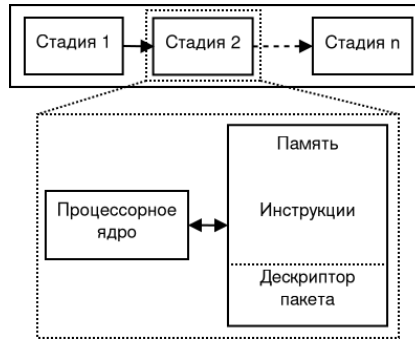


Рис. 1: Схема конвейера СПУ.

Рассмотрим применимость алгоритмов обработки пакетов с хранением состояния, требующих синхронизации состояния между портами в СПУ с данной архитектурой. В данном СПУ конвейеры не имеют связи друг с другом и не имеют общих ресурсов, которые бы позволяли передавать информацию об изменении состояния

между конвейерами. Помимо этого, на стадиях вычислительного конвейера не предусмотрены устройства памяти, позволяющие хранить состояние алгоритма обработки пакетов, то есть данные, доступные на запись для вычислительного ядра и сохраняемые при переходе к обработке следующего пакета. Таким образом, алгоритмы обработки пакетов с хранением состояния, требующие синхронизации состояния между портами СПУ неприменимы в СПУ с данной архитектурой. Следовательно, требуется предложить модификации архитектуры СПУ RuNPU, позволяющие применять подобные алгоритмы.

3. Предлагаемые подходы к хранению и синхронизации состояния алгоритма обработки пакетов

3.1 Общая шина

Данный метод основан на подходе, описанном в работе [5]. В данном методе предлагаются следующие модификации архитектуры СПУ: на каждую стадию каждого вычислительного конвейера добавлены устройства памяти для хранения переменных состояния алгоритма обработки пакетов [6]. Стадии вычислительных конвейеров одинаковой глубины связываются общими шинами. Шины используются для синхронизации данных в устройствах памяти для хранения состояния алгоритма обработки пакетов на стадиях вычислительных конвейеров.

При записи в устройство памяти для хранения состояния по шине передаётся сообщение, содержащее адрес и новое значение соответствующей ячейки памяти. Все стадии, имеющие доступ к данной шине, обновляют значение соответствующей ячейки памяти в устройстве памяти для хранения состояния. Вычислительному ядру доступны две операции с памятью для хранения состояния: чтение из памяти по произвольному адресу и запись в память по произвольному адресу.

При чтении данных из устройства памяти для хранения состояния происходит извлечение значения из соответствующей ячейки локального для данного вычислительного ядра устройства памяти (находящегося на той же стадии того же вычислительного конвейера, что и вычислительное ядро). При записи значения в устройство памяти для хранения состояния помимо записи в локальное устройство памяти новое значение передаётся

устройствам памяти на всех вычислительных конвейерах по шине. Таким образом, при получении сообщения со стороны шины устройство памяти выполняет обновление значения соответствующей ячейки памяти.

Для координации доступа к шине используется арбитр шины. Для избегания ресурсного голодания используется циклическая передача приоритета между стадиями вычислительных конвейеров, требующих доступ к шине.

3.2 Сеть

Данный подход основан на методе, предложенном в работе [7] и предполагает соединение устройств памяти для хранения состояния и процессорных ядер при помощи сети комммутации пакетов со специализированными коммутаторами, позволяющими выполнять операцию объединения запросов В качестве сети, соединяющей вычислительные ядра и устройства памяти, рассматривается Ω-сеть [8]. Сеть разделена на слои, коммутатор на каждом уровне сети имеет четыре соединения: по два для предыдущего слоя и следующего слоя. Каналы связи между слоями строятся по следующему правилу: пусть $s_1 s_2 \dots s_{n-1} s_n$ — это двоичная запись номера некоторого выхода слоя k. Тогда этот выход соединён с таким входом слоя $k + 1$, номер которого имеет двоичное представление $s_2 \dots s_{n-1} s_n s_1$. Количество слоёв (и, соответственно, длина пути от вычислительного ядра до устройства памяти) в Ω-сети составляет $\log_2(N)$, где N — количество входов сети.

Запрос к памяти состоит из следующей последовательности операций: чтение, изменение, запись (Листинг 1). При обращении к памяти формируется и передаётся по сети тройка значений $(id, addr, f)$, где id — номер запроса (должен быть уникальным), $addr$ — адрес ячейки памяти, к которой обращён запрос, f — идентификтор функции, применяемой к значению в ячейке памяти (операция изменения значения). В качестве возвращаемого значения операции доступа к памяти используется значение, находившееся в ячейке памяти до выполнения операции преобразования, соответствующей запросу. Это значение передаётся из устройства памяти в вычислительное ядро в ответе на запрос к памяти, который имеет вид (id, val), где id — номер запроса, val — передаваемое значение из ячейки памяти.

Листинг 1: Последовательность операций при запросе к памяти.

```
function RMW(X, f)
```

```
{
    temp = X;
    X = f(X);
    return temp;
}
```

В виде, в котором данный подход рассмотрен в статье [7], отсутствует возможность выполнять операции, которые требуют передачи параметра функции преобразования, например, присваивание. Для решения данной проблемы предлагается добавление к запросу к памяти нового поля, содержащего параметр функции преобразования: $(id, addr, f, parameter)$.

4. Описание разработанного программного средства

4.1 Имитационная модель СПУ

Для оценки предложенных в разделах 3.1 и 3.2 подходов проведено экспериментальное исследование. Для проведения экспериментального исследования было разработано программное средство, представляющее собой имитационную модель СПУ. За основу было взято программное средство, представляющее собой имитационную модель исходной архитектуры СПУ RuNPU [9]. Данная имитационная модель написана на языке программирования Python.

В ходе исследования на вход имитационной модели подаются текстовый файл с текстом программы на языке ассемблера и pcap файлы с сетевыми пакетами, подаваемыми на входные порты СПУ, каждый файл соответствует своему входному порту. Формат pcap позволяет хранить в файле последовательность сетевых пакетов.

Рис. 2: Схема работы имитационной модели.

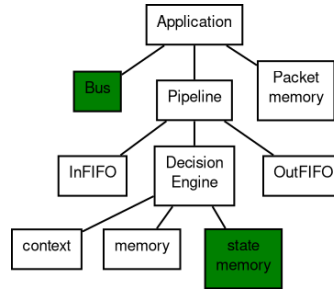


Рис. 3: Схема связи модулей имитационной модели с общей шиной.

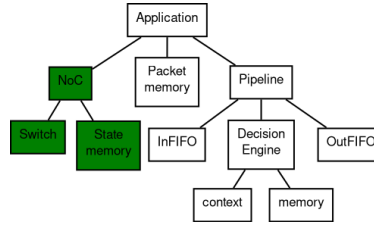


Рис. 4: Схема связи модулей имитационной модели с сетью с объединением запросов.

Имитационная модель позволяет оценивать производительность блока вычислительных конвейеров. Оценивается пропускная способность (в пакетах в секунду), среднее и максимальное количество тактов, требующееся для обработки одного пакета.

Имитационная модель состоит из следующих модулей:

- **Application** — основной модуль программы;
- **Pipeline** — модуль, моделирующий работу конвейера;
- **InFIFO** — модуль, моделирующий работу входной очереди. В данном модуле происходит чтение pcap файлов с пакетами, подаваемыми на входные порты;
- **OutFIFO** — модуль, моделирующий работу выходной очереди. В данном модуле происходит запись pcap файла, соответствующего данному порту;
- **DE** — модуль, моделирующий работу вычислительного ядра;
- **Memory** — модуль, моделирующий работу устройства памяти, находящегося на стадии конвейера;

- **Context** — модуль, хранящий состояние вычислительного ядра (адрес текущей инструкции, значение аккумулятора, регистра сдвига);

- **Packet memory** — модуль, моделирующий работу устройства памяти, используемого для хранения тел пакетов.

Имитационная модель моделирует потактовую работу модулей. Каждый такт происходит вызов метода `tick` каждого модуля имитационной модели. При этом каждый модуль выполняет действия, соответствующие данному такту (Рис. 2).

Для каждого из рассматриваемых подходов в имитационную модель внесён соответствующий набор модификаций.

4.2 Общая шина

Для реализации данного подхода в имитационную модель были добавлены следующие модули (Рис. 3):

- **State memory** — модуль, реализующий функциональность устройств памяти для хранения состояния;

- **Bus** — модуль, моделирующий работу шины, связывающий устройства памяти разных конвейеров.

В методе доступа к шине модуля Bus реализован алгоритм циклической передачи приоритета, реализующий функции арбитра шины.

4.3 Сеть

Для реализации данного подхода в имитационную модель были добавлены следующие модули (Рис. 4):

- **State memory** — модуль, реализующий функциональность устройств памяти для хранения состояния;

- **NoC** — модуль, моделирующий работу сети передачи данных, связывающей устройства памяти и вычислительные ядра на стадиях конвейеров.

- **Switch** — модуль, моделирующий работу коммутатора в сети, соединяющей вычислительные ядра и устройства памяти для хранения состояния.

Модуль NoC отвечает за создание топологии сети (создание коммутаторов и каналов связи между коммутаторами, устройствами памяти и вычислительными ядрами). Обработка одного запроса коммутатором происходит за один такт.

5. Экспериментальное исследование

5.1 Цель экспериментального исследования

Цель экспериментального исследования — оценить применимость подходов, предложенных в разделах 3.1 и 3.2 и оценить скорость обработки заголовков пакетов модифицированным блоком вычислительных конвейеров. Оценка применимости означает проверку, приводят ли предложенные модификации к нарушению ограничений архитектуры СПУ.

5.2 Методика экспериментального исследования

Для проведения экспериментального исследования была использована имитационная модель, описанная в части 4. Написана программа на языке ассемблера для СПУ, реализующая алгоритм flowlet switching [4], используемый для балансировки транспортных потоков. Для этого алгоритма требуется хранение состояния в СПУ и его синхронизация между вычислительными конвейерами.

В ходе экспериментального исследования для каждого рассматриваемого подхода производится серия запусков имитационной модели и измеряются характерстики модифицированной архитектуры. С каждым запуском в серии увеличивается на единицу числом конвейеров, одновременно работающих с устройствами памяти для хранения состояния алгоритма обработки пакетов. Таким образом, серия состоит из 24 запусков, в которых от 1 до 24 конвейеров одновременно работают с устройствами памяти для хранения состояния.

Для проведения исследования были сгенерированы наборы тестовых пакетов для каждого запуска имитационной модели. Наборы тестовых пакетов содержат последовательности пакетов, формирующие транспортные потоки. Количество потоков в наборе тестовых пакетов состоавляет 500 потоков на каждый порт СПУ.

5.3 Результаты экспериментального исследования

Общие шины

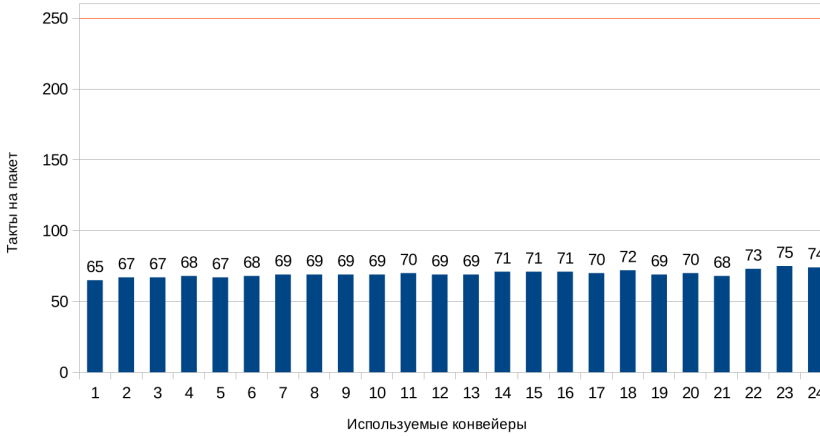


Рис. 5: Результаты экспериментального исследования подхода с общими шинами.

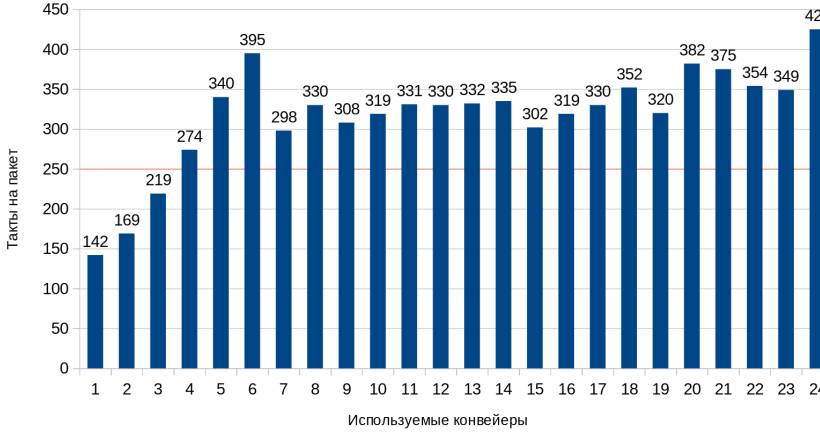


Рис. 6: Результаты экспериментального исследования подхода с механизмом объединения запросов.

По итогам исследования количество тактов, требующихся для обработки заголовка пакета, не превышает 250 (Рис. 5). Таким образом, данный подход применим.

Сеть передачи данных

В ходе исследования данного подхода было обнаружено, что по мере увеличения числа конвейеров, работающих с состоянием, число тактов, требующихся для обработки пакета, быстро растёт и превышает 250 тактов (Рис. 6), что делает данный подход в текущем виде неприменимым в данном СПУ.

Заключение

В ходе выполнения данной работы были выбраны методы синхронизации данных между устройствами памяти. Выбранные методы были адаптированы к имеющейся архитектуре СПУ и реализованы в имитационной модели. С помощью имитационной модели было проведено экспериментальное исследование свойств модифицированной архитектуры СПУ.

В результате только подход, основанный на синхронизации данных при помощи общих шин, является применимым. Однако из экспериментального исследования подхода с сетью передачи данных видна следующая особенность: при одновременной работе с состоянием с большого числа конвейеров не происходит дальнейшее увеличение времени обработки пакета.

Таким образом, дальнейшие исследования могут быть проведены в области оптимизации подхода, основанного на сети, или рассмотрения похожих подходов, так как подходы на основе сети позволяет добиться существенной экономии памяти в отличие от подхода, основанного на общих шинах, так как не требует дублирования одних и тех же данных в разных вычислительных ядрах.

Литература

1. Смелянский Р. Л. *Программно-конфигурируемые сети* //Открытые системы, № 9, 2012, С. 15-26.

2. Беззубцев С. О., Васин В. В., Волканов Д. Ю., Жайлауова Ш. Р., Мирошник В. А., Скобцова Ю. А., Смелянский Р. Л. *Об одном подходе к построению сетевого процессорного устройства* //Моделирование и анализ информационных систем. – 2019. – Т. 26, № 1. – С. 39-62.

3. Ford A., Raiciu C., Handley M., Bonaventure O., Paasch C. *TCP Extensions for Multipath Operation with Multiple Addresses draft-ietf-mptcp-rfc6824bis* //IETF RFC-6824. – 2016.

4. Cascone C., Pollini L., Sanvito D., Capone A. *Traffic Management Applications for Stateful SDN Data Plane* //2015 Fourth European Workshop on Software Defined Networks. – IEEE, 2015. – P. 85-90.

5. Stenstrom P. *A survey of cache coherence schemes for multiprocessors* //Computer. – 1990. – Т. 23. – №. 6. – P. 12-24.

6. Кузьмин Я. К., Волканов Д. Ю., Скобцова Ю. А. *Исследование применимости алгоритмов обработки пакетов с сохранением состояния в архитектуре сетевого процессорного устройства RuNPU* //Программные Системы и Инструменты – Тематический сборник № 20. – Москва, 2020. – С. 82-93.

7. Kruskal C. P., Rudolph L., Snir M. *Efficient Synchronization of Multiprocessors with Shared Memory* //ACM Transactions on Programming Languages and Systems (TOPLAS). – 1988. – Т. 10. – №. 4. – P. 579-601.

8. Lawrie D. H. *Access and alignment of data in an array processor* //IEEE Transactions on Computers. – 1975. – Т. 100. – №. 12. – P. 1145-1155.

9. Markoborodov A., Skobtsova Y., Volkanov D. *Representation of the OpenFlow Switch Flow Table* //2020 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTec). – IEEE, 2020.

Ларин А.В., Пашков В.Н.

Применение нейронных сетей для прогнозирования нагрузки на контроллер в программно-конфигурируемых сетях

Введение

В программно-конфигурируемых сетях (ПКС) [1] в отличие от сетей с традиционной архитектурой ключевым компонентом управления сетью является контроллер - специальное программное обеспечение, запущенное на выделенном сервере в сети. ПКС контроллер осуществляет постоянный мониторинг топологии сети, состояния каналов связи и сетевого обрудования (коммутаторов), и логически централизованное управление маршрутизацией пользовательских потоков данных в сети за счет удаленного изменения конфигурации сетевого оборудования посредством специального протокола управления. Одним из наиболее широко используемых протоколов управления коммутаторами в ПКС сетях является протокол OpenFlow [2].

В соответствии со спецификацией протокола OpenFlow при возникновении нового потока в ПКС/OpenFlow сети коммутатор, в связи с отсутствием соответствующего правила маршрутизации в таблице потоков, отправляет по защищенному TLS/SSL соединению на контроллер соответствующий запрос (в виде Packet-In сообщения). Предустановленные приложения контроллера, получив такой запрос, на основе информации о глобальном состоянии сети (топологии и состоянии таблиц потоков коммутаторов) формируют набор необходимых правил маршрутизации, которые устанавливаются на коммутаторы сети по всему маршруту следования нового пользовательского потока данных с помощью OpenFlow сообщений вида Flow-Mod и Packet-Out.

Таким образом, производительность, стабильность, надежность и масштабируемость ПКС сети существенно зависит от характеристик производительности контроллера: количества Packet-In запросов, которые контроллер способен обработать за единицу времени. Однако предельно допустимая, максимальная производительность контроллера ограничивается физическими характеристиками сервера (объемом оперативной памяти,

производительностью процессора и количеством ядер), на котором он запущен. Поэтому при увеличении количества коммутаторов в сети или при резком росте количества пользовательских потоков в сети потенциально возможно возникновение ситуации перегрузки контроллера, то есть когда количество запросов от коммутаторов к контроллеру превышает предельно допустимую величину. Возникновение перегрузки контроллера в сети может привести к прерываниям в работе пользовательских сетевых сервисов и их недоступности для конечных пользователей, что недопустимо особенно для критически значимых приложений и сервисов.

Одним из основных подходов к предотвращению перегрузки и отказа контроллера в ПКС сети является использование распределенного контроллера в сети, когда каждый узел управляет некоторым ПКС сегментом сети [3]. Это существенно расширяет возможности масштабирования сети, существенно повышает надежность и производительность контура управления ПКС, однако не исключает возможность перегрузки отдельного узла распределенного контроллера.

Для решения этой проблемы в данной работе рассматривается задача прогнозирования нагрузки на контроллер (или отдельный узел распределенного контроллера) для предотвращения ситуации потенциальной перегрузки. Краткосрочное прогнозирование количества входящих запросов на установление новых потоков в сети на контроллер позволяет увеличить и выйграть время для реакции на возможную перегрузку [4]. Например, контур управления может перераспределить управление коммутаторами между узлами распределенного контроллера, тем самым уменьшая нагрузку на перегруженный узел. Таким образом, задача прогнозирования нагрузки на контроллер, рассматриваемая в данной работе, является крайне актуальной для дальнейшего внедрения подхода ПКС в реальных сетях провайдеров, операторов связи и центрах обработки данных.

Целью данной работы является исследование метода на основе нейронных сетей для прогнозирования нагрузки на ПКС контроллер в программно-конфигурируемых сетях. В разделе 1 статьи формулируется задача прогнозирования нагрузки на контроллер в ПКС сети, в разделе 2 проводится сравнительный анализ существующих подходов к прогнозированию нагрузки, в разделе 3 описывается разработанный метод прогнозирования с использованием нейронных сетей, в разделах 4 и 5 описываются реализация и проведенное экспериментальное исследование разработанного метода.

1. Задача прогнозирования нагрузки на контроллер ПКС

Пусть задана программно-конфигурируемая сеть или сегмент ПКС сети, в которой зафиксированы следующие параметры:

- топология сети, находящаяся под управлением одного контроллера;
- количество коммутаторов и каналов связи;
- сообщения между контроллером и коммутаторами передаются по защищенным логическим соединениям.

Пусть взаимодействие между коммутаторами и контроллером осуществляется по протоколу управления OpenFlow [2] версии 1.3.

Введем дискретное время и рассмотрим характеристики взаимодействия контроллера с коммутаторами. Для каждого коммутатора в каждый момент времени известен набор значений следующих показателей:

- количество *Packet-in* запросов, отправленных коммутатором на контроллер в связи с появленеим новых потоков;
- общее количество OpenFlow сообщений, отправленных коммутатором на контроллер.

Количество сообщений оценивается за промежуток времени между двумя последовательными моментами дискретного времени.

Сообщения типа *Packet-in* выделены в отдельную категорию, поскольку они составляют основную нагрузку на контроллер со стороны коммутаторов. В отличие от них, сообщения прочих типов имеют периодичный характер и создают относительно стабильную нагрузку, которая зависит в наибольшей степени от числа коммутаторов в ПКС сети.

Нагрузка на контроллер со стороны одного коммутатора в конкретный момент времени определяется показателем количества *Packet-in* сообщений с данного коммутатора. Общая нагрузка на контроллер в конкретный момент времени определяется, как сумма всех запросов со всех коммутаторов по отдельности.

Ключевыми параметрами, характеризующими производительность контроллера, являются пропускная способность — количество сообщений, которые контроллер способен обработать в единицу времени, и задержка (или время реакции) контроллера на запрос от коммутатора.

Перегрузка контроллера может возникать по следующим причинам:

- увеличение количества потоков в сети;
- DDoS атака;
- подключение нового оборудования или целых сегментов сети;
- аварии и отказы в сети с экстренным переключением и построением резервных маршрутов в сети;
- некоторые запросы могут требовать большого количества генерации контроллером исходящих управляющих сообщений;
- из-за реактивного режима работы.

Эти причины могут приводить к резкому росту количества запросов, которые контроллер не в состоянии обработать, что влечет за собой увеличение времени реакции контроллера на каждый запрос от сетевого оборудования. В связи с этим предположим, что для каждого контроллера задано некоторое пороговое значение количества сообщений в единицу времени, которое он может корректно и без задержек обрабатывать. Данное пороговое значение может быть определено экспериментально (например, с помощью утилиты cbench) за вычетом 10-15 процентов от максимальной нагрузки на контроллер по обработке Packet-in сообщений.

Таким образом, задача прогнозирования нагрузки на контроллер заключается в построении краткосрочного прогноза для показателя суммарной нагрузки от всех коммутаторов на контроллер на несколько ближайших шагов дискретного времени. В случае, если прогноз показывает превышение допустимого порогового значения контроллера, целесообразно заблаговременно осуществить передачу управления одним или несколькими коммутаторами другому узлу распределенного контроллера. Однако для определения этих коммутаторов необходима дополнительная информация, которая позволила бы, например, выбрать коммутатор, создающий максимальную нагрузку. В связи с этим помимо суммарной нагрузки на контроллер требуется прогнозировать нагрузку со стороны каждого из коммутаторов ПКС сети.

2. Анализ существующих подходов к прогнозированию нагрузки на контроллер

В настоящее время существует два основных подхода к прогнозированию временных рядов — это статистические методы и методы машинного обучения. В статистических методах функциональная зависимость между значениями ряда в каждый момент времени и внешними факторами задается аналитически. В методах машинного обучения, к которым относятся в том числе методы на основе нейронных сетей, характерной чертой является не прямое решение задачи, а построение прогноза на основе обучения на множестве решений схожих задач.

2.1 Применение модели ARIMA для прогнозирования нагрузки

В работе [4] для прогнозирования нагрузки на контроллер за основу взята модель ARIMA (Autoregression Integrated Moving Average), или интегрированная модель авторегрессии скользящего среднего, которая объединяет модель авторегрессии и модель скользящего среднего, и обобщается на случай нестационарного временного ряда. Среди достоинств авторегрессионных моделей в литературе называют их простоту и прозрачность моделирования, а также универсальность. Среди недостатков — большое количество параметров моделей (их определение трудоемко и обычно требует применения численных методов), линейность, не позволяющая проводить моделирование нелинейный процессов. Эти модели обычно используют для краткосрочного прогнозирования.

В работе был разработан метод прогнозирования нагрузки на контроллер в программно-конфигурируемых сетях с целью увеличения времени на предотвращение возможной перегрузки. В качестве входных данных модели используются временные ряды нагрузки Packet-in сообщений и временные ряды нагрузки всех OpenFlow сообщений со стороны каждого из коммутаторов. В качестве прогнозируемого объекта была выбрана оценка суммарной нагрузки на контроллер со стороны коммутаторов.

2.2 Применение нейронных сетей для прогнозирования нагрузки

В работе [5] для прогнозирования нагрузки за основу метода машинного обучения используется модель искуственной нейронной сети [6].

В качестве входных данных используются значения Packet-in сообщений с дополнительным подсчетом количество содержащихся в них байт, а также аппаратные значения ЦПУ и оперативной памяти среды, в которой запущен контроллер.

Связь между случайными величинами вычислялась с помощью ковариации и линейного коэффициента корреляция (коэффициента корреляции Пирсона).

В работе используется нейронная сеть, использующая обучение с учителем. Обучение с учителем предполагает, что для каждого входного вектора существует целевой вектор, представляющий собой требуемый выход. Также нейронная сеть является многослойной, где входной слой представляет из себя три параметра: количество Packet-in сообщений, количество байт и шаг дискретизации равный константе (1 секунде). А на выходном слое прогнозируются эти же параметры, исключая шаг дискретизации.

В результате работы была проведена оценка качества разработанного метода машинного обучения на основе искуственной нейронной сети с целью минимизации среднеквадратичной ошибки прогноза. Так MSE уже на 20 проходе на наборе данных выдает значение меньше $0.5*10^8$.

Таким образом, работа демонстрирует вариант проверки взаимосвязей двух случайных величин и работающий метод машинного обучения с использованием модели искуственной нейронной сети для прогнозирования нагрузки на контроллер программно-конфигуриуемых сетей.

2.3 Сравнительный анализ подходов

Для сравнительного анализа представленных подходов использовались также выводы, полученные в работе [7], в которой рассматриваются и сравниваются особенности представленных методов. Традиционные статистические методы более точны (при использовании их в прогнозировании), чем методы машинного обучения. Однако при сравнении статистических методов и методов машинного обучения, необходимо отметить, что результаты и качество методов могут быть связаны непосредственно с конкретным набором данных, на котором проводится исследование

методов. Поэтому на определенных задачах и при определенных исходных данных может получаться выигрыш по времени при использовании методов машшиного обучения по сравнению со статистическими методами. Поэтому в данной работе проводится исследование применения машинного обучения на основе нейронных сетей для задачи прогнозирования нагрузки на контроллер в программно-конфигурируемых сетях.

3. Разработка метода прогнозирования нагрузки на контроллер

Разработанный метод включает в себя два этапа:

- Этап 1: выявление зависимости между количеством запросов от коммутаторов на контроллер и показателями о загрузке процессора и утилизацией оперативной памяти;
- Этап 2: прогнозирование нагрузки на ПКС контроллер для некоторого фиксированного момента времени.

На этапе 1 для оценки зависимости между количеством запросов от коммутаторов на контроллер и показателями о загрузке сервера было предложено использовать многопараметрический корреляционный анализ. В данной работе зависимость двух случайных величин определяется по формуле коэффициента корреляции Пирсона:

$$r_{xy} = \frac{\sum_{i=1}^{i=N} \left(x_i - \bar{X}\right) \cdot \left(y_i - \bar{Y}\right)}{\sqrt{\sum_i \left(x_i - \bar{X}\right)^2 \cdot \sum_i \left(y_i - \bar{Y}\right)^2}}$$

, где X,Y — два временных ряда; x_i — значение переменной X; y_i — значение переменной Y; $\overline{X}$ — среднее арифметическое для переменной X; $\overline{Y}$ — среднее арифметическое для переменной Y.

X для исследуемой задачи это временной ряд из показателей CPU либо RAM загрузки сервера. A Y – это временной ряд из количество поступающих пакетов от коммутаторов на контроллер.

Коэффициент линейной корреляции r_{xy} в данной формуле меняется от -1 до 1 и характеризует значимость линейной зависимости между случайными величинами (параметрами):

- При $r_{xy} = \pm 1$, корреляционная связь представляет собой линейную функциональную зависимость;

- При $r_{xy} = 0$, линейная корреляционная связь отсутствует;
- Иначе, чем ближе значение $|r_{jk}|$ к единице, тем с большим основанием можно считать, что изучаемые величины находятся в линейной зависимости.

Второй этап прогнозирования нагрузки на контроллер ПКС основан на модели машинного обучения с использованием искуственной нейронной сети и состоит из следующих основных шагов:

- Выбор входных параметров нейронной сети;
- Выбор прогнозируемых параметров, т.е. выходных параметров нейронной сети;
- Выбор гиперпараметров для нейронной сети.

В данной задаче рассматривается один выходной параметр - это значение PacketCount(t) — суммарное количество Packet-in запросов, приходящих на контроллер от коммутаторов в дискретный момент времени t_0. Шаг интервала выбран равным 1 секунде.

Входными данными являются значения PacketCount(t), за момент времени $t < t_0$, где момент t_0 — дискретное время, для которого нужно спрогнозировать значение, и временной интервал TS, как разность времен двух ближайших измерений.

Рассмотрим гиперпараметры искуственной нейронной сети. Данные делятся на тренировочные и тестовые в общепринятом соотношении 8:2. Так как модель нейронной сети в начале не обучена, то ей нужно отдать на обучение часть датасета для вычисления весов нейронов. Далее обученной модели подаются входные параметры тестовых данных для того, чтобы она предсказала выходные значение. И только после этого происходит сравнение с выходными тестовыми данными для вычисления коэффициентов рассматриваемой модели.

Другим гиперпараметром служит количество скрытых слоев в нейронной сети, и сколько нейронов находится на каждом из них. Еще одним важным параметром является количество эпох, или сколько раз модель нейронной сети пройдет массив данных перед тем, как начать прогноз на тестовых значениях.

Другие не менее важные параметры нейронной сети — это метод оптимизации, количество значений в итерацию, скорость обучения и др.

Для каждого прогноза модели с фиксированными гиперпараметрами и набором данных сохраняются значения

следующих показателей (при поступлении соответствующих наблюдаемых значений):

- Среднеквадратичная ошибка MSE (Mean Squared Error);
- Абсолютная средняя ошибка MAE (Mean Absolute Error);
- Средняя абсолютная ошибка в процентах MAPE (Mean Absolute Percentage Error).

На основе этих показателей производится проверка качества модели искусственной нейронной сети с учетом специфики, берущейся из поставленной задачи и значений собранных данных.

4. Реализация

Для решения задачи реализовано приложение для работы с программно-конфигурируемой сетью под управлением контроллера RUNOS 2.0 [8], которое через REST API запрашивает у контроллера RUNOS статистические данные посекундно об общем количестве OpenFlow сообщений и количестве Packet-In запросов от коммутаторов. RUNOS — это отечественный распределенный контроллер ПКС/OpenFlow с открытым исходным кодом для корпоративных программно-конфигурируемых сетей, центров обработки данных и сетей ПКС профессиаонального уровня.

Программная реализация разработана на языке Python с использованием библиотек Pandas [10], scikit-learn [11] и Psutil [12].

Программная реализация состоит из трех основых модулей:

- **Модуль запроса данных с контроллера RUNOS.** Данный модуль представлен скриптом на языке Python, который реализует периодический опрос по REST API контроллера RUNOS с заданной частотой. Модуль также проводит периодический опрос сервера контроллера о текущих показателях утилизации процессора и оперативной памяти сервера (при помощи библиотеки Psutil). Результатом работы являются временные ряды суммарной нагрузки (количества запросов) на контроллер со стороны каждого из коммутаторов, общей нагрузки и утилизации вычислительных ресурсов сервера.
- **Модуль проверки взаимосвязи входных данных.** Данный модуль осуществляет оценку линейного коэффициента корреляции для входных данных, полученных

из модуля запроса данных. Результатом работы модуля являются два значения: линейный коэффициент корреляции между показателями PacketCount и утилизацией процессора CPU; линейный коэффициент корреляции между показателями PacketCount и утилизацией оперативной памяти RAM.

- **Модуль прогнозирования.** Модуль реализует модель искусственной нейронной сети с подбором оптимальных гиперпараметров и вычисляет метрики для уже обученной модели. Результатом является статистика по прогнозированию поступающих пакетов на контроллер, а также параметры обучения нейронной сети, на основе которых получается данный прогноз.

5. Экспериментальное исследование

Целью экспериментального исследования является проверка взаимосвязи входных данных и оценка качества работы разработанного метода прогнозирования нагрузки для контроллера RUNOS 2.0.

Исследование метода проводилось на экспериментальном стенде, представленном на рисунке 1.

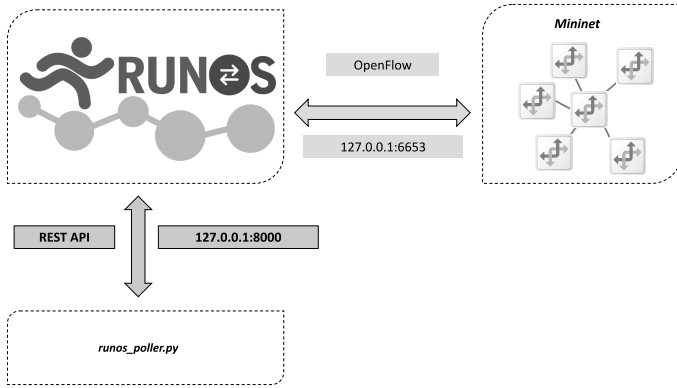


Рис. 1: Логическая схема экспериментального стенда.

Экспериментальный стенд включает следующие программные средства:

- **Контроллер RUNOS** версии 2.0. На контроллере было запущено приложение L2 LearningSwitch.
- **Модуль запроса данных с контроллера RUNOS (runos-poller).** Модуль осуществляет периодический запрос статистики контроллера RUNOS по его REST API интерфейсу.
- **Генерация трафика.** На каждом из хостов, создаваемых в сети мининет и подключенных к коммутаторам, производится ping на все другие хосты при помощи утилиты Mininet pingall.

В среде Mininet разворачивались две сети с древовидной топологией: из 6 коммутаторов и 25 хостов и из 20 коммутаторов и 361 хоста. Соответственно данные о загрузке контроллера RUNOS представляют собой 2 набора данных временных рядов с количеством наблюдений 2047 и 8327. Интервал мониторинга составляет 1 секунду.

По результатам двух экспериментов был вычислен линейный коэффициента корреляции (формула для r_{xy} из третьего раздела) для искомых пар. Для PacketCount и CPU линейный коэффициент корреляции был получен равным 0.905, а для PacketCount и RAM равным $8.1*10^{-17}$ (для первого набора данных). Значения показателя PacketCount и CPU позволяют сделать вывод о наличии взаимосвязи между принимаемыми Packet-in сообщениями контроллером и его аппаратной нагрузкой, второй набор также подтверждает этот вывод, выдавая результат равный 0.964. Однако значение показателя PacketCount и RAM показывает, что между параметрами нет явной линейной зависимости.

Таким образом, мы можем прогнозировать нагрузку на контроллер в ПКС, анализируя количество поступающих на него пакетов с коммутаторов.

В рамках экспериментальных проверок используется следующая последовательность действий:

1. **Выбор гиперпараметров метода.** На данном шаге фиксируются гиперпараметры модели искуственной нейронной сети
2. **Запуск программного средства.** На данном шаге осуществляются запуски программного средства с фиксированными ранее параметрами.
3. **Анализ результатов.** На данном шаге сравниваются качества прогноза запусков с различными параметрами и выбирается оптимальный.

После выполнения данного процесса программа предоставляет самый лучший (по показателю MSE) вариант из выданных ей параметров. Чтобы нейронная сеть лучше улавливала аномалии, перегрузки контроллера, поиск оптимального варианта осуществлялся по показателю MSE, где ошибка берется в квадрате, что и обеспечивает точность в нахождении отклонений во временных рядах.

Таким образом были получены следующие оптимальные параметры для фиксированного набора данных:

- **hidden_layer_sizes** = (400,100,27) - количество скрытых слоев с нейронами в модели;
- **alpha** = 0.00025 - скорость обучения нейронов при градиентном спуске;
- **max_iter** = 400 - количество пройденных эпох обучения;
- **tol** = 0.001, прекратить поиск минимума (или максимума) после достижения некоторого порога;
- **n_iter_no_change** = 5 - критерий останова процесса после заданного количества шагов без изменений весов;
- **solver** = adam - метод оптимизации Adam.

И для запущенной модели искуственной нейронной сети с данными параметрами (датасет из 2047 наблюдений) были получены следующие качественные показатели:

Показатель	Значение
MAE	0.020939361
MSE	0.005098226
MAPE	0.688270665

Для датасета из 8327 наблюдений:

Показатель	Значение
MAE	0.111236695
MSE	0.049814991

Оптимальной разбиение тренировочных значений к тестовым — 6:4 для набора из 8327 наблюдений.

Прогноз, получаемый разработанным методом прогнозирования нагрузки на контроллер, при рекомендованном выборе параметров, обеспечивает низкие показатели MSE и MAE, 0.005 и 0.02, соответсвенно, на одном из выбранных наборах данных.

Заключение

В данной работе был предложен и разработан подход к прогнозированию нагрузки на контроллер, создаваемой запросами от коммутаторов, с использованием метода машинного обучения на основе нейронной сети. В рамках исследования была показана зависимость между количество запросов к контроллеру и утилизацией оперативной памяти и производительности процессора.

В рамках работы было разработано приложение для мониторинга текущей загрузки контроллера RUNOS 2.0 через его REST API и краткосрочного прогнозирования нагрузки.

В ходе работы были достигнуты следующие результаты:

- Проведена проверка статической взаимосвязи входных данных.
- На основе результатов анализа подходов к прогнозированию нагрузки разработан метод решения поставленной задачи прогнозирования с использованием машинного обучения.
- Предложенный метод прогнозирования нагрузки на контроллер реализован для программно-конфигурируемой сети под управлением контроллера RUNOS.
- Разработана методика и проведено экспериментальное исследование метода, проанализированы полученные результаты.

Литература

1. Смелянский Р. Л. *Программно-конфигурируемые сети. Открытые системы.* М.: СУБД, No9, 2012.
2. Open Networking Foundation. Specification OpenFlow Version 1.3.0 `https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf`
3. Pashkov V., Smeliansky R. *On High Availability Distributed Control Plane for Software-Defined Networks.* // 2018 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC). – IEEE, 2018. – pp. 1-10.

4. Скобцова Ю. А., Пашков В. Н. *Исследование и разработка сетевого приложения для распределенной платформы управления в программно-конфигурируемых сетях.* // Программные системы и инструменты. — Т. 16 — Москва, МАКС Пресс, 2016. — с. 7–18.

5. Volkov A., Proshutinskiy K., Adam A. B. M., Ateya A. A., Muthanna A., Koucheryavy A. *SDN Load Prediction Algorithm Based on Artificial Intelligence.* M.: DCCN, 2019.

6. Гафаров Ф. М., Галимянов А. Ф. *Искуственные нейронные сети и их приложения.* М.: Издательство Казанского университета, 2018.

7. Makridakis S., Spiliotis E., Assimakopoulos V. *Statistical and Machine Learning forecasting methods: Concerns and ways forward.* M.: PLOS ONE, 2018.

8. Контроллер RUNOS 2.0. `https://github.com/ARCCN/runos`

9. Mininet `http://mininet.org/`

10. Pandas `https://pandas.pydata.org/`

11. scikit-learn `https://scikit-learn.org/stable/`

12. psutil `https://psutil.readthedocs.io/en/latest/`

Маркобородов А. А., Волканов Д. Ю.

Трансляция групповой таблицы коммутатора ПКС в язык ассемблера сетевого процессора RuNPU[1]

Введение

В настоящее время продолжается активное развитие технологии программно-конфигурируемых сетей (ПКС) [1], [2]. В ПКС сетевые устройства (коммутаторы) выполняют функцию передачи данных, а функцию управления устройствами и потоками данных выполняет специальное программное обеспечение (контроллер). Для взаимодействия контроллера и коммутаторов используются специальные протоколы управления, такие как протокол OpenFlow [3].

В статье рассматривается коммутатор на базе программируемого сетевого процессора RuNPU, функционирующий под управлением протокола OpenFlow 1.3 [3]. Для обработки пакетов в соответствии с протоколом OpenFlow необходимо производить трансляцию абстракций протокола OpenFlow в программу на языке ассемблера процессора RuNPU. Правила обработки пакетов в протоколе OpenFlow задаются с использованием двух основных абстракций: таблицы потоков и групповой таблицы. Ранее уже была рассмотрена задача трансляции таблицы потоков [4]. Настоящая статья посвящена исследованию методов трансляции групповой таблицы OpenFlow.

Поскольку содержимое групповой таблицы постоянно обновляется контроллером (добавляются и удаляются группы действий), возникает необходимость получения машинного кода для обновленной групповой таблицы. Учитывая частоту обновления программы обработки пакетов, целесообразно поддерживать промежуточное представление групповой таблицы для её трансляции после обновления. Каждый раз при обновлении правил обработки пакетов выполняется трансляция промежуточного представления групповой таблицы в программу на языке ассемблера, и машинный код, полученный из этой программы, загружается в сетевой процессор.

Статья имеет следующую структуру. В разделе 1 приводится описание групповой таблицы протокола OpenFlow. В разделе 2

[1]Работа выполнена при частичной поддержке РФФИ, грант № 19-07-01076.

описана архитектура сетевого процессора RuNPU. Раздел 3 содержит постановку решаемой задачи. В разделе 4 представлены разработанные методы трансляции групповой таблицы в язык ассемблера сетевого процессора RuNPU. Раздел 5 посвящен экспериментальному исследованию и анализу его результатов.

1. Групповая таблица OpenFlow

Спецификация протокола OpenFlow [3] содержит описание абстракций, в терминах которых контроллер управляет коммутатором путем обмена OpenFlow сообщениями. При этом спецификация протокола не накладывает требований на реализацию коммутатором предлагаемых абстракций, задавая только логику обработки пакетов.

Коммутатор использует набор таблиц потоков для классификации получаемых пакетов. Классификация пакета — это определение номера порта (портов), на который необходимо отправить пакет, и действий над заголовком пакета, которые необходимо выполнить перед отправкой пакета. Одним из действий может быть отправка пакета на дополнительную обработку в определенную группу действий групповой таблицы.

Групповая таблица коммутатора — это абстракция протокола OpenFlow, предназначенная для группирования нескольких потоков с одинаковыми наборами действий. Данная таблица также позволяет описывать дополнительные способы обработки пакетов с выборочным выполнением наборов действий или с применением нескольких наборов действий.

Групповая таблица состоит из записей, называемых *группами действий*. Каждая запись групповой таблицы содержит номер, тип и упорядоченный список бакетов. *Бакет (англ. bucket)* представляет собой набор действий, которые применяются к пакетам при обработке в группе действий. Номер является уникальным идентификатором группы действий, а тип определяет способ обработки пакета. Каждый из бакетов обычно содержит модифицирующие заголовок пакета действия (*Set Field*, *Push/Pop Tag* и др.) и действие отправки пакета на определенный выходной порт (*Output*).

Протокол OpenFlow определяет четыре типа групп действий: *Indirect*, *All*, *Select* и *Fast Failover*. С точки зрения обработки пакета группа действий *Indirect* является частным случаем группы *All* с одним бакетом. Группа действий *Fast Failover* аналогична группе действий *Indirect* после проведения проверки активности

используемых в бакетах выходных портов. Наиболее сложной является обработка пакета в группе типа *All*, так как в ней требуется независимое применение действий каждого бакета, и на выходе получается несколько пакетов с различными заголовками.

Таким образом, программа обработки пакетов RuNPU должна позволять перенаправлять пакеты на обработку в группы действий, для которых может потребоваться отправка нескольких пакетов с различными заголовками.

2. Сетевой процессор RuNPU

Сетевой процессор RuNPU содержит по одному вычислительному конвейеру для каждого порта (рисунок 1). Основная обработка заголовка пакета осуществляется на восьми одинаковых стадиях конвейера, исполняющих загруженный в них машинный код [5].

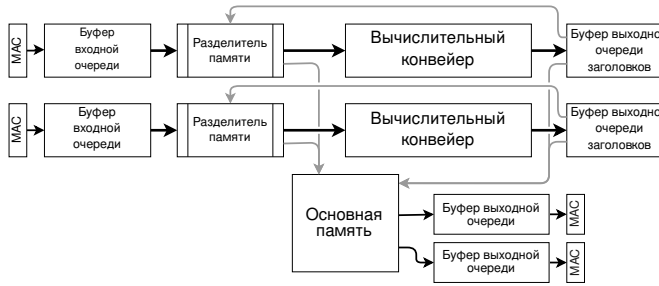


Рис. 1: Сетевой процессор RuNPU

Стадия вычислительного конвейера содержит единственный регистр общего назначения длиной 128 бит (*регистр-аккумулятор*), который выступает в качестве регистра-операнда и/или регистра-результата команды.

Память, доступ к которой осуществляется через команды, разделена на две части: *область заголовка пакета* и *область метаданных* (размер заголовка, номер входного порта, маска выходных портов, 16 байт пользовательских метаданных и др.).

Язык ассемблера, на котором описывается программа для стадий конвейера, содержит команды чтения из памяти и записи в память, арифметические и логические операции, битовые сдвиги, безусловные и условные переходы на метку. Язык ассемблера сетевого процессора RuNPU также поддерживает специальные

команды для вставки деревьев поиска по точному совпадению (`tree_in`) и LPM-поиска (`tree_lpm`).

На программу на языке ассемблера сетевого процессора RuNPU накладываются следующие ограничения:

- Максимальная битовая длина сравниваемого значения в командах условного перехода и файлах деревьев поиска равна 64.
- Метки переходов, содержащиеся в командах и файлах деревьев поиска, должны встречаться в программе после команды или директивы соответственно. Это требование обусловлено архитектурой вычислительного конвейера сетевого процессора RuNPU, в котором не допускается переход к командам, записанным в памяти по меньшему адресу, чем адрес текущей исполняемой команды.

Важной особенностью также является отсутствие отдельной области памяти для данных программы. Данные задаются в виде аргументов команд и являются неотделимой частью машинных команд.

Конвейеры сетевого процессора RuNPU также поддерживают механизм отправки заголовка пакета на повторную обработку (далее — *разворот*). В этом случае при первом прохождении конвейера необходимо установить специальный бит в метаданных пакета, сигнализирующий о необходимости отправки заголовка на разворот. В случае выставления данного бита буфер выходной очереди передаст пакет обратно в начало конвейера.

Таким образом, при реализации групповой таблицы на RuNPU возможно использовать механизм разворота для отправки пакетов с различными заголовками, а для сохранения значений после разворота может быть использована область пользовательских метаданных.

3. Постановка задачи

В работе необходимо разработать алгоритм трансляции групповой таблицы OpenFlow, описанной в разделе 1, в программу на языке ассемблера сетевого процессора RuNPU, архитектура которого рассмотрена в разделе 2. Входными данными алгоритма являются групповая таблица, содержащая n групп действий, и $mlen$ — количество байт пользовательских метаданных, доступных для хранения вре́менных значений в программе.

Необходимость параметра $mlen$ обоснована тем, что пользовательские метаданные одновременно могут использоваться для трансляции не только групповой таблицы, но и для таблиц потоков, поэтому значение $mlen$ зависит от используемых в программе абстракций OpenFlow.

Получаемая алгоритмом программа должна использовать не более $mlen$ байт пользовательских метаданных и содержать метку для каждой группы действий, после перехода на которую заголовок пакета будет модифицирован в соответствии с наборами действий этой группы действий и отправлен на указанные в них порты.

4. Методы трансляции групповой таблицы

Обозначим V_{all} *множество допустимых значений заголовков.* Будем называть *действием* пару $A = (a, t)$, состоящую из отображения $a : V_{all} \to V_{all}$ и *порядкового числа* $t \in \mathbb{N}$, определяющего взаимный порядок применения нескольких действий бакета к заголовку пакета.

Бакетом назовем упорядоченный кортеж действий $B = (A_1, A_2, \ldots, A_k)$ такой, что для $\forall A_i = (a_i, t_i), \forall A_j = (a_j, t_j), i < j$ верно, что $t_i \leq t_j$. *Модификацией заголовка бакетом* $B = (A_1, A_2, \ldots, A_k)$ назовем отображение $b : V_{all} \to V_{all}$, являющееся композицией отображений действий бакета, то есть $b = a_k \circ a_{k-1} \circ \ldots \circ a_1$.

Будем называть *группой действий* пару $g = (i, B)$, состоящую из *номера* $i \in \mathbb{N}$ и *множества бакетов* $B = \{B_1, B_2, \ldots B_m\}$.

Групповой таблицей назовем множество групп действий $G = \{g_1, g_2, \ldots, g_n\}$ таких, что для $\forall g_r = (i_r, t_r, B_r), \forall g_s = (i_s, t_s, B_s), r \neq s$ верно, что $i_r \neq i_s$.

Трансляция групповой таблицы состоит из двух стадий: *построения структуры данных промежуточного представления групповой таблицы* и *трансляции структуры данных в программу на языке ассемблера.* В качестве промежуточного представления групповой таблицы используется *модифицированная групповая таблица*, бакеты которой могут содержать не только действия, определенные в протоколе OpenFlow, но и обратные к ним действия, а также действия сохранения значения поля заголовка пакета в области метаданных.

Определим обратное действие как обратное отображение $a^{-1} : V_{all} \to V_{all}$, которое по заголовку, полученному после

применения действия A, восстанавливает исходный заголовок. Достаточно определить следующие обратные действия:

- Вставка тега VLAN (*Push VLAN*).
- Снятие тега VLAN (*Pop VLAN*).
- Увеличение TTL (*Increment TTL*).
- Установка исходного значения val_0 поля f (*Set Field*).

При выполнении действия установки исходного значения val_0 поля f требуется значение val_0. Оно считывается из области пользовательских метаданных, куда предварительно записывается действием сохранения значения поля f в области метаданных (*Store Field*).

Предлагается три метода добавления группы действий в модифицированную групповую таблицу:

1. *Трансляция с полной отменой действий.* Во все бакеты, кроме первого, добавляются обратные действия для всех действий предыдущего бакета, что позволяет вернуть заголовок и его значение к исходному виду. Далее во все бакеты, кроме последнего, добавляются действия сохранения значений полей заголовка, которые будут изменены данным бакетом. После этого добавляются действия исходных бакетов группы действий.

2. *Трансляция с отменой части действий.* В отличие от предыдущего метода обратные действия установки значения поля добавляются только для тех полей предыдущего бакета, которые не изменяются в рассматриваемом. Действия сохранения полей в области метаданных также добавляются только для оставшихся обратных действий в следующих бакетах.

3. *Трансляция с выбором порядка бакетов.* Сначала производится поиск перестановки бакетов группы действий, для которой может быть построена программа, удовлетворяющая ограничению на размер пользовательских метаданных $mlen$. Далее для выбранной перестановки бакетов применяется второй метод.

Рассмотрим подробнее алгоритм добавления группы действий в модифицированную групповую таблицу с отменой части действий. Остальные методы — это либо упрощение данного метода (отмена всех действий), либо применение данного метода при различном порядке бакетов (выбор порядка бакетов).

4.1 Трансляция с отменой части действий

Пусть $F(B)$ — это множество полей исходного заголовка, значения которых требуется сохранить для восстановления после применения действий бакета $B = (A_1, A_2, \ldots, A_k)$ в случае, если какой-либо из следующих бакетов не изменяет данное поле. Для построения множества $F(B)$ из $F(B) = \emptyset$ необходимо для каждого действия $A \in B$ выполнить следующее:

1. Если A — это действие снятия тега VLAN, то $F(B) := F(B) \cup (F(A) \backslash \{tpid\})$ (значение поля *tpid* определяется однозначно по другим полям заголовка [3]).

2. Если A — это действие установки значения для поля из тега VLAN (*pcp* или *vid*), и если:

   (a) бакет B содержит действие снятия тега VLAN, но не содержит действие вставки тега VLAN, то $F(B) := F(B) \cup \{pcp\_inner\}$ или $F(B) := F(B) \cup \{vid\_inner\}$ соответственно для полей *pcp* и *vid*. Поля *pcp_inner* и *vid_inner* обозначают поля внутреннего тега VLAN.

   (b) бакет B не содержит действий снятия и вставки тега VLAN, то $F(B) := F(B) \cup F(A)$.

3. Если A — это действие установки значения для поля не из тега VLAN, то $F(B) := F(B) \cup F(A)$.

Пусть S — множество полей заголовка, сохраненных в пользовательских метаданных. Для добавления группы действий $g = (i, \{B_1, B_2, \ldots, B_m\})$ в модифицированную групповую таблицу G' необходимо добавить группу действий $g' = (i, \emptyset)$ в G' и выполнить следующую последовательность шагов для каждого бакета $B_j, j = \overline{1, m}$.

1. *Создание бакета.* Добавить в группу действий g' бакет $B'_j = \emptyset$.

2. *Добавление обратных действий.* Если $j \neq 1$, то:

   (a) если B_{j-1} содержит действие снятия тега VLAN, но не содержит действие вставки тега VLAN, то добавить в B'_j обратное действие *Push VLAN*;

   (b) если B_{j-1} содержит действие вставки тега VLAN, но не содержит действие снятия тега VLAN, то добавить в B'_j обратное действие *Pop VLAN*;

(c) если B_{j-1} содержит действие уменьшения TTL, и $ip\_ttl \notin F(B_{j-1})$, то добавить в B_j обратное действие *Increment TTL*;

(d) для каждого поля $f \in S$ добавить в B'_j обратное действие *Set Field*.

3. *Добавление действий сохранения значений полей.* Если $j \neq m$, то:

(a) $S := \emptyset$, и добавить в S те поля $f \in F(B_j)$, для которых $\exists B_{j'}, j' > j$ и $f \notin F(B_{j'})$, то есть $S := \{f | f \in F(B_j) \setminus \bigcap_{j'=j+1}^{m} F(B_{j'})\}$;

(b) если $\sum_{f \in S} len(f) > mlen$, то удалить группу действий g' из G' и завершить добавление группы действий g с ошибкой;

(c) для каждого поля $f \in S$ добавить в B'_j действие *Store Field*.

4. *Добавление действий исходного бакета.* Добавить в B'_j действия из B_j.

4.2 Трансляция модифицированной групповой таблицы в язык ассемблера RuNPU

Все три метода добавления группы действий позволяют построить модифицированную групповую таблицу. Трансляция модифицированной групповой таблицы в граф программы состоит из двух стадий: *добавление вспомогательных блоков* и *добавление блоков для групп действий*.

На первой стадии в граф программы добавляются следующие вершины (не раскрашенные на рисунке 2 вершины).

- *Начальная вершина Start* выполняет проверку значения бита разворота в метаданных пакета.
- *Дерево поиска по метке возврата*, записанной в метаданных пакета, выполняет переход на соответствующую метку возврата для продолжения обработки пакета после разворота.
- *Установка бита разворота* устанавливает бит разворота и завершает обработку пакета.
- *Конечная вершина End*.

На второй стадии для каждой группы действий из модифицированной групповой таблицы добавляются *вершины действий*. Вершины действий одного бакета последовательно соединены дугами. Для каждого бакета, кроме первых, соответствует дуга из дерева поиска по метке возврата. Также в конец цепочки вершин каждого бакета, кроме последних, добавляется *вершина записи метки* следующего бакета.

На рисунке 2 приведен пример графа программы для групповой таблицы, состоящей из двух групп действий. Первая группа действий содержит бакеты $(A_{1,1}, A_{1,2}, \ldots, A_{1,n1})$ и $(A_{1,n1+1}, A_{1,2}, \ldots, A_{1,n1+n2})$. Вторая группа действий содержит бакеты $(A_{2,1}, A_{2,2}, \ldots, A_{2,k1})$ и $(A_{2,k1+1}, A_{2,2}, \ldots, A_{2,k1+k2})$. При трансляции в граф программы, помимо вспомогательных блоков, для каждой группы действий было добавлено по две цепочки вершин. Светло-серым обозначены вершины, соответствующие первой группе действий, а темно-серым — второй.

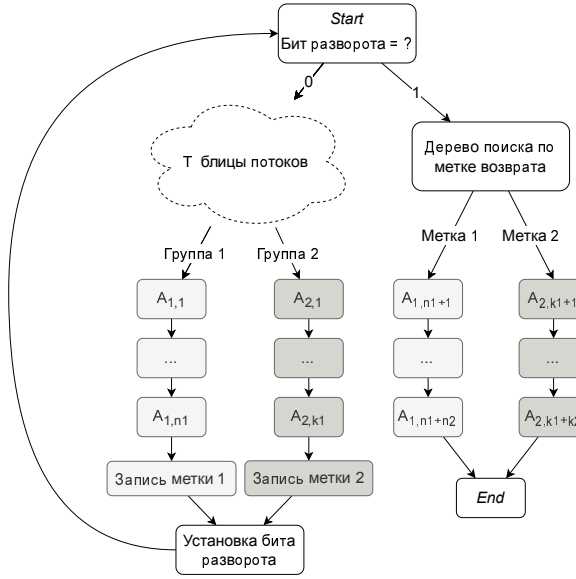


Рис. 2: Пример графа программы

Рассмотренные в данной главе методы позволяют преобразовать исходную групповую таблицу в модифицированную групповую таблицу, для которой затем применяется алгоритм трансляции в программу RuNPU. Такая модифицированная таблица может поддерживаться управляющим ПО коммутатора в качестве

промежуточного представления для быстрого внесения изменений в групповую таблицу.

5. Экспериментальное исследование

Разработанные методы были реализованы на языках C (основные алгоритмы) и C++ (вспомогательные функции ввода/вывода) с использованием стандартных библиотек. Система трансляции групповой таблицы была интегрирована в систему трансляции таблиц потоков, разработанную ранее в работе [4].

Целью экспериментального исследования является оценка на различных входных данных следующих характеристик разработанной системы трансляции групповой таблицы:

- доля успешно добавленных групп действий в промежуточное представление групповой таблицы (без возникновения ошибки в связи с нехваткой области метаданных);
- среднее время добавления группы действий в промежуточное представление групповой таблицы;
- средний объем памяти, необходимый для хранения бинарного кода программы сетевого процессора RuNPU, выполняющей обработку по одной группе действий.

5.1 Методика экспериментального исследования

В ходе экспериментального исследования использовалась следующая последовательность действий:

1. *Выбор параметров экспериментальных данных* (количество групп действий, количество бакетов в группах действий, количество и типы действий в бакетах, размер пользовательских метаданных $mlen$).
2. *Генерация групповой таблицы.* Для выбранных параметров создаются экспериментальные наборы групп действий для различных сценариев обработки пакетов [6].
3. *Оценка исследуемых величин.* Производится запуск разработанной системы трансляции с целью получения соответствующей программы на языке ассемблера. Затем выполняется запуск имитационной модели сетевого процессора RuNPU на полученной программе. При

исследовании времени добавления группы действий выполняется несколько последовательных вызовов функции добавления группы действий в модифицированную групповую таблицу с измерением времени выполнения данной функции. Далее значение усредняется. Время измерялось средствами стандартной библиотеки языка C++ на процессоре Intel Core i5-1035G1.

5.2 Анализ результатов

Для обозначения типа экспериментальной групповой таблицы используются три числа m-v-k, где m — количество бакетов в группе действий, k — количество действий в бакете, а v равно 1, если бакеты содержат действия вставки или снятия тега VLAN, и равно 0 иначе. Метод 1 соответствует трансляции с полной отменой действий, метод 2 — трансляции с отменой части действий, и метод 3 — трансляции с выбором порядка бакетов.

На рисунке 3 изображены значения доли успешно добавленных в модифицированную групповую таблицу групп действий для различных экспериментальных групповых таблиц, методов трансляции и размера метаданных *mlen* байт. Доля 100% успешно добавленных групп действий была достигнута только методом 3 при $mlen = 16$ байт. Метод 2 добавлял 100% групп действий на большинстве рассмотренных групповых таблиц при $mlen = 16$. Метод 1 добавлял 100% групп действий только на половине рассмотренных групповых таблиц при $mlen = 16$. При $mlen = 8$ ни один метод не добавлял 100% групп действий.

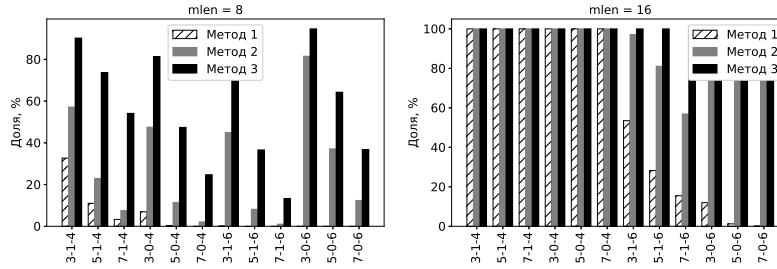


Рис. 3: Доля успешно добавленных групп действий

На рисунке 4 изображены значения среднего времени добавления группы действий в модифицированную групповую таблицу для различных экспериментальных групповых таблиц,

методов трансляции и размера метаданных *mlen* байт. Метод 3 имеет наибольшее время добавления группы действий в модифицированную групповую таблицу. Наименьшее время добавления показал метод 1. Это связано с минимальным количеством операций в методе 1 и перебором перестановок бакетов в методе 3.

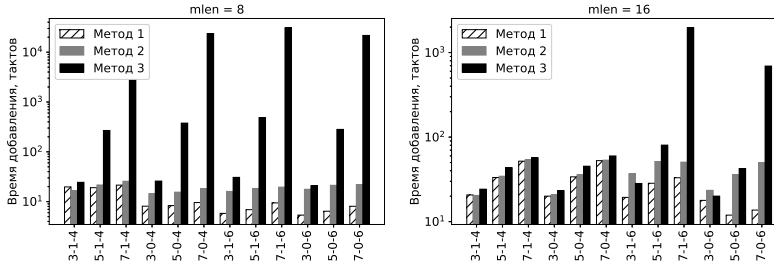


Рис. 4: Среднее время добавления группы действий

На рисунке 5 изображены значения среднего объема памяти вычислительного конвейера сетевого процессора RuNPU, необходимого для трансляции одной группы действий для различных экспериментальных групповых таблиц, методов трансляции и размера метаданных *mlen* байт. Наибольший объем памяти занимают группы действий с большим числом бакетов и большим числом действий, а также с бакетами, содержащими действия снятия или вставки тега VLAN. При этом все три метода требуют примерно одинаковый объем памяти на одних и тех же групповых таблицах.

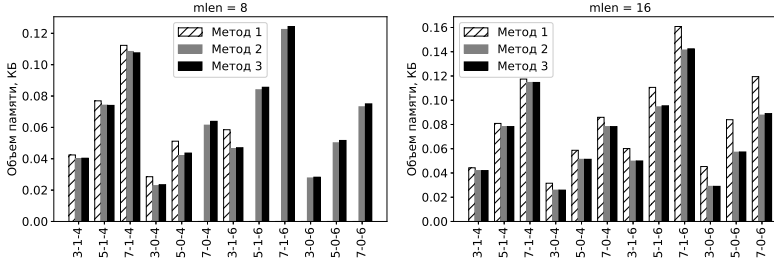


Рис. 5: Средний размер памяти программы на группу действий

На большем количестве групп действий требование по размеру пользовательских метаданных смогли удовлетворить методы с

отменой части действий и выбором порядка бакетов. Однако время добавления группы действий методом с выбором порядка бакетов в некоторых случаях было сильно больше среднего значения.

Заключение

В статье были предложены три метода трансляции групповой таблицы OpenFlow в язык ассемблера сетевого процессора RuNPU с использованием промежуточного представления в виде модифицированной групповой таблицы. Соответствующие алгоритмы были интегрированы в транслятор таблиц потоков OpenFlow.

В ходе экспериментального исследования было выяснено, что метод трансляции с выбором порядка бакетов позволяет успешно транслировать в программу на языке ассемблера сетевого процессора RuNPU наибольшее число групп, однако в среднем требует больше времени на добавление группы действий. Метод трансляции с отменой части действий также успешно добавляет группы действий для большинства экспериментальных групповых таблиц и имеет меньшее время добавления.

В качестве направлений дальнейших исследований можно рассмотреть разработку методов трансляции групп действий, направляющих пакет на обработку в еще одну группу действий вместо отправки на порт.

Литература

1. Open Networking Foundation. *Software-defined networking: the new norm for networks.* ONF white paper, 2012.

2. Смелянский Р. Л. *Программно-конфигурируемые сети.* Открытые системы, № 9, с. 15–26, 2012.

3. Open Networking Foundation. *OpenFlow Switch Specification Version 1.3.0 (Wire Protocol 0x04).* 2012.

4. Markoborodov A., Skobtsova Y., Volkanov D. *Representation of the OpenFlow Switch Flow Table.* International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC), Moscow, Russia, p. 97–103, 2020.

5. Волканов Д. Ю., Маркобородов А. А. *Исследование ячейки конвейера сетевого процессорного устройства на модели уровня регистровых передач.* Программные системы и инструменты. Тематический сборник № 19, Москва, с. 30–41, 2019.

6. Беззубцев С.О., Васин В.В., Волканов Д.Ю., Жайлауова Ш.Р., Мирошник В.А., Скобцова Ю.А., Смелянский Р.Л. *Об одном подходе к построению сетевого процессорного устройства.* Моделирование и анализ информационных систем, 26(1), с. 39–62, 2019.

Никифоров Н. Н., Волканов Д. Ю.

Исследование применимости алгоритмов сжатия данных к таблицам классификации в сетевом процессорном устройстве [1]

Введение

В настоящее время наблюдается активное развитие технологии программно-конфигурируемых сетей (ПКС) [1]. Для работы ПКС требуются высокопроизводительные коммутаторы, которые выполняют функцию передачи данных. Возникает задача разработки программируемого сетевого процессорного устройства (СПУ) [2], являющегося основным функциональным элементом коммутаторов. В работе рассматривается коммутатор функционирующий под управлением протокола OpenFlow [3]. OpenFlow — один из наиболее распространенных протоколов для управления сетевым коммутатором ПКС. В этой статье рассматривается OpenFlow версии 1.3 [3], правила обработки пакетов в котором представляются в виде таблиц классификации (таблицы потоков в терминах протокола OpenFlow). В данной работе не рассматриваются групповые таблицы OpenFlow. В СПУ таблицы потоков представляются в виде программы обработки заголовков сетевых пакетов на языке ассемблера, получаемой с помощью транслятора таблиц потоков. Таблица потоков — это набор правил, определенных протоколом OpenFlow. Каждое правило содержит поле признака, набор битовых строк, по которым пакет может быть идентифицирован, и набор действий, которые СПУ выполняет с этим пакетом.

Данная работа посвящена разработке алгоритмов сжатия данных [4] для применения в трансляторе таблиц потоков в рамках рассматриваемой архитектуры сетевого процессорного устройства. Поскольку современные таблицы потоков занимают до нескольких десятков мегабайтов памяти [5], то возникает задача сжатия таблиц потоков, для использования рассматриваемого СПУ в коммутаторах ПКС.

В разделе 1 приводится описание предметной области, в разделе 2 ставится главная задача работы, в разделе 3 описываются алгоритмы сжатия, в разделе 4 описываются предложенные

[1] Работа выполнена при частичной поддержке РФФИ, грант № 19-07-01076

изменения, в разделе 5 приводится описание и результаты экспериментального исследования.

1. Описание предметной области

1.1 Протокол OpenFlow

В данной работе рассматривается протокол OpenFlow 1.3 [3] – один из наиболее распространённых протоколов для управления коммутаторов в ПКС сетях.

В спецификации протокола OpenFlow описываются абстракции: поток, признак, правило, действия, таблица потоков и т.д. С помощью этих абстракций описываются протоколы сетевого взаимодействия, а также описываются команды по средствам которых контроллер управляет коммутатором. В протоколе OpenFlow отсутствуют ограничения на реализацию команд в коммутаторе.

Для процесса классификации сетевых пакетов коммутатором используются таблицы потоков, представляющие из себя набор правил. Каждое правило состоит из набора признаков и набора действий. Процесс классификации является процессом определения номера исходящего порта коммутатора, а также набора действий, которые необходимо применить к заголовку пакета перед его отправкой.

1.2 Архитектура сетевого процессора (RuNPU)

В СПУ используется конвейерная архитектура, каждый конвейер состоит из 10 вычислительных блоков. Вычислительный блок – это набор более низкоуровневых RISC ядер, которые в данной работе не рассматриваются. Каждый вычислительный блок имеет доступ к участку памяти, в котором располагаются микрокод и данные (рис. 1). Существует ограничение на количество тактов, которое один пакет может обрабатываться на вычислительном блоке, в данной работе оно соответствует 25 тактам. Данное ограничение обусловлено требованием к производительности сетевого процессора, а именно каждый сетевой пакет должен обрабатываться фиксированное количество времени. Также один вычислительный блок имеет доступ к 64 килобайтам памяти. Из-за особенностей микроархитектуры, отсутствует отдельная область памяти, в которой хранятся данные. Поэтому микрокод содержит в себе все данные, необходимые для классификации пакетов.

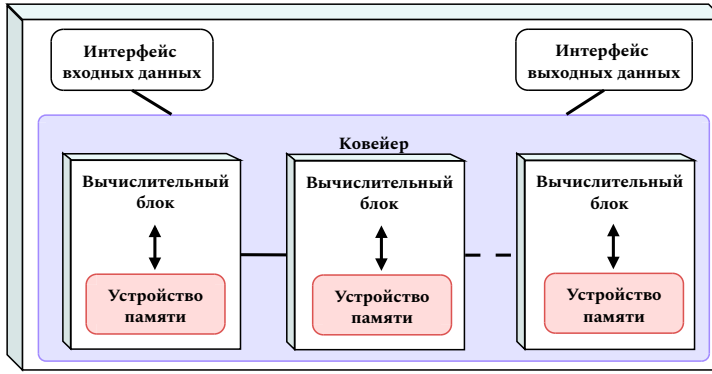


Рис. 1: Архитектура рассматриваемого сетевого процессора

На программу на языке ассемблера рассматриваемого сетевого процессора накладываются ограничения на максимальную битовую длину операнда в командах условного перехода и файлах деревьев поиска, которая равна 64 битам. Также метки переходов, содержащиеся в командах и файлах деревьев поиска, должны встречаться в программе после команды или директивы соответственно. Это требование обусловлено архитектурой вычислительного конвейера сетевого процессора, в котором не допускается переход к командам, записанным в памяти по меньшему адресу, чем адрес текущей исполняемой команды.

1.3 Система трансляции таблиц потоков

Для получения программы на языке ассемблера из таблицы потоков используется система трансляции таблиц потоков. Входными данными для системы трансляции таблиц потоков являются: конфигурация таблицы потоков, таблица потоков, конфигурация трансляции. После трансляции на выходе находится программа на языке ассемблера, которая в дальнейшем используется для обработки сетевых пакетов на СПУ.

2. Постановка задачи

Рассматривается коммутатор, построенный на базе сетевого процессора (RuNPU). Для управления коммутатором используется протокол OpenFlow. Сетевой процессор работает под управлением программы на языке ассемблера, которая получается путём трансляции таблиц потоков OpenFlow.

Пусть имеется таблица потоков OpenFlow, содержащая в себе правила с признаками точного совпадения и маскирующие признаки. Данная таблица транслируется в программу обработки сетевых пакетов на языке ассемблера с помощью системы трансляции таблиц потоков на ЦПУ коммутатора. СПУ выполняет классификацию полученного пакета по полученной программе на языке ассемблера. Необходимо разработать алгоритм сжатия таблиц потоков, который должен удовлетворять следующим условиям:

- программа полученная путём трансляции сжатой таблицы потоков должна занимать меньше памяти, чем программа полученная из исходной таблицы потоков;
- программа получаемая из сжатой таблицы потоков должна быть эквивалентна программе получаемой из исходной таблицы потоков;
- для трансляции сжатой таблицы потоков не должна использоваться декомпрессия.

2.1 Формальная постановка задачи

Введём формализацию OpenFlow таблиц. Упорядоченное множество всех рассматриваемых признаков в правилах обозначим $I = \{m_1, m_2, \ldots, m_k\}$. Каждый признак m_i из множества признаков I характеризуется битовой строкой, некоторой длины $m_i \in \{0, 1, *\}_i^W$, в данном случае символ $*$ обозначает любой бит. При этом, если $\exists m_i^j \in m_i$, такое, что $m_i^j = *$, то для $\forall m_i^k$, где $k > j$, то $m_i^k = *$. Длиной признака обозначим $len(m_i) = W_i$

Представим таблицу потоков в виде множества правил $R = \{r_1, r_2, \ldots, r_n\}$. С каждым правилом r_i связаны:

- номер i;
- приоритет $p_i \in Z_+$;

- вектор значений признаков $f_i = \{f_i^1, f_i^2, \ldots, f_i^k\}$, где f_i^j соответствует значению признака $m_j \in I$.

- набор действий, $A_i = \{a_1, a_2, \ldots, a_z\}$, который определяет дальнейшие действия сетевого процессора над пакетом.

Будем говорить, что заголовок пакета x и его метаданные с вектором значений признаков $g = \{g^1, g^2, \ldots, g^k\}$ (далее $x \to g$), **соответствуют правилу** $r_i \in R$ с вектором значений признаков $f_i = \{f_i^1, f_i^2, \ldots, f_i^k\}$ и приоритетом p_i (правило $r_i \in R$ идентифицирует пакет с вектором значений признаков g), если:

1. вектор значений признаков g соответствует вектору значений признаков f_i, то есть $\forall g_i \in g$, $len(g_i) = len(f_i)$. И $\forall f_i^{lj} \in f_i^l$, где l - индекс бита, $f_i^{lj} \in \{*, g^{lj}\}$, $l = \overline{1,k}$;

2. приоритет p_i максимален среди всех правил $r_j \in R$, для которых g соответствует вектору значений признаков f_j.

Под мощностью множества понимается количество элементов в рассматриваемом множестве. Множество R также должно удовлетворять следующему ограничению. Для любых двух правил $r_i, r_j \in R, r_i \neq r_j$, если существует набор значений признаков, который соответствует обоим правилам, то $p_i \neq p_j$. Например, правила с векторами значений признаков $f_i = \{110, 011, 1*\}$ и $f_j = \{11*, 011, 11\}$ должны иметь разный приоритет, так как набор значений признаков $g = \{110, 011, 11\}$ соответствует обоим правилам.

Введём функцию **идентификации** заголовка пакета $x \to g$ в таблице потоков R и обозначим её $R(x)$. Функция идентификации заголовка возвращает набор действий, соответствующий правилу, идентифицирующему заголовок пакета $x \to g$. Таким образом $R(x) = A_{r_i}$, где A_{r_i} набор действий правила $r_i \in R$.

Введём понятие **аналогичности** множеств R_1 и R_2. Множество R_1 аналогично множеству R_2, если для любого заголовка пакета, для которого существует идентифицирующее его правило $r_i \in R_1$, найдётся правило идентифицирующее его в множестве $r_j \in R_2$, при этом $A_i = A_j$.

Введём операцию **последнего** значащего бита признака $last(m_i) = j$, где $m_i^j \in \{0, 1\}$ и $m_i^{(j+1)} = *$. Назовём правила $r_i \in R$ и $r_j \in R$ **похожими**, если для $\forall u \in len(f_i)$ верно, что $last(f_i^u) = last(f_j^u) = l$, при этом $f_i^{ul} \neq f_j^{ul}$, и $A_i = A_j$.

Дано

Множество признаков I и таблица потоков R_1 с правилами, содержащими признаки из I.

Необходимо

Разработать алгоритм сжатия таблиц потоков, который будет переводить исходное множество – R_1, соответствующее исходной таблице потоков, в новое множество R_2, которое соответствует новой таблице потоков. Множество R_1 должно быть **аналогично** множеству R_2. Мощность множества R_2 должна быть меньше либо равна мощности множества R_1.

3. Алгоритмы сжатия

Был проведён обзор существующих решений, по результатам которого для дальнейшей реализации были выбраны алгоритмы оптимального кеширования и рекурсивного сокращения. Критериями обзора выступали: сложность построения, степень сжатия, необходимость использования внешней памяти, возможность использования сжатых данных без декомпрессии.

3.1 Классические алгоритмы сжатия

Под классическими алгоритмами сжатия будем понимать алгоритмы, которые сжатые данные представляют в бинарном виде [6]. Данные алгоритмы делятся на две большие группы: Алгоритмы сжатия без потерь — наиболее простые алгоритмы сжатия, делятся на несколько категорий (рис. 2). Алгоритмы сжатия с потерями — особенность данных алгоритмов в том, что исходные данные не могут быть однозначно получены из сжатых. Данная особенность позволяет более хорошую степень сжатия, чем у алгоритмов сжатия без потерь. Также данная особенность ограничивает область использования таких алгоритмов и поэтому они не будут рассматриваться в рамках данной работы. Среди алгоритмов сжатия без потерь можно выделить: статические алгоритмы Шенона-Фано, алгоритм Хаффмана, алгоритм арифметического кодирования, алгоритм RLE, алгоритм KWE; словарные и словарно-статистические алгоритмы LZ, LZW. Среди алгоритмов сжатия с потерями можно упомянуть такие алгоритмы как: алгоритм JPEG, семейство алгоритмов H.26x, семейство алгоритмов MPEG.

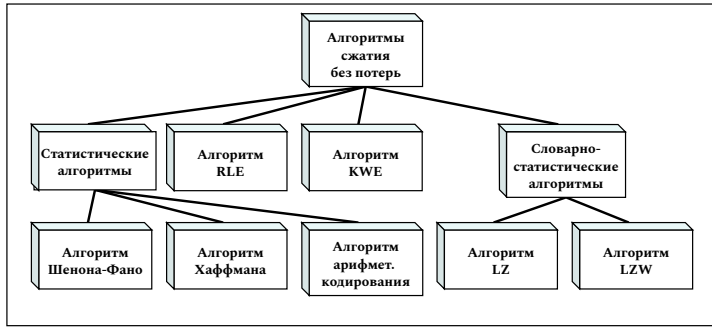


Рис. 2: Алгоритмы сжатия без потерь

Алгоритм оптимального кеширования

Данный алгоритм основан на построение дерева поиска правил относительно их частот [5], при этом, в данное дерево попадают только правила с определёнными частотами использования (далее популярность). Правила, не попавшие в основное дерево, хранятся на центральном процессоре коммутатора и доступ к ним осуществляется по запросу СП.

Для описания данного алгоритма потребуется ввести дополнительные обозначения. Введём понятие распределения заголовков пакетов P, где p_x обозначает вероятность получения пакета $x \to g = \{g^1, g^2, \ldots, g^k\}$. Также введём понятие коэффициента правильности $T_P(R_1, R_2)$, где R_1 и R_2 две различные таблицы потоков. Таким образом коэффициент правильности обозначает вероятность того, что заголовок пакета, согласно распределению P, будет идентифицироваться правилами $r_1 \in R_1$ и $r_2 \in R_2$ и их наборы действий совпадают $A_1 = A_2, A_1 \in r_1, A_2 \in r_2$.

$$T_P(R_1, R_2) = \sum_{x \to g, R_1(x) = R_2(x)} p_x$$

Введём оптимальное значение коэффициента правильности для заданной таблицы потоков R, числа правил n и распределения заголовков P.

$$\zeta(n, R, P) = \max_{R_i, |R_i| \leqslant n} T_P(R, R_i)$$

Таким образом, алгоритму необходимо найти и построить таблицу потоков R_a, основанную на данной таблице потоков R с наименьшим количеством правил n_0 и максимальным оптимальным коэффициентом правильности $\zeta(n, R, P)$. Пусть p^i вероятность выбора правила $r_i \in R$, в соответствии с распределением заголовков P. Пусть правила в таблице потоков R расположены в порядке невозрастания их популярности. Тогда:

$$\zeta(n, R, P) \geqslant \sum_{i \in [1,n]} p^i + 1 - \sum_{i \in [1,n_0]} p^i \geqslant n/n_0$$

Алгоритм рекурсивного сокращения

Данный алгоритм основан на применении дерева HyperSplit [7], а сжатие производится за счёт удаления дублирующихся правил [8]. Данный алгоритм поддерживает добавление и удаление правил в таблице потоков.

Под частично дублирующимся правилом в вершине понимается правило дублирующееся правилом, находящемся в листовой вершине. Полностью дублирующимся правилом, является правило для которого в каждой листовой вершине существует дублирующее его правило. Дублирующиеся правила перемещаются наверх дерева, что позволяет удалить полностью дублирующиеся правила.

Данный алгоритм рекурсивно использует процедуры NewHypersplit [9][10] для удаления повторяющихся правил из дерева, во время его построения. Затем удалённые повторяющиеся правила собираются в виде второй таблицы правил, называемой рекурсивной таблицей, для построения второго дерева. Возможно, что во втором дереве все ещё существуют повторяющиеся правила, и некоторые из них также удаляются и используются для построения третьего дерева. Этот процесс построения дерева выполняется рекурсивно до тех пор, пока в последнем дереве не будет дублированных правил.

Алгоритм с использованием битовых векторов для представления таблиц потоков

Данный алгоритм основан на применении битовых строк для представления таблиц потоков [11]. А именно таблица потоков разбивается на несколько частей, в каждой из которых для всех битов префиксов записывается два значения: подходит ли данный префикс, если в искомой строке 1, подходит ли данный префикс, если в искомой строке 0. Таким образом поиск по таблицам потоков

будет состоять из последовательного применения операции and. Все совпадающие поля делятся на $\frac{L}{s}$ sub-полей, где $(1 \leqslant s \leqslant L)$ обозначает длину подполя в битах. $K_j(j = 0, 1, ..., Ls - 1)$ обозначает бит в подполе j. Данный алгоритм уменьшает количество подходящих поисков, а также уменьшает задержку поиска.

3.2 Сравнение алгоритмов сжатия

Сравнение представленных алгоритмов сжатия представлено в таблице 1.

Название алгоритма сжатия	**Сложность построения**	**Степень сжатия**	**Внешняя память**	**Необх. декомпрессии**
Алгоритм оптимального кеширования	$O(N^2)$	$0.1 \ldots 0.9$	да	нет
Алгоритм рекурсивного сокращения	$O(N * log(N))$	0.1	нет	нет
Алгоритм с исп. битовых строк	$O(\frac{W}{K} * L)$	0.5	нет	нет
Распр. алгоритмы	$O(K * \log_2 N)$	$0.1 \ldots 0.8$	нет	да

Таблица 1: Сравнение алгоритмов сжатия.

У каждого рассмотренного алгоритма сжатия есть свои достоинства и недостатки, рассмотрим их:

1. **Распространённые алгоритмы сжатия** – в среднем имеют хорошие коэффициенты сжатия, но при этом требуется декомпрессия данных.

2. **Алгоритм оптимального кеширования** – имеет наибольший коэффициент сжатия, и быстро реализуем в рассматриваемой архитектуре сетевого процессорного устройства. Необходимость использования внешней памяти накладывает дополнительные расходы на обработку некоторых пакетов.

3. **Алгоритм рекурсивного сокращения** – имеет наименьший коэффициент сжатия, реализуем сложнее, чем

алгоритм оптимального кеширования. При этом данный алгоритм не требует использования внешней памяти.

4. **Алгоритм с использованием битовых строк** – имеет средний коэффициент сжатия, но при этом трудно реализуем в рассматриваемой архитектуре сетевого процессорного устройства. Так как для реализации данного алгоритма, необходимо добавить механизм циклов в язык ассемблера СПУ.

Таким образом, на основе обзора для дальнейшей реализации были выбраны два алгоритма сжатия данных, а именно алгоритм оптимального кеширования и алгоритм с использованием битовых строк.

4. Предложенные методы сжатия

Алгоритмы сжатия встраиваются в систему трансляции таблиц потоков и применяются при первоначальной трансляции таблицы потоков и при обновлении, полученном от контроллера. Весь алгоритм сжатия состоит из двух этапов: предварительной оптимизации и основного алгоритма сжатия.

4.1 Алгоритм предварительной оптимизации таблицы потоков

Данный алгоритм служит для предварительной оптимизации таблиц потоков перед их сжатием выбранными алгоритмами сжатия. Основная идея данного алгоритма заключается в удалении незначащих и дублирующихся правил из таблицы потоков. На вход алгоритму предварительной оптимизации подаётся корень дерева построенного по таблице потоков. Основным принципом является обход в глубину, в процессе которого выбираются похожие правила в дереве и попарно объединяются, затем все листовые вершины, в которых не осталось правил, удаляются.

4.2 Основные алгоритмы сжатия

Алгоритм оптимального кеширования

Основной частью алгоритма оптимального кеширования [5] является операция разделения исходной таблицы потоков на две. Для применения алгоритма необходимо построить дерево с указанием суммы частот листовых вершин. Получить набор вершин дерева V, отсортировать этот набор в невозрастающем порядке сумм частот листовых вершин. Создать набор вершин S, из которого впоследствии будет строится дерево. Создать счётчик, хранящий сумму вероятностей вершин в наборе вершин S. Получить первую вершину с максимальной суммой вероятностей, увеличить счётчик на данную величину, добавить эту вершину в набор вершин S и удалить из набора вершин V. Повторять последние три операции, пока счётчик меньше 0.95. Построить дерево из набора вершин S.

После выполнения этих операций мы получаем два набора вершин S, V. Набор вершин V соответствует второстепенному дереву, а набор вершин S соответствует первостепенному дереву. Первостепенное дерево преобразуется в программу на языке ассемблера, которая загружается в СПУ. А второстепенное дерево хранится на ЦПУ, доступ к нему осуществляется через операцию `cpucall`, которая получает на вход значение регистра, т.е. текущее значение признака, по которому происходит поиск по дереву V. Результатом является запись необходимых действий в специальную область памяти.

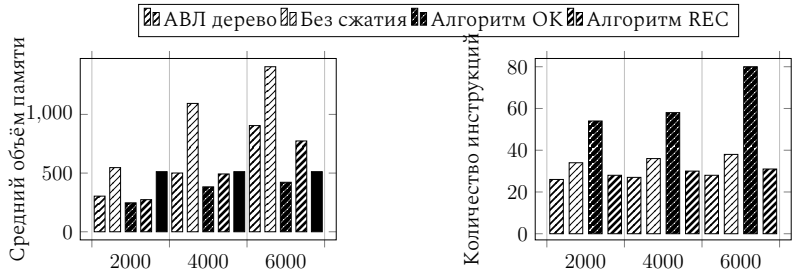
Алгоритм рекурсивного сокращения

Алгоритм использует деревья для рекурсивного удаления дублирующихся правил. Изначально строится дерево, в каждой вершине которого хранится правило.

Поиском в глубину выявляются частично дублирующиеся правила и полностью дублирующиеся правила. Согласно алгоритму они перемещаются наверх дерева, а затем удаляются. Таким образом пакет будет обрабатываться корректным правилом, так как удалённый правила были дубликатами правил в листовых вершинах.

5. Экспериментальное исследование

При проведении экспериментального исследования ставилась цель получить оценку степени сжатия программы на языке ассемблера сетевого процессорного устройства, на различных данных и оценку времени обновления таблиц потоков при использовании алгоритмов сжатия, на различных данных.



(a) Зависимость объёма используемой памяти СПУ от количества правил

(b) Зависимость среднего количества инструкций на обработку одного пакета от количества правил

Рис. 3: Результаты экспериментального исследования

5.1 Методика экспериментального исследования

Для оценки параметров необходимо исследовать программу на языке ассемблера, получаемую при использовании системы трансляции с алгоритмами сжатия и без. Для каждой программы, с помощью эмулятора сетевого процессорного устройства, будут исследоваться такие параметры как: объём памяти, занимаемой программой при обработке пакетов на эмуляторе сетевого процессорного устройства, и среднее время обработки пакета в тактах сетевого процессорного устройства.

Для проведения экспериментального исследования, необходимо последовательно выполнять следующие действия для каждого набора входных данных:

1. Выбрать таблицу потоков для данного эксперимента.
2. Провести трансляцию выбранной таблицы потоков в программу на языке ассемблера:
   – без использования алгоритмов сжатия, обычное дерево;

— без использования алгоритмов сжатия, с АВЛ деревом [12];

— с использованием разработанных алгоритмов сжатия.

3. Провести эмуляцию работы сетевого процессорного устройства с полученными программами на языке ассемблера.

4. Провести оценку результатов полученных в данном эксперименте.

5.2 Результаты экспериментального исследования

Было проведено экспериментальное исследование реализованных алгоритмов сжатия на имитационной модели СПУ. В ходе экспериментального исследования было выявлено, что использование реализованных алгоритмов сжатия позволяет сократить использование памяти СПУ.

В экспериментальном исследовании были использованы таблицы потоков различных размеров. Максимальный размер таблицы потоков составлял 6000 правил. В экспериментальном исследованием выбраны следующие: шаблоны таблиц потоков исходя из того, что ими покрывается большая вариантов использования OpenFlow коммутатора.

- Первый шаблон — шаблон правила таблицы потока содержит значения трех атрибутов: номер входного порта, MAC-адрес назначения и MAC-адрес источника.

- Второй шаблон — схема правила таблицы потока содержит значения двух атрибутов: Адрес назначения IPv4 и адрес источника IPv4.

- Третий шаблон — таблица правил содержит пять атрибутов: номер входного порта, MAC-адрес назначения, ID VLAN, ID заголовка L3-уровня (EtherType) и IPv4-адрес назначения.

Алгоритм оптимального кеширования показал наилучший коэффициент сжатия (рис. 3a), но при этом на обработку данного пакета в среднем затрачивалось наибольшее количество инструкций (рис. 3b). Это вызвано необходимостью обращения СПУ к ЦПУ, для получения правил, которые не хранятся в памяти СПУ.

Алгоритм рекурсивного сокращения показал коэфициент сжатия хуже, чем алгоритм оптимального кеширования (рис. 3a),

но меньшее среднее количество инструкций на обработку одного пакета (рис. 3b).

Заключение

Результатом данной работы стала разработка алгоритма сжатия таблиц потоков OpenFlow, использование которого позволило снизить объём используемой памяти СПУ, для хранения программ на языке ассемблера. Экспериментальное исследование предложенного алгоритма оптимального кеширования показало, что объём занимаемой памяти снижается в 2-2.5 раза, а для алгоритма рекурсивного сокращения в 1.5 - 2 раза. Что позволило увеличить возможное количество правил в таблицах потоках. Также в результате работы были внесены изменения в существующий эмулятор СПУ, что позволило использовать его для проведения экспериментального исследования предложенных алгоритмов сжатия.

В качестве возможных направлений дальнейших исследований можно указать изучение возможности применения более продвинутых алгоритмов сжатия с использованием специализированной TCAM памяти.

Литература

1. Смелянский Р.Л. *Программно-конфигурируемые сети*: Открытые системы, 2012. 9, С. 15–26.

2. Беззубцев С. О., Васин В. В., Волканов Д. Ю., Жайлауова Ш.Р., Мирошник В. А., Скобцова Ю. А., Смелянский Р. Л. *Об одном подходе к построению сетевого процессорного устройства*: Моделирование и анализ информационных систем, 2019. 26, 1, С. 1015–1818.

3. Open Networking Foundation. *OpenFlow Switch Specification Version 1.3.0 (Wire Protocol 0x04)* June 2012.

4. Rétvári, Gábor and Tapolcai, János and Kőrösi, Attila and Majdán, András and Heszberger, Zalán. *Compressing IP forwarding tables: Towards entropy bounds and beyond*: ACM SIGCOMM Computer Communication Review, 2013. 43, 2, p. 111–122.

5. Rottenstreich, Ori and Tapolcai, János. *Optimal rule caching and lossy compression for longest prefix matching*: IEEE/ACM Transactions on Networking, 2016. 25, 2, p. 864–878.

6. Kodituwakku, SR and Amarasinghe, US *Comparison of lossless data compression algorithms for text data*: Indian journal of computer science and engineering, 2010. 1, 4, p. 416–425.

7. Qi, Yaxuan and Xu, Lianghong and Yang, Baohua and Xue, Yibo and Li, Jun. *Packet Classification Algorithms: From Theory to Practice*: Proceedings - IEEE INFOCOM, 2009. p. 648–656.

8. Chang, Yeim-Kuan and Chen, Han-Chen. *Fast packet classification using recursive endpoint-cutting and bucket compression on FPGA*: The Computer Journal, 2019. 62, 2, p. 198–214.

9. Gupta, Pankaj and McKeown, Nick. *Packet classification using hierarchical intelligent cuttings*: Hot Interconnects VII, 1999. 40.

10. Singh, Sumeet and Baboescu, Florin and Varghese, George and Wang, Jia. *Packet classification using multidimensional cutting*: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, 2003. p. 213–224.

11. Shi, Zilin and Yang, Hui and Li, Junnan and Li, Chenglong and Li, Tao and Wang, Baosheng. *MsBV: A Memory Compression Scheme for Bit-Vector-Based Classification Lookup Tables*: IEEE Access, 2020. 8, p. 38673–38681.

12. Никифоров Н.И., Волканов Д.Ю., Скобцова Ю.А. *Анализ и исследование структур данных для поиска в таблицах классификации в сетевом процессорном устройстве с архитектурой RuNPU*: Программные системы и инструменты. — Т. 20 — Москва, МАКС Пресс, 2020. — с. 118–132.

Пантюхин Л.К., Антоненко В.А.

Разработка метода борьбы с атаками типа отказ в обслуживании при L4 балансировке

Введение

Для установлении TCP-соединения (Transmission Control Protocol) между клиентом и сервером используется трехкратное рукопожатие [2]:

1. Клиент отправляет пакет с TCP-флагом SYN (Synchronize) (далее *SYN пакет*).
2. Сервер, получив SYN пакет, отвечает пакетом с TCP-флагами SYN и ACK (Acknowledgment) (далее *SYN-ACK пакет*).
3. Клиент, получив SYN-ACK пакет, отвечает на него пакетом с TCP-флагом ACK (далее *ACK пакет*) и считает TCP-соединение открытым.
4. Сервер, получив ACK пакет, считает TCP-соединение открытым.

На втором шаге сервер создает *TCB* (Transmission Control Block) – структуру данных, в которой хранится информация о TCP-соединении [2]. Также у сервера существует *бэклог* (backlog queue) – очередь из полуоткрытых TCP-соединений, имеющая максимальный размер, отражающий максимальное количество TCP-соединений, для которых сервер может одновременно ожидать подтверждения. Таким образом, если злоумышленник будет отправлять много SYN пакетов, но не будет отправлять ACK пакеты, то он сможет провести атаку на сервер, истощив его ресурсы, и другие клиенты не смогут установить TCP-соединение. Такой вид атаки типа отказ в обслуживании (DoS, Denial-of-Service) называется *SYN-flood* [1].

Огромное количество сервисов, развернутых в облаке, нуждаются в балансировке нагрузки [4]. Балансировщик нагрузки в рассматриваемой задаче распределяет трафик между виртуальными машинами, на которых запущены локальные экземпляры приложения. Такие виртуальные машины далее будем называть *бэкендами* (backend).

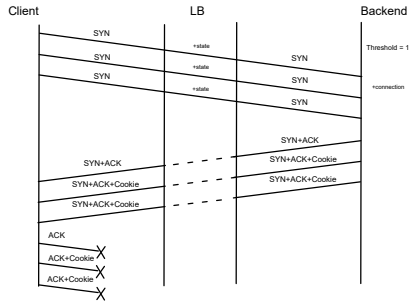


Рис. 1: Атака SYN-flood на балансировщик нагрузки с сохранением состояния

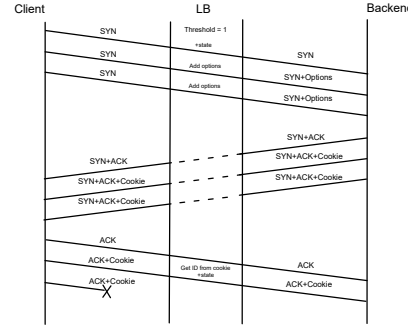


Рис. 2: Концепция применения SYN-cookies в балансировщике нагрузки

Балансировщик нагрузки можно классифицировать по уровню модели ISO/OSI, на котором он работает. В текущей задаче рассматривается балансировщик, который работает на уровне TCP, что соответствует четвертому уровню (L4).

Балансировщики нагрузки распределяют нагрузку при помощи различных стратегий и алгоритмов. В зависимости от стратегий балансировщики можно разделить на 2 класса:

1. **Без сохранения состояния (stateless)**
   Количество хранимой информации на балансировщиках не зависит от количества активных TCP-соединений.

2. **С сохранением состояния (stateful)**
   Балансировщики сохраняют состояния TCP-соединений и количество хранимой информации зависит от количества активных TCP-соединений.

В текущей работе рассматривается балансировщик нагрузки с сохранением состояния. Далее под «балансировщиком» подразумевается L4 балансировщик с сохранением состояния.

Рассмотрим проблему, которая возникает при SYN-flood атаке на L4 балансировщик нагрузки с сохранением состояния. Рассмотрим балансировщик (LB) с одним бэкендом (Backend), изображенные на рисунке [1]:

1. Пусть после достижения количеством полуоткрытых TCP-соединений 1 бэкенд включает механизм защиты SYN-cookies (о нем подробнее написано далее), позволяющий

бэкенду не сохранять информацию о полуоткрытых TCP-соединениях.

2. Пусть клиент (Client) посылает 3 SYN пакета.

3. На каждый входящий пакет балансировщик сохранит по состоянию (state) и перенаправит их на единственный бэкенд.

4. Бэкенд, ответит на первый SYN пакет SYN-ACK пакетом и сохранит информацию о полуоткрытом TCP-соединении (connection). На второй и третий SYN пакет бэкенд включит механизм SYN-cookies и не сохранит информацию.

5. Клиент не отвечает на SYN-ACK пакеты.

6. В итоге у балансировщика хранится информация о 3 TCP-соединениях, а у бэкенда только об одном.

Данный простой пример демонстрирует, как при SYN-flood атаке бэкенды защищаются от нее в то время, как балансировщик вынужден сохранять состояния каждого TCP-соединения. Цель данной работы разработать метод защиты для балансировщика.

Далее во второй главе представлен обзор схожих работ. В третьей главе на предложен метод защиты от SYN-flood атаки для балансировщика. В четвертой главе описана программная реализация метода. В пятой главе проведено экспериментальное исследование с целью демонстрации работоспособности предложенного метода.

1. Схожие работы

Подробное описание SYN-flood атаки и основных методов защиты от нее для конечных устройств приведено в [1]. Также более современное описание SYN-flood атаки представлено в [6]. Среди методов присутствует метод SYN-cookies, изначально описанный в [5]. Идея метода SYN-cookies – кодирование в TCP-поле Sequence Number SYN-ACK пакета счетчика времени и значения MSS (Max Segment Size) при помощи хеширования вместе с IP-адресами и TCP-портами пакета. Полученное значение еще называется *SYN-cookie*. При получении ACK пакета хост декодирует счетчик времени и значение MSS, проверяет валидность значения в ACK пакете и только тогда открывает TCP-соединение. Таким образом, хосту нет необходимости создавать TCB и хранить информацию о полуоткрытом TCP-соединении при получении SYN

пакета. Данный метод лежит в основе предложенного метода борьбы для балансировщика нагрузки.

В [7] представлена устойчивая к SYN-flood атаке архитектура балансировщика Prism, в которой таблица TCP-соединений разделена на несколько таблиц. В этой архитектуре происходит анализ трафика со стороны бэкендов. При этом бэкенды могут использовать механизм SYN-cookies.

В [8] приведено описание балансировщика нагрузки Maglev компании Google. В данной работе упоминается о том, что таблица TCP-соединений балансировщика нагрузки конечна и может быть переполнена SYN-flood атакой или большой нагрузкой.

В [9] проводится анализ средств защиты AWS от DDoS-атак, в частности, от SYN-flood атаки. Из данной работы можно сделать вывод, что балансировщик нагрузки компании Amazon защищается с помощью фильтрации и быстрого масштабирования.

Балансировщик Ananta [10] от компании Microsoft при обнаружении SYN-flood атаки изолирует весь трафик к атакованному сервису.

В [11] представлен балансировщик без сохранения состояния Beamer, устойчивый к SYN-flood атаке.

В [17] также представлен балансировщик без сохранения состояния CHEETAH, устойчивый к SYN-flood атаке. Так же, как и Prism использует трафик от бэкендов к клиентам. Представленный в данной статье метод схож с CHEETAH тем, что использует cookie для сохранения информации о полуоткрытом TCP-соединении.

2. Описание метода борьбы с SYN-flood атакой для балансировщика нагрузки

На основе SYN-cookies предложен следующий метод:

1. При достижении количеством состояний некоторого порога или при обнаружении SYN-flood атаки балансировщик перестает сохранять состояния и вместо этого добавляет во входящий SYN пакет некоторую информацию.

2. На бэкенде, на который был переслан входящий SYN пакет, может быть 2 случая:

    - Бэкенд не обнаружил атаку и ничего не знает о ней, поэтому его механизм SYN-cookies выключен, но по запросу балансировщика он генерирует SYN-cookie.

- Бэкенд обнаружил атаку, поэтому его механизм SYN-cookies включен.

В обоих случаях на основе полученной от балансировщика информации генерирует значение TCP-поля Sequence Number исходящего SYN-ACK пакета.

3. Если балансировщик получает входящий ACK пакет, он считывает значение TCP-поля Acknowledgment Number (это должно быть значение Sequence Number SYN-ACK пакета, увеличенное на 1), проверяет его валидность и декодирует информацию, необходимую для создания состояния и отправки ACK пакета на тот же бэкенд, на который был отправлен SYN.

На рисунке 2 представлен пример применения концепции на основе метода SYN-cookies:

1. Пусть балансировщик (LB) имеет значение порога 1 и клиент (Client) отправил 3 SYN пакета. Балансировщик при получении этих входящих пакетов сохраняет состояние для первого TCP-соединения (state), а для остальных не сохраняет, так как достигнут порог (Threshold). Вместо этого балансировщик добавляет идентификаторы тех бэкендов, на которые распределил эти TCP-соединения, в TCP-опции SYN пакетов.

2. Бэкенд (Backend), получивший первый SYN пакет генерирует SYN-ACK пакет. Бэкенды, получившие второй и третий SYN-пакеты генерируют SYN-ACK пакеты, в которых в TCP-поле Sequence Number (Cookie) закодированы их идентификаторы, переданные через TCP-опции.

3. Клиент отвечает только на первые два SYN-ACK пакета.

4. На балансировщике при получении ACK пакетов:

   - Когда на балансировщик приходит ACK пакет первого TCP-соединения, состояние которого сохранено на балансировщике, поэтому балансировщик просто направляет его на необходимый бэкенд.
   - Когда на балансировщик приходит ACK пакет второго TCP-соединения, у него нет сохраненного состояния, поэтому он не знает, куда его направить. Балансировщик читает значения поля TCP-поля Acknowledgment Number

(Cookie), проверяет его на валидность, получает из него идентификатор бэкенда, на который ранее направлял SYN пакет текущего TCP-соединения и сохраняет состояние. Далее балансировщик направляет ACK пакет на нужный бэкенд.

- Третий ACK пакет не был получен балансировщиком, однако в балансировщике не хранится состояние его TCP-соединения.

Первое TCP-соединение в примере демонстрирует легитимное соединение при отсутствии атаки. Второе TCP-соединение демонстрирует легитимное TCP-соединение при совершении SYN-flood атаки на балансировщик. Последнее TCP-соединение демонстрирует трафик злоумышленника при совершении SYN-flood атаки на балансировщик.

3. Реализация метода борьбы с SYN-flood атакой для балансировщика нагрузки

Для демонстрации работы предложенного метода защиты балансировщика необходим экспериментальный стенд. С помощью Mininet [18] реализованы клиент и хост с балансировщиком нагрузки. Virtualbox [19] будет использоваться для развертывания виртуальной машины с Mininet и виртуальной машины с бэкендом, так как для него необходимо производить модификацию ядра Linux. Virtualbox позволяет обеспечить соединение виртуальных машин друг с другом, поэтому существует возможность подключить бэкенд к сети в Mininet.

В качестве балансирощика нагрузки взят модельный балансировщик, реализованный на языке Python с использованием модуля Scapy [23]. В модели реализованы таймауты для каждого TCP-соединения, таким образом, давно не используемые состояния удаляются. Максимальное число состояний не установлено. Также в качестве балансировщиков рассматривались модуль Linux IPVS (IP Virtual Server), реализованный на его основе балансировщик нагрузки LVS (Linux Virtual Server) [20], и модифицированная версия LVS с DPDK (Data Plane Development Kit) [22] – DVPS (DPDK-LVS) [21].

Схема экспериментального стенда представлена на рисунке 3. Рассмотрим отдельно каждую виртуальную машину.

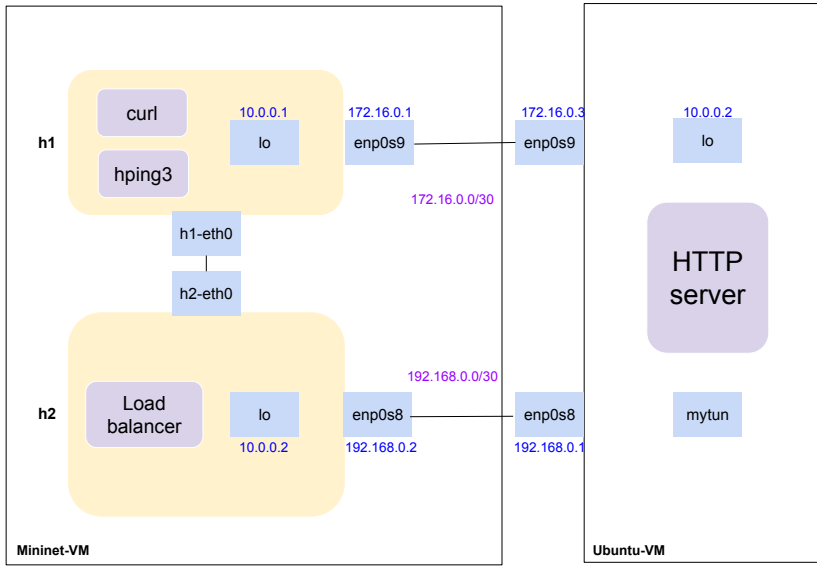


Рис. 3: Экспериментальный стенд

Mininet-VM

На данной виртуальной машине запускается сеть Mininet с 2 хостами:

1. **h1**
   Реализует клиента-злоумышленника, на котором запускается утилита **hping3** с параметрами для SYN-flood атаки балансировщика на h2. Также со задержкой, равной случайной величине с распределением Пуассона с параметром 1, вызывается фоновый процесс с утилитой **curl**, имитирующей легитимные запросы на установку TCP-соединения.

2. **h2**
   Реализует балансировщик нагрузки, который распределяет входящие TCP-соединения между бэкендами (в данном случае один бэкенд). Между балансировщиком и бэкендом трафик пересылается через IPIP-туннель. В качестве TCP-опции для передачи информации используется экспериментальная TCP-опция 253 [3]. При атаке балансировщик передает бэкенду идентификатор бэкенда и секретный ключ для хеширования.

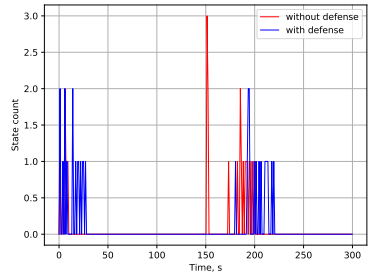


Рис. 4: Количество состояний при отсутствии атаки

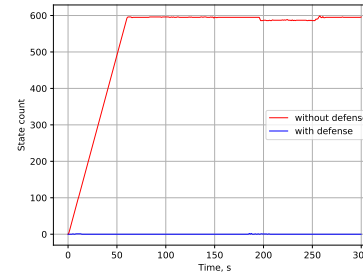


Рис. 5: Количество состояний при атаке 10 пакетов в секунду

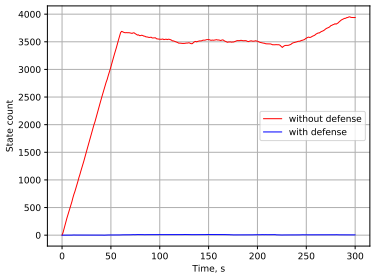


Рис. 6: Количество состояний при атаке 100 пакетов в секунду

Ubuntu-VM

Данная виртуальная машина реализует бэкенд с модифицированным ядром Linux версии 5.11.14. На нем запущен простой HTTP-сервер. SYN-ACK пакеты отправляются в обход балансировщика сразу на h1.

В ядре Linux версии 5.11.14 были осуществлены следующие модификации:

- Добавлена обработка TCP-опции 253 входящих SYN пакетов.
- При наличии TCP-опции 253 в SYN пакете ядро обязательно формирует SYN-cookie.
- Переделан механизм генерации SYN-cookie.

Интенсивность атаки	Механизм защиты	Среднее значение	Ср. кв. отклонение
Без атаки	Без защиты	0.0833	0.341
Без атаки	С защитой	0.103	0.355
10 пакетов в секунду	Без защиты	531.876	144.537
10 пакетов в секунду	С защитой	0.06	0.264
100 пакетов в секунду	Без защиты	3205.986	880.808
100 пакетов в секунду	С защитой	6.207	3.052

Таблица 1: Среднее значение и среднеквадратичное отклонение каждой кривой

4. Экспериментальное исследование

Проведем экспериментальное исследование с целью демонстрации работоспособности предложенного метода.

Каждый эксперимент длится 300 секунд (5 минут). В балансировщике удаляются не используемые более 60 секунд состояния. Такое значение выбрано, так как 60 секунд – значение по умолчанию таймаута состояния после получения SYN пакета до получения ACK пакета в IPVS. SYN-flood генерируется с подмененными IP-адресами источника. У каждого эксперимента задана одна из 3 интенсивностей атаки:

- без использования SYN-flood;
- 10 пакетов в секунду;
- 100 пакетов в секунду;

Также в каждом эксперименте балансировщик в одном из двух состояний:

- механизм защиты включен;
- механизм защиты выключен.

Таким образом, всего проведено 6 экспериментов. В каждом получена зависимость количества состояний от времени.

На рисунках 4, 5 и 6 представлены результаты проведенных экспериментов. Также в таблице 1 приведены среднее значение и среднеквадратичное отклонение каждой кривой. Стоит отметить, что в случаях, когда среднее значение меньше 1, среднеквадратичное отклонение не является информативным, так как точность значений превышает точность измерений.

При отсутствии атаки (рисунок 4) curl устанавливает TCP-соединение на короткое время и завершает его. При завершении TCP-соединения состояние в балансировщике удаляется, поэтому интенсивности запусков curl не хватает, чтобы в балансировщике сохранилось больше 3 состояний одновременно независимсмо от наличия включенного механизма защиты.

При отключенном механизме защиты и проведении атаки (рисунки 5 и 6) в начале экспериментов наблюдается сильное увеличение количества состояний, а затем стабилизация. Это обусловлено установкой таймаута состояний размером равным 60 секундам. Таким образом, после 1 минуты состояния давно не используемых TCP-соединений удаляются. При включенном же механизме защиты количество состояний растет исключительно за счет открытия TCP-соединений утилитой curl. В случае размера атаки в 100 пакетов в секунду количество состояний, меньшее 6000, обсуловлено достижением предела пропускной способности экспериментального стенда.

Как видно из графиков, данный механизм защиты действительно способен значительно уменьшить количество сохраненных состояний в балансировщике при проведеннии на него SYN-flood атаки.

Заключение

На основе полученных результатов можно сделать вывод, что предложенный метод работоспособен и позволяет не сохранять состояния полутокрытых TCP-соединений на балансировщике нагрузки. Однако перед прикладным использованием необходимо провести исследования на реальном оборудовании с промышленным L4 балансировщиком нагрузки с сохранением состояния.

Литература

1. Eddy, Wesley. TCP SYN Flooding Attacks and Common Mitigations URL: `https://rfc-editor.org/rfc/rfc4987.txt` RFC Editor,August 2007.

2. Transmission Control Protocol URL: `https://rfc-editor.org/rfc/rfc793.txt` RFC Editor,September 1981.

3. Touch, Dr. Joseph D. Shared Use of Experimental TCP Options URL: `https://rfc-editor.org/rfc/rfc6994.txt` RFC Editor,August 2013.

4. Rahman, Mazedur and Iqbal, Samira and Gao, Jerry. Load balancer as a service in cloud computing 2014 IEEE 8th International Symposium on Service Oriented System Engineering 2014.

5. Bernstein, D. J. SYN cookies [Электронный ресурс]. — Электрон. дан. — URL: `http://cr.yp.to/syncookies.html`

6. Scholz, Dominik and Gallenmüller, Sebastian and Stubbe, Henning and Jaber, Bassam and Rouhi, Minoo and Carle, Georg. Me love (SYN-) cookies: SYN flood mitigation in programmable data planes 2020.

7. Cohen, Reuven and Kadosh, Matty and Lo, Alan and Sayah, Qasem. Hardware SYN Attack Protection For High Performance Load Balancers 2021 IEEE Symposium on High-Performance Interconnects (HOTI) 2021.

8. Eisenbud, Daniel E and Yi, Cheng and Contavalli, Carlo and Smith, Cody and Kononov, Roman and Mann-Hielscher, Eric and

Cilingiroglu, Ardas and Cheyney, Bin and Shang, Wentao and Hosein, Jinnah Dylan. Maglev: A fast and reliable software network load balancer 13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16) 2016.

9. Khan, Waseem. DDOS Mitigation Analysis of AWS Cloud Network 2017.

10. Patel, Parveen and Bansal, Deepak and Yuan, Lihua and Murthy, Ashwin and Greenberg, Albert and Maltz, David A and Kern, Randy and Kumar, Hemant and Zikos, Marios and Wu, Hongyu and others. Ananta: Cloud scale load balancing ACM New York, NY, USA,2013.

11. Olteanu, Vladimir and Agache, Alexandru and Voinescu, Andrei and Raiciu, Costin. Stateless datacenter load-balancing with beamer 15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18) 2018.

12. Aumasson, Jean-Philippe and Bernstein, Daniel J. SipHash: a fast short-input PRF International Conference on Cryptology in India 2012.

13. Eddy, Wesley M. Defenses against TCP SYN flooding attacks Cisco,2006.

14. Bogdanoski, Mitko and Suminoski, Tomislav and Risteski, Aleksandar. Analysis of the SYN flood DoS attack MECS Publisher,2013.

15. Lemon, Jonathan. Resisting SYN Flood DoS Attacks with a SYN Cache Proceedings of the BSD Conference 2002 on BSD Conference USENIX Association,2002.

16. Salunkhe, H and Jadhav, Sanjay and Bhosale, Vijay. Analysis and review of TCP SYN flood attack on network with its detection and performance metrics 2017.

17. Barbette, Tom and Tang, Chen and Yao, Haoran and Kostić, Dejan and Jr., Gerald Q. Maguire and Papadimitratos, Panagiotis and Chiesa, Marco. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20) URL: `https://www.usenix.org/conference/nsdi20/presentation/barbette` USENIX Association,February 2020.

18. Mininet [Электронный ресурс]. — Электрон. дан. — URL: https://github.com/mininet/mininet/

19. VirtualBox [Электронный ресурс]. — Электрон. дан. — URL: https://www.virtualbox.org/

20. LVS [Электронный ресурс]. — Электрон. дан. — URL: https://github.com/alibaba/LVS

21. DPVS [Электронный ресурс]. — Электрон. дан. — URL: https://github.com/iqiyi/dpvs

22. DPDK [Электронный ресурс]. — Электрон. дан. — URL: https://www.dpdk.org/

23. Scapy documentation [Электронный ресурс]. — Электрон. дан. — URL: https://scapy.readthedocs.io/en/latest/

Рябченков В. М., Глонина А. Б.

Разработка аналитического метода оценки времени отклика задач в системах интегрированной модульной авионики

Введение

В данной работе рассматриваются многопроцессорные системы на базе архитектуры интегрированной модульной авионики (ИМА), которая в настоящее время является основной концепцией проектирования и разработки вычислительных систем воздушных судов. Особенности организации таких систем приведены в работе [1]. Системы этого класса представляют собой распределённые системы реального времени, состоящие из вычислительных модулей, соединеных сетью [2]. Типичная система ИМА использует несколько типов унифицированных модулей и коммутируемую сеть, поддерживающую виртуальные каналы передачи данных.

Каждый модуль содержит один или несколько многоядерных процессоров. Вычислительные задачи для систем ИМА сгруппированы в разделы. Задачи одного раздела выполняются строго на одном ядре процессора. Все задачи являются периодическими; наименьшее общее кратное периодов всех задач называется интервалом планирования. Выполнение задачи в течение периода будем называть экземпляром задачи или работой. Для каждого раздела на интервале планирования выделен фиксированный набор окон выполнения — набор интервалов времени, в течение которых могут выполняться задачи из данного раздела. Внутри окон раздела за управление работами отвечает локальный планировщик с фиксированными приоритетами и вытеснением.

Каждая вычислительная задача обладает следующими характеристиками: период выполнения, интервал длительности выполнения, фиксированный и уникальный в рамках раздела приоритет. Модули соединены некоторой средой передачи данных, обычно основанной на коммутации виртуальных каналов. Между задачами как из разных модулей, так и из одного модуля, могут существовать зависимости по данным, т. е. задача-получатель не может начать выполнение своей работы, пока не получит

сообщения от соответствующих работ всех задач-отправителей.

Под конфигурацией системы ИМА понимается: количество вычислителей; набор разделов, в т.ч. характеристики входящих в них прикладных задач и зависимости по данным между ними; привязка разделов к вычислителям; расписание окон разделов.

Время отклика задачи — это величина равная максимальному по всем работам этой задачи значению времен их завершения относительно начала периода. Наихудшее время отклика задачи (WCRT) — это максимально возможное время отклика задачи на данной конфигурации системы.

WCRT используется при проектировании систем ИМА для проверки выполнения ограничений реального времени, например, при решении задачи, описанной в работе [3]. Как правило при проектировании систем ИМА считают, что время выполнения каждой работы фиксировано и равно WCET (Worst Case Execution Time — правая граница интервала длительности выполнения работ задачи) соответствующей задачи. В таком предположении WCRT может быть получено с помощью прогона имитационной модели с заданными временами выполнения задач. Однако известно, что в случае наличия зависимостей по данным могут возникать аномалии, при которых уменьшение времени выполнения работы одной задачи может привести к увеличению времени отклика другой задачи. Время выполнения работы варьируется от запуска к запуску в зависимости от некоторых факторов, например таких как наличие кэш памяти или различных путей выполнения кода программы при разных входных данных. На практике время выполнения работы задачи обычно меньше чем WCET. Поэтому очень важно учитывать интервальную неопределенность длительности выполнения работ. Также важно отметить, что поиск комбинации времен выполнения работ, при которых WCRT достигается, для систем реальной размерности занимает неприемлемо большое время. Аналитические методы лишены данного недостатка, и позволяют достаточно быстро находить верхнюю оценку значения WCRT.

В данной работе предложен аналитический метод для решения проблемы оценки WCRT для конкретной задачи на заданной конфигурации системы ИМА, основанный на методе STBA [5]. Затем, приведено сравнение предложенного алгоритма с аналитическим методом WCDO из работы [6].

1. Математическая постановка задачи

Введем формальное определение конфигурации системы ИМА.

Пусть $HW = \{N_i\}_{i=1}^{N^p}$ — набор вычислительных ядер, где $N^p \in \mathbb{N}$ — количество вычислительных ядер в системе ИМА.

Введем обозначения для рабочей нагрузки, которая состоит из набора задач, сгруппированных в разделы, и информации о зависимостях по данным между задачами. Рабочая нагрузка будет представлена в виде кортежа $WL = \langle Part, Msg \rangle$, где $Part$ — набор разделов, Msg — набор сообщений, передаваемых между задачами.

Набор разделов $Part$ представляется в виде $Part = \{P_i\}_{i=1}^{M}$, где $M \in \mathbb{N}$ — количество разделов, $P_i = \{\tau_{i_j}\}_{j=1}^{K_i}$ — набор задач i-го раздела, $\tau_{i_j} = \langle pr_{i_j}, [C_{i_j}^b, C_{i_j}], T_{i_j} \rangle$ — атрибуты j-й задачи i-го раздела, $K_i \in \mathbb{N}$ — количество задач в i-ом разделе, $pr_{i_j} \in \mathbb{N}$ — уникальный в рамках раздела приоритет задачи, $[C_{i_j}^b, C_{i_j}]$, $C_{i_j}^b, C_{i_j} \in \mathbb{N}$ — интервал длительности выполнения работ задачи, $T_{i_j} \in \mathbb{N}$ — период задачи.

Обозначим за $\hat{T}$ множество всех задач в системе ИМА: $\hat{T} = \cup_{i=1}^{M} P_i$. Набор сообщений Msg представляется в виде $Msg = \{Msg_j\}_{j=1}^{H}$, где H — количество синхронных сообщений, передаваемых между задачами. Сообщение Msg_j представляется в виде кортежа $Msg_j = \langle S_j, R_j, dn_j^{min}, dn_j^{max} \rangle$, где $S_j \in \hat{T}$ — задача-отправитель, $R_j \in \hat{T}$ — задача-получатель, $dn_j^{min} \in \mathbb{Z}_+$ — наименьшая длительность передачи сообщения, $dn_j^{max} \in \mathbb{Z}_+$ — наибольшая длительность передачи сообщения.

Отображение $Bind : Part \mapsto HW$ задает привязку разделов к ядрам. Каждый раздел привязан к одному ядру. На интервале планирования длительностью L ($L \in \mathbb{N}$) задано расписание окон разделов в виде $Sched = \{\{\langle start_{ij}, stop_{ij} \rangle\}_{j=1}^{N_i^w}\}_{i=1}^{M}$, где $N_i^w \in \mathbb{N}$ — количество окон для i-го раздела, $start_{ij}, stop_{ij} \in \overline{0, L}$ — времена начала и завершения j-го окна i-го раздела.

L как правило равно наименьшему общему кратному всех T_{i_j}. В связи с этим далее будем считать, что L кратно T_{i_j} для любых $i \in \overline{1, M}$ $\forall j \in \overline{1, K_i}$.

Во введенных выше обозначениях конфигурация системы ИМА — это кортеж $\langle HW, WL, Bind, Sched \rangle$, где HW — набор вычислительных ядер, WL — описание рабочей нагрузки, $Bind$ — привязка разделов к ядрам, $Sched$ — расписание окон разделов.

Рабочей нагрузке соответствует ациклический направленный граф, далее обозначаемый как G. Вершины в нем соответствуют задачам, а дуги — сообщениям.

Пусть $CONF$ — множество всевозможных конфигураций системы ИМА, $conf = \langle HW, WL, Bind, Sched \rangle$, $\forall\ conf \in CONF$. При анализе некоторой конфигурации системы $conf \in CONF$ рассматривается ее функционирование в течение интервала планирования длительностью L.

Для каждой задачи $\tau_i \in \hat{T}$ на L определен набор работ: $W_i = w_{ij}$, где $j = 1, \ldots, \frac{L}{T_i}$ (T_i — период задачи τ_i). Пусть $\tau_i \in \hat{T}$ — исследуемая задача, R_{ij} — время отклика j-й работы i-й задачи ($j \geq 1$), Out_{ij} — время завершения работы j задачи i, тогда:

$$R_{ij} = Out_{ij} - (j-1) * T_i$$

Время отклика задачи τ_i вычисляется по формуле:

$$R_i = \max_j R_{ij}$$

Пусть $\alpha_i = (\alpha_{i1}, \ldots, \alpha_{ih_i})$ — набор конкретных длительностей выполнения работ задачи $\tau_i \in \hat{T}$, где $h_i = \frac{L}{T_i}$, $\alpha_{ij} \in [C_i^b, C_i]$. Тогда $\alpha = (\alpha_1, \ldots, \alpha_{|\hat{T}|})$ — набор конкретных длительностей выполнения всех работ всех задач для заданной конфигурации. Пусть $\mu_i = (\mu_{i1}, \ldots, \mu_{ih_i})$ — набор конкретных длительностей передач экземпляров сообщения Msg_i, где $S_i = \tau_i$, $\mu_{ij} \in [dn_i^{min}, dn_i^{max}]$. Тогда $\mu = (\mu_1, \ldots, \mu_{|H|})$ — набор конкретных длительностей передач для всех экземпляров всех сообщений в заданной конфигурации.

Пусть $C = C(conf)$ — множество всевозможных наборов α для конфигурации $conf$, а $Cm = Cm(conf)$ — множество всевозможных наборов μ для конфигурации $conf$.

Дано: конфигурация системы ИМА $conf \in CONF$, исследуемая задача $\tau_i \in \hat{T}$.

Требуется найти:

$$\max_{C,\, Cm} R_i$$

2. Адаптация алгоритма STBA под системы ИМА

В данном разделе представлено подробное описание аналитического метода STBA из работы [5], адаптированного под постановку задачи, описанную выше. В данный метод внесена модификация, позволяющая повысить точность получаемых оценок за счет изменения вида используемых в алгоритме

неравенств, а также предложена его адаптация под системы ИМА и интервальную неопределенность длительности передачи сообщений.

Метод STBA представляет собой вычисление границ интервалов для каждой работы из набора W_i каждой задачи $\tau_i \in \hat{T}$. Данные интервалы характеризуются следующими парами: $(w_{ij}^{minA}, w_{ij}^{maxA})$, $(w_{ij}^{minS}, w_{ij}^{maxS})$, $(w_{ij}^{minF}, w_{ij}^{maxF})$, которые обозначают минимальное и максимальное время готовности к выполнению, начала выполнения и завершения выполнения работы w_{ij} соответственно.

2.1 Модифицированный алгоритм STBA

Задачу, у которой нет предшественников, назовем задачей-истоком. В данной работе считается, что для каждой задачи-истока время готовности к выполнению первой работы равно нулю.

Пусть W — множество всех работ всех задач графа G, т. е. $W = \{w_{ij} \mid \exists\, \tau_i : \tau_i \in \hat{T}, w_{ij} \in W_i\}$. В соответствии с постановкой задачи, количество работ каждой задачи τ_i определяется формулой $\frac{L}{T_i}$, где L — интервал планирования, а T_i — период задачи τ_i. Пусть τ_i — задача-исток, тогда:

$$w_{ij}^{minA} = T_i * (j-1) \tag{1}$$

$$w_{ij}^{maxA} = T_i * (j-1) \tag{2}$$

Пусть $pred[\tau_i]$ — множество непосредственных предшественников задачи τ_i. Тогда, если $|pred[\tau_i]| \geq 1$, то:

$$w_{ij}^{minA} = \max_{\tau_p \in pred[\tau_i]} (w_{pj}^{minF}) \tag{3}$$

$$w_{ij}^{maxA} = \max_{\tau_p \in pred[\tau_i]} (w_{pj}^{maxF}) \tag{4}$$

Пусть $proc(\tau_i)$ — номер процессорного ядра, на котором выполняется задача τ_i. Далее введем следующие множества: $hp(w_{ij})$, w_{ij}^{pmtor} и w_{ij}^{excl}.

$hp(w_{ij}) = \{w_{sk} \mid proc(\tau_s) = proc(\tau_i),\ pr_s > pr_i\}$ — множество работ задач более высокого приоритета чем задача τ_i, выполняющихся с ней на одном процессорном ядре. w_{ij}^{pmtor} — множество работ, которые повлияли на w_{ij}^{maxS} или w_{ij}^{maxF} работы w_{ij}. Изначально w_{ij}^{pmtor} пустое множество. При вычислении максимального времени готовности к выполнению в соответствии с уравнением 4, множество w_{ij}^{pmtor} наследуется от каждого

непосредственного предшественника работы w_{ij}: $w_{ij}^{pmtor} = w_{pj}^{pmtor} \cup w_{ij}^{pmtor}, \forall \tau_p \in pred[\tau_i] : proc(\tau_p) = proc(\tau_i)$. w_{ij}^{excl} — множество работ более высокого приоритета, которые зависят по данным от работы w_{ij} в топологическом порядке.

Формулы для $w_{ij}^{minS}, w_{ij}^{maxS}$ работы w_{ij} имеют следующий вид:

$$w_{ij}^{minS} = \max(w_{ij}^{minA}, \alpha_{w_{ij}}(w_{ij}^{minS})) \tag{5}$$

$$w_{ij}^{maxS} = w_{ij}^{maxA} + Delay_{w_{ij}}^{h} \tag{6}$$

где:

$$\alpha_{w_{ij}}(w_{ij}^{minS}) = \max_{w_{sk} \in L_{w_{ij}}} (w_{sk}^{minF}) \tag{7}$$

$$L_{w_{ij}} = \left\{ w_{sk} \;\middle|\; \begin{array}{c} w_{sk} \in hp(w_{ij}), \\ w_{ij}^{minA} < w_{sk}^{minF}, \\ w_{sk}^{maxS} \leq w_{ij}^{minS} \end{array} \right\}$$

$L_{w_{ij}}$ — множество работ из $hp(w_{ij})$, которые продолжают своё выполнение в момент готовности к выполнению работы w_{ij}.

$$Delay_{w_{ij}}^{h} = \sum_{w_{sk} \in \Delta_{w_{ij}}} \min(C_s, w_{sk}^{maxF} - w_{ij}^{maxA}) \tag{8}$$

где

$$\Delta_{w_{ij}} = \left\{ w_{sk} \;\middle|\; \begin{array}{c} w_{sk} \in hp(w_{ij}), \\ w_{sk}^{minS} \leq w_{ij}^{maxS}, \\ w_{ij}^{minA} < w_{sk}^{maxF}, \\ w_{sk} \notin w_{ij}^{pmtor}, w_{sk} \notin w_{ij}^{excl} \end{array} \right\}$$

— множество работ из $hp(w_{ij})$, которые могут продолжать свое выполнение в момент готовности к выполнению работы w_{ij}.

Наименьшее и наибольшее время завершения выполнения работы w_{ij} определяется следующими формулами:

$$w_{ij}^{minF} = w_{ij}^{minS} + C_i^b + \gamma_{w_{ij}}(w_{ij}^{minS}) \tag{9}$$

$$w_{ij}^{maxF} = w_{ij}^{maxS} + C_i + \delta_{w_{ij}}(w_{ij}^{maxS}) \tag{10}$$

где

$$\gamma_{w_{ij}}(w_{ij}^{minS}) = \sum_{w_{sk} \in X_{w_{ij}}} C_s^b \tag{11}$$

$$X_{w_{ij}} = \left\{ w_{sk} \;\middle|\; \begin{array}{c} w_{sk} \in hp(w_{ij}), \\ w_{ij}^{minS} \leq w_{sk}^{minS} < w_{ij}^{minF}, \\ w_{ij}^{minS} \leq w_{sk}^{maxS} < w_{ij}^{minF} \end{array} \right\}$$

$X_{w_{ij}}$ — множество работ из $hp(w_{ij})$ которые всегда будут вытеснять работу w_{ij}.

$$\delta_{w_{ij}}(w_{ij}^{maxS}) = \sum_{w_{sk} \in Y_{w_{ij}}} C_s \tag{12}$$

где

$$Y_{w_{ij}} = \left\{ w_{sk} \left| \begin{array}{c} w_{sk} \in hp(w_{ij}), \\ w_{ij}^{maxS} < w_{sk}^{maxF}, \\ w_{sk}^{minS} < w_{ij}^{maxF} \\ w_{sk} \notin w_{ij}^{pmtor}, w_{sk} \notin w_{ij}^{excl} \end{array} \right. \right\}$$

— множество работ из $hp(w_{ij})$ которые могут вытеснять работу w_{ij}. В отличие от базового алгоритма STBA [5], во множестве $Y_{w_{ij}}$ используются строгие неравенства. Авторами доказано, что переход от нестрогих неравенств к строгим не нарушает консервативности получаемых оценок. Данная модификация позволяет улучшить точность получаемых оценок за счет того, что во множество $Y_{w_{ij}}$ попадает меньше работ.

Между уравнениями 1–12 есть циклическая зависимость, поэтому уравнения решаются итеративным образом до тех пор, пока решение не сойдётся к фиксированному набору значений.

На первой итерации нужно инициализировать величины $w_{ij}^{minS}, w_{ij}^{maxS}, w_{ij}^{minF}, w_{ij}^{maxF}$ следующими начальными значениями: $w_{ij}^{minS} = w_{ij}^{minA}, w_{ij}^{maxS} = w_{ij}^{maxA}$, $w_{ij}^{minF} = w_{ij}^{minS} + C_i^b$, $w_{ij}^{maxF} = w_{ij}^{maxS} + C_i$. Вычисление временных границ для каждой работы стоит проводить в следующем порядке: w^{minA}, w^{minS}, w^{minF}, w^{maxA}, w^{maxS}, w^{maxF}. После вычисления всех временных границ каждой работы, можно вычислить наихудшее время отклика исследуемой задачи τ_k по следующей формуле:

$$WCRT = \max_{W_k}(w_{kj}^{maxF} - T_k * (j-1))$$

Наихудшее время отклика исследуемой задачи, а также временные границы для интервалов $(w_{ij}^{minS}, w_{ij}^{maxS})$, $(w_{ij}^{minF}, w_{ij}^{maxF})$, получаемые с помощью данного алгоритма — вычисляются консервативно, то есть реальные значения времени начала выполнения и завершения выполнения работы w_{ij} не выходят за рамки данных интервалов.

2.2 Адаптация алгоритма STBA под системы ИМА

Для адаптации метода STBA [5] под системы ИМА можно воспользоваться идеей подхода [7] с некоторыми изменениями.

В алгоритме STBA все вычисления привязаны не к задачам, а к работам задач. Поэтому, как показано на рис. 1, для каждой работы из раздела P_i, окна других разделов, привязанных к тому же процессорному ядру (или же просто интервалы времени, которые не входят в окна раздела P_i), можно рассматривать как работы с более высоким приоритетом.

Например, интервал времени между двумя последовательными окнами раздела P_i, который начинается в момент времени $minA$, обозначим за работу w_q, тогда $w_q^{minA} = w_q^{maxA} = w_q^{minS} = w_q^{maxS} = minA$, $C_q^b = C_q$, где C_q — длительность данного интервала времени, $w_q^{minF} = w_q^{maxF} = minA + C_q$. Аналогично вычисляются границы и для следующего подобного интервала w_{q+1}.

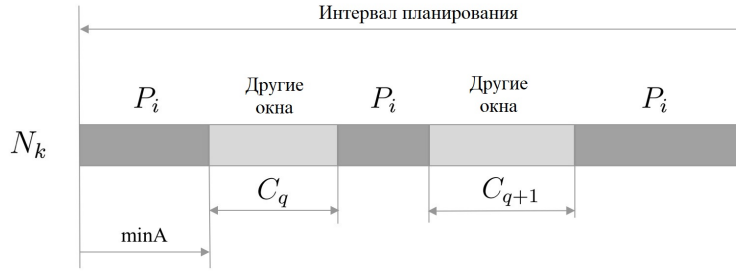


Рис. 1: Адаптация STBA для учета окон разделов

Таким образом, такие работы w_q и w_{q+1} необходимо предварительно добавить во множество $hp(w_{sk})$ для каждой работы w_{sk} из раздела P_i.

Так как работы задач из разных разделов не могут вытеснять друг друга, то можно модифицировать множество $hp(w_{ij})$ следующим образом: $hp(w_{ij}) = \{w_{sk} \mid part(\tau_s) = part(\tau_i),\ pr_s > pr_i\}$, где $part(\tau_i)$ — раздел, которому принадлежит задача τ_i.

Также стоит учесть передачу сообщений $Msg_i \in Msg$ между задачами. Перед началом выполнения алгоритма, каждое передаваемое сообщение $Msg_i \in Msg$ будем считать отдельной задачей $\tau_{\hat{i}}$, где $\hat{i} = |\hat{T}| + i$, выполняющейся на своем отдельном процессорном ядре и обладающей следующими характеристиками: $C_{\hat{i}}^b = dn_i^{min}$; $C_{\hat{i}} = dn_i^{max}$; $T_{\hat{i}} = T_p$, где $\tau_p = S_i$. Причем

$pred[\tau_{\hat{i}}] = \{S_i\}$, $succ[\tau_{\hat{i}}] = \{R_i\}$, где $pred[\tau_{\hat{i}}]$ и $succ[\tau_{\hat{i}}]$ — множества непосредственных предшественников и преемников задачи $\tau_{\hat{i}}$ соответственно. Стоит отметить, что приоритет такой задачи не важен, так как она выполняется на своем отдельном процессорном ядре.

Вычисление временных границ для работ $w_{\hat{i}j} \in W_{\hat{i}}$ происходит по следующим формулам: $w_{\hat{i}j}^{minA} = w_{pj}^{minF}$, $w_{\hat{i}j}^{maxA} = w_{pj}^{maxF}$, $w_{\hat{i}j}^{minS} = w_{\hat{i}j}^{minA}$, $w_{\hat{i}j}^{maxS} = w_{\hat{i}j}^{maxA}$, $w_{\hat{i}j}^{minF} = w_{\hat{i}j}^{minS} + C_{\hat{i}}^{b}$, $w_{\hat{i}j}^{maxF} = w_{\hat{i}j}^{maxS} + C_{\hat{i}}$, где $\tau_p = S_i$.

Утверждение: После применения данной модификации, получаемые оценки для любой работы $w_{ij} \in W$ из любого раздела P_k остаются консервативными в заданной конфигурации системы ИМА.

Доказательство: Докажем для $maxF$. Достаточно показать, что вычисленная граница w_{ij}^{maxF} может принадлежать только окнам раздела P_k.

Допустим, что существует такая работа w_p соответствующая интервалу времени между двумя последовательными окнами раздела P_k и $w_p^{start} < w_{ij}^{maxF} < w_p^{finish}$. Но тогда работа w_p будет принадлежать множеству $Y_{w_{ij}}$ работы w_{ij}. Следовательно $w_{ij}^{maxF} = w_{ij}^{maxF} + w_p^{finish} - w_p^{start}$.

Исходя из начального предположения о том, что $w_p^{start} < w_{ij}^{maxF}$ следует, что $w_{ij}^{maxF} > w_p^{finish}$. Это противоречит неравенству $w_p^{start} < w_{ij}^{maxF} < w_p^{finish}$. Данное противоречие доказывает утверждение.

Для остальных оценок доказательство аналогичное.

3. Экспериментальное исследование

3.1 Цели исследования и описание наборов данных

Целями экспериментального исследования являются:

а) Сравнение алгоритмов STBA и WCDO по точности получаемых оценок;

б) Исследование разницы между верхними оценками, получаемыми с помощью метода STBA и гарантированно достижимыми нижними оценками WCRT, получаемыми с помощью имитационной модели [8].

Исследование проводилось на следующих видах конфигураций:

1. Искусственно сгенерированные конфигурации, которые основаны на топологиях, приближенных к реальным [4].

2. Конфигурации, соответствующие реальным данным. Была рассмотрена одна конфигурация, полученная из описания конфигурации реальной системы [2]. В данной конфигурации примерно $n = 150$ задач и $l = 100$ сообщений, привязанных к 6 процессорным ядрам.

Для каждой пары (конфигурация + задача) из наборов 1 и 2 выполнялся поиск верхней оценки WCRT с помощью алгоритмов STBA и WCDO, а также выполнялся поиск достижимой нижней оценки WCRT с помощью имитационного моделирования (SM). Совпадение верхней оценки WCRT, получаемой с помощью метода STBA, с такой достижимой нижней оценкой говорит о том, что эта оценка является точной. Если эти оценки не совпадают, то нельзя сделать вывод об их (не)точности.

3.2 Результаты экспериментов

В таблицах 1 и 2 представлены результаты экспериментального исследования. В данном исследовании для каждой конфигурации были рассмотрены два случая: BCET = 0 и BCET = 0.5 * WCET. Где [BCET, WCET] — интервал длительности выполнения работ задачи.

В таблице 1 представлены результаты исследования на искусственно сгенерированных конфигурациях. В каждой ячейке таблицы содержится значение отношения $\frac{WCRT_{Alg}}{WCRT_{SM}}$ (его максимальное или среднее значение), где $WCRT_{Alg}$ — наихудшее время отклика, полученное соответствующим алгоритмом, а $WCRT_{SM}$ — наихудшее время отклика, полученное с помощью имитационного моделирования. В столбце avg представлено среднее значение отношения $\frac{WCRT_{Alg}}{WCRT_{SM}}$ взятое по всем задачам из соответствующей конфигурации, тогда как в столбце max представлено максимальное значение данного отношения.

В таблице 2 представлены результаты исследования на конфигурации, соответствующей реальным данным. В каждой ячейке таблицы представлено значение отношения $\frac{WCRT_{Alg}}{WCRT_{SM}}$ для конкретной задачи. В таблице присутствуют все задачи, в которых расхождение оценок, полученных методами STBA и SM, превышает 1%. Остальные задачи были выбраны случайным образом, при этом на большинстве задач значения WCRT совпали.

Конфигурация	BCET = 0.5 * WCET				BCET = 0			
	STBA		WCDO		STBA		WCDO	
	avg	max	avg	max	avg	max	avg	max
KBO5	1.0	1.0	1.05	1.26	1.0	1.0	1.05	1.26
TREE5	1.0	1.0	1.83	2.85	1.0	1.0	1.83	2.85
FAR20	1.04	1.26	2.78	23.8	1.04	1.26	2.78	23.8
KBO20	1.0	1.0	1.67	3.09	1.33	4.3	1.86	4.30
TREE20	1.0	1.13	2.32	9.27	1.0	1.13	2.32	9.27
FAR38	1.06	1.39	2.20	4.10	1.07	1.39	2.27	4.50
KBO38	1.09	1.74	1.47	2.24	1.15	1.74	1.47	2.24
TREE38	1.01	1.39	2.46	4.45	1.03	1.81	2.46	4.45
FAR56	1.09	1.71	2.47	4.50	1.10	1.71	2.58	10.02
KBO56	2.17	4.3	3.04	13.88	2.17	4.3	3.04	13.88
TREE56	1.04	1.98	2.36	4.48	1.04	1.98	2.36	4.48
FAR101	1.17	1.91	2.10	7.65	1.18	1.91	2.19	11.54
KBO101	1.37	3.47	2.71	11.92	2.71	11.92	3.98	76.66
TREE101	1.08	3.08	3.61	11.14	1.10	3.08	3.61	11.14

Таблица 1: Сравнение SM, STBA и WCDO на искусственно сгенерированных конфигурациях

Задача	BCET = 0.5 * WCET		BCET = 0	
	STBA	WCDO	STBA	WCDO
1	1.0	2.44	1.0	2.44
2	1.0	2.81	1.0	2.81
3	1.0	2.61	1.0	2.61
4	1.0	1.7	1.0	1.7
5	1.14	1.62	1.14	1.62
6	1.0	1.44	1.0	1.44
7	1.0	2	1.0	2
8	1.0	2.1	1.0	2.1
9	1.0	2.25	1.0	2.25
10	1.0	1.32	1.0	1.32
11	1.03	3.57	1.04	3.57
12	1.15	2.11	1.15	2.11
13	1.0	2.19	1.0	2.19
14	1.0	1.73	1.0	1.73

Таблица 2: Сравнение SM, STBA и WCDO на конфигурации соответствующей реальным данным

Эксперименты показали, что модифицированный алгоритм

STBA оказался значительно точнее алгоритма WCDO как на искусственно сгенерированных данных, так и на конфигурации соответствующей реальным данным.

Также алгоритм STBA показал высокую точность при исследовании разницы между верхними оценками получаемыми с помощью метода STBA и нижними оценками получаемыми с помощью SM. На большинстве задач из рассмотренных конфигураций данные оценки либо совпадали, либо отличались незначительно. Следовательно, можно сделать вывод о том, что STBA показал точное или близкое к нему значение WCRT на большинстве наборов входных данных.

Заключение

В ходе выполнения данной работы было выбрано несколько подходов к решению задачи оценки WCRT: метод STBA и метод WCDO. Проведена модификация метода STBA, а также выполнена его адаптация под системы ИМА, которая позволяет учитывать окна разделов и интервальную неопределённость длительности передачи сообщений. Доказана корректность внесённых изменений, а также разработаны программные средства, реализующие выбранные методы. Проведено экспериментальное исследование реализованных методов на искусственно сгенерированных и реальных данных. Данное исследование показало, что метод STBA значительно точнее алгоритма WCDO, и что на ряде наборов входных данных метод STBA позволяет получить точные оценки.

В качестве дальнейших исследований предлагается адаптировать предложенный метод на случай, когда в разделах используются алгоритмы планирования, отличные от рассмотренного в данной работе (например EDF).

Литература

1. Балашов В. В., Костенко В. А. *Комплексы бортового радиоэлектронного оборудования с архитектурой ИМА* // Программные системы и инструменты. Т 16. Москва, МАКС Пресс, 2016. С 91–95.

2. Balashov V. V., Balakhanov V. A., Kostenko V. A. *Scheduling of computational tasks in switched network-based IMA systems* //

OPT-i 2014-1st International Conference on Engineering and Applied Sciences Optimization, Proceedings. 2014. P. 1001–1014.

3. Балашов В. В., Антипина Е. А. *Итерационная схема планирования вычислений в модульных системах реального времени* // Программные системы и инструменты. Т 19. Москва, МАКС Пресс, 2019. С 18–29.

4. Гонопольский М. Г., Глонина А. Б. *Алгоритм оценки максимального времени отклика задач в многопроцессорных системах с интервальной неопределенностью длительности выполнения работ* // Моделирование и анализ информационных систем. 2020. Т. 27. № 2. С. 218–233.

5. Kim J., Oh H., Choi J., Ha H., Ha S. *A novel analytical method for worst case response time estimation of distributed embedded systems* // Proceedings of the 50th Annual Design Automation Conference. 2013. P. 1–10.

6. Amurrio A. et al. *Response-Time Analysis of Multipath Flows in Hierarchically-Scheduled Time-Partitioned Distributed Real-Time Systems* // IEEE Access. 2020.

7. Palencia J. C., Gonzalez Harbour M., Gutierrez J. J., Rivas J. M. *Response-time analysis in hierarchically-scheduled time-partitioned distributed systems* // IEEE Transactions on Parallel and Distributed Systems. 2016. Т 28. № 7. P. 2017–2030.

8. Bakhmurov A. G., Balashov V. V., Glonina A. B., Pashkov V. N., Smeliansky R. L., Volkanov D. Y. *Simulation modeling based method for choosing an effective set of fault tolerance mechanisms for real-time avionics systems* // Progress in Flight Dynamics, Guidance, Navigation, Control, Fault Detection, and Avionics. 2013. Т 6. P. 487–500.

Селезнев Л. Е., Костенко В. А.

Влияние способа задания начального приближения на эффективность обучения нейронных сетей

Введение

Нейронные сети широко применяются во множестве задач включая распознавание образов, аппроксимация функциональных зависимостей и прогнозирование. Важную роль при обучении нейронной сети играет выбор способа задания начального приближения (начальных значений весов и параметров сдвига) сети. Неудачный выбор алгоритма инициализации начального приближения может привести к полной необучаемости сети или же значительно сократить время и число шагов алгоритма обучения сети. Независимо от области применения при построении сети необходимо решить следующие задачи: задачу выбора архитектуры (число слоёв, а также количество и типы нейронов в каждом слое), задачу инициализации начального приближения сети и задачу обучения сети (подбора требуемых весовых порогов, при которых сеть удовлетворяет заданным требованиям). Основные исследования проводились для многослойных персептронов, но не исследовалась зависимость качества обучения от типов функций активации в сети. В работе приводятся обзор и результаты сравнения эффективности применения различных алгоритмов инициализации начального приближения в том числе для сетей с различными функциями активации.

1. Задача и алгоритм построения сети

Нейронные сети могут иметь совершенно разную архитектуру и состоять из различных компонентов. В данной работе предлагается провести сравнение качества обучения классических полносвязных однородных нейронных сетей прямого распространения [1], рассматриваемых для решения задач классификации изображений из датасетов MNIST [2], Fashion Mnist [3], Cifar 10 [4].

Начальная сеть задаётся в виде набора слоёв, описывающих число нейронов каждого слоя, а также функции активации [5] всех слоёв со второго по последний. Для заданного набора гиперпараметров сети строится полносвязная однородная

нейронная сеть прямого распространения. Параметры (включающие в себя веса нейронов и величины смещения) сети инициализируются согласно некоторому алгоритму. После этого сеть обучается алгоритмом, основанном на методе градиентного спуска и производится тестирование точности распознавания для описанных датасетов.

В качестве алгоритмов задания начального приближения в данной работе рассматриваются

1. Алгоритм инициализации нулями.
2. Алгоритм инициализации константным значением,
3. Алгоритм инициализации константой с зависимостью от размера слоя,
4. Алгоритм случайной инициализации с помощью случайных значений с равномерным распределением,
5. Алгоритм случайной инициализации с помощью случайных значений с нормальным распределением,
6. Алгоритм Xavier [6],
7. Алгоритм Kaiming [7]

Для каждого из перечисленных алгоритмов производится построение и обучение сети с последующим тестированием на тестовой выборке из указанных датасетов. Сети сравниваются по проценту правильно классифицированных изображений.

2. Алгоритм инициализации нулями и константными значениями

Алгоритм инициализации нулями, а также алгоритм инициализации константными значениями являются самыми простыми с точки зрения реализации и вычислительной сложности. Для данных алгоритмов все параметры сети (веса и параметры смещения нейронов) инициализируются нулём в случае инициализации нулями или некоторым заранее заданным числом (константой) обычно в интервале $(-1, 1)$.

Реализация на языке python с использованием библиотеки numpy [8] для одного слоя выглядит следующим образом:

```
params = numpy.full(
    shape=(layer_size),
    fill_value=constant_value,
)
```

где $layer\_size$ - размер слоя, $constant\_value$ – заданное значение.

Очевидным недостатком данных алгоритмов является то, что в случае, когда все параметры сети равны нулю, основанные на градиентных методах алгоритмы обучения не будут изменять параметры сети во время итерации корректирования весов и сеть не будет обучаться. В случае инициализации константой, сеть будет обучаться и менять значение параметров, однако все нейроны слоя будут эквивалентны всего одному нейрону и такая сеть будет неэффективной.

3. Алгоритм инициализации константными значениями с зависимостью от размера слоя

Отличие алгоритма инициализации константой с зависимостью от размера слоя состоит в том, что в этом случае заранее заданное число (константа) делится на число нейронов слоя, параметры которого инициализируются в данный момент. В основном константа имеет значение 0.5.

Реализация на языке python с использованием библиотеки numpy для одного слоя выглядит следующим образом:

```
params = numpy.full(
    shape=(layer_size),
    fill_value=constant_value / layer_size,
)
```

где $layer\_size$ - размер слоя, $constant\_value$ – заданное значение.

Преимуществом данного алгоритма является равномерность среднего значения весов в каждом слое. При такой инициализации сумма весов в среднем будет равна константе, заданной для этого алгоритма, что позволит избежать проблемы затухания или роста градиента ошибки [9].

4. Алгоритм инициализации случайными значениями с равномерным распределением

Алгоритм инициализации случайными значениями с равномерным распределением задаёт параметры сети как значения случайной величины, равномерно распределенной на интервале. Зачастую интервал выбирается как один из двух: $(0, 1)$ или $(-1, 1)$, однако в общем случае он может быть любым.

Реализация на языке python с использованием библиотеки numpy для одного слоя выглядит следующим образом:

```
params = numpy.random.uniform(
    size=(prev_size, layer_size),
)
```

где $layer\_size$ - размер слоя, $prev\_size$ – размер предыдущего слоя.

Преимуществом данного алгоритма является уход от проблемы недообучения (или вовсе необучаемости) сети, свойственной для предыдущего алгоритма инициализации. В данном случае все нейроны имеют различные параметры и их значения будут приобретать более различные изменения на каждом этапе коррекции параметров.

5. Алгоритм инициализации случайными значениями с нормальным распределением

Алгоритм инициализации случайными значениями с нормальным распределением задаёт параметры сети как значения случайных величин с нормальным распределением с некоторыми параметрами. В основном используется стандартное нормальное распределение с мат. ожиданием и дисперсией равными 0 и 1 соответственно, однако возможно использование и другого математического ожидания и дисперсии.

Реализация на языке python с использованием библиотеки numpy для одного слоя выглядит следующим образом:

```
params = numpy.random.normal(
    size=(prev_size, layer_size),
)
```

где $layer\_size$ - размер слоя, $prev\_size$ – размер предыдущего слоя.

Преимуществом данного алгоритма также является уход от проблемы недообучения сети, когда все нейроны ведут себя как один, а также уменьшение первичных колебаний значений параметров, так как в отличии от нормального распределения, значения почти всех параметров сосредоточены в окрестности нуля, что обеспечит меньшие изменения значений параметров за счёт меньшей чувствительности параметра к величине ошибки. Такая инициализация может позволить за меньшее число итераций попасть в область локального минимума ошибки при применении градиентных методов обучения.

6. Алгоритм Xavier

Алгоритм инициализации Xavier задаёт параметры сети с учетом размера каждого слоя сети. Параметры сети вычисляются относительно случайных величин с равномерным или нормальным распределением, а также размера предыдущего (по отношению к инициализируемому) и текущего слоя сети. Алгоритм может использовать как нормальное распределение случайных значений, так и равномерное, соответственно Xavier normal и Xavier uniform.

Преимуществом алгоритма является простота реализации и уменьшение интервала равномерного или дисперсии нормального распределения, что снижает первичные колебания весов на начальных этапах обучения. Помимо простоты, алгоритм позволяет увеличить точность обучения в ряде задач по сравнению с равномерным распределением. Ввиду того, что алгоритм учитывает число нейронов в каждом слое, при обучении с помощью метода обратного распространения ошибки не происходит затухания или резкого роста градиентов в среднем по слоям. Алгоритм в первую очередь нацелен на улучшение качества обучения для сетей с нелинейными функциями активации как, например, Sigmoid или TanH [5], однако он может оказаться слабо эффективным для инициализации сетей, использующих в роли функции активации кусочно-линейную функцию ReLU. Константные множители в формулах в общем случае используются для решения проблем с функциями активации, которые отбрасывают отрицательные входные значения (Например ReLU

или Sigmoid), для того, чтобы дисперсия выходных значений соотвествовала дисперсии входных значений. Подробное описание данных значений приводится в оригинальной статье с описанием алгоритма.

Реализация Xavier uniform на языке python с использованием библиотеки numpy для одного слоя выглядит следующим образом:

```
a = gain * numpy.sqrt(
    6.0 / (prev_size + layer_size),
)
params = numpy.random.uniform(
    low=-a,
    high=a,
    size=(prev_size, layer_size),
)
```

где *layer_size* - размер слоя, *prev_size* – размер предыдущего слоя, *gain* – параметр усиления, зависящий от типа функций активации.

Реализация Xavier normal на языке python с использованием библиотеки numpy для одного слоя выглядит следующим образом:

```
params = numpy.random.normal(
    scale=gain * numpy.sqrt(
        2.0 / (prev_size + layer_size),
    ),
    size=(prev_size, layer_size),
)
```

где *layer_size* - размер слоя, *prev_size* – размер предыдущего слоя, *gain* – параметр усиления, зависящий от типа функций активации.

7. Алгоритм Kaiming

Алгоритм инициализации Kaiming является улучшением алгоритма Xavier, так как может повысить эффективность для линейных и кусочно-линейных функций активации. Алгоритм может использовать как нормальное распределение случайных значений, так и равномерное, соответственно Kaiming normal и Kaiming uniform. Алгоритм является улучшением алгоритма инициализации Xavier для сетей, использующих в роли функции активации функцию ReLU вместо Sigmoid или TanH. Константные множители в формулах в общем случае используются для решения

проблем с функциями активации, которые отбрасывают отрицательные входные значения (Например ReLU или Sigmoid), для того, чтобы дисперсия выходных значений соотвествовала дисперсии входных значений. Подробное описание данных значений приводится в оригинальной статье с описанием алгоритма.

Реализация Kaiming uniform на языке python с использованием библиотеки numpy для одного слоя выглядит следующим образом:

```
a = gain * numpy.sqrt(3.0 / prev_size)
params = numpy.random.uniform(
    low=-a,
    high=a,
    size=(prev_size, layer_size),
)
```

где *layer_size* - размер слоя, *prev_size* – размер предыдущего слоя, *gain* – параметр усиления, зависящий от типа функций активации.

Реализация Kaiming normal на языке python с использованием библиотеки numpy для одного слоя выглядит следующим образом:

```
params = numpy.random.normal(
    scale=gain * numpy.sqrt(
        1.0 / prev_size
    ),
    size=(prev_size, layer_size),
)
```

где *layer_size* - размер слоя, *prev_size* – размер предыдущего слоя, *gain* – параметр усиления, зависящий от типа функций активации.

8. Определение параметра усиления (gain) для типов функций активации

Параметр усиления (gain), используемый в алгоритме Xavier и Kaiming требуется для подстройки алгоритма под используемые типы функций активации в сети [10]. Изменение значения данного параметра в зависимости от используемых в сети типов функций активации может как повысить качество обучения после инициализации, так и понизить, а потому для рассматриваемых функций активации ReLU и Sigmoid в указанной ранее статье предлагается использовать значения параметра gain соответственно $\sqrt{2}$ и 1.

9. Сравнение эффективности алгоритмов задания начального приближения

Сравнение производится на задачах классификации изображений из датасетов MNIST, FashionMNIST, Cifar 10. Для каждого датасета рассматривается своя архитектура нейронной сети (В общем случае полносвязная однородная нейронная сеть прямого распространения).

В случае классификации рукописных цифр и элементов одежды из датасетов MNIST и FashionMNIST рассматриваются сети со следующей структурой:

- Входной слой сети имеет размер 784 нейрона (изображение 28x28 пикселей),
- Два скрытых слоя сети имеют размеры 100 и 50 нейронов,
- Размер выходного слоя – 10 нейронов, что соответствует 10 различным классам.

В случае классификации цветных изображений из датасета Cifar-10 рассматривается следующая архитектура сети:

- Входной слой имеет размер 3072 нейрона (трехцветное изображение 32x32 пикселя),
- Два скрытых слоя сети имеют размеры 100 и 50 нейронов,
- Размер выходного слоя – 10 нейронов, что соответствует 10 различным классам.

При тестировании используется функция потерь Cross Entropy Loss [11], а также для каждой сети проверяется сравнение эффективности применения каждого алгоритма инициализации с разными функциями активации в сети (В качестве функций активации используется либо ReLU, либо Sigmoid). Данный подход требуется для проверки предположения об эффективности алгоритма инициализации Kaiming перед алгоритмом Xavier в случае сетей с функциями активации ReLU.

Построенная и инициализированная нейронаня сеть обучается с помощью алгоритма оптимизации ADAM [12], являющегося развитием метода стохастического градиентного спуска [13]. В качестве числа пакетов для обучения берется 64, число эпох - 10, коэффициент обучения 0.001. Данный алгоритм обучения часто

применяется в задачах машинного зрения и задачах классификации изображений. В качестве оценки точности обучения используется процент верно распознанных примеров из тестовой выборки.

Initializer	MNIST ReLU	MNIST Sigmoid	Fashion MNIST ReLU	Fashion MNIST Sigmoid	Cifar-10 ReLU	Cifar-10 Sigmoid
Zero	11.35%	89.11%	10.00%	79.20%	10.00%	33.48%
Const 0.5	11.35%	11.35%	10.01%	30.78%	16.05%	19.62%
Layer dependent Const 0.5	11.35%	81.05%	82.56%	81.62%	36.35%	32.69%
Uniform	11.35%	11.35%	18.89%	74.34%	19.69%	21.70%
Normal	11.35%	9.92%	25.64%	73.50%	19.36%	21.97%
Xavier uniform	97.06%	97.20%	87.39%	87.25%	50.58%	47.36%
Xavier normal	97.11%	97.19%	**87.68%**	**87.75%**	50.83%	47.78%
Kaiming uniform	**97.30%**	**97.32%**	87.41%	87.54%	50.78%	**47.99%**
Kaiming normal	96.97%	97.22%	87.64%	87.64%	**50.86%**	47.12%

Таблица 1: Результаты тестирования алгоритмов инициализации

Усредненные результаты тестирования рассмотренных алгоритмов при описанных условиях представлены в таблице 1. Для каждого датасета представлены по два значения метрики: точность при использовании функции активации ReLU и точность при использовании функции активации Sigmoid. Дополнительно жирным шрифтом выделены максимальные знаяения точности ждядля каждой из задач.

Как показали результаты экспериментального исследования, приведенные в таблице 1, алгоритм инициализации константными значениями оказался неэффективен для сетей с функциями активации ReLU. Алгоритмы инициализации константными значениями (Zero, Const 0.5) и случайными величинами с нормальным и равномерным распределением (Uniform, Normal) в общем случае оказались неэффективны для инициализации сетей, позволив поднято точность выше 50% только для двух случаев из 12 (для случайной инициализации). Алгоритм инициализации константными значениями с учетом размера слоя (Layer dependent

Const 0.5) оказался эффективнее алгоритма инициализации константами и случайными значениями, позволив поднять качество обучения до 80% в ряде случаев. Как и ожидалось ранее, алгоритм Xavier и Kaiming оказались эффективными для инициализации сетей в задачах классификации, при этом алгоритм Kaiming оказался более эффективен для сетей с функцией активации ReLU, а алгоритм Xavier – для сетей с сигмоидальными функциями активации. При этом оба алгоритма значительно повлияли на качество обучения по сравнению со случайной и константной инициализацией, повысив точность классификации даже на датасете Cifar-10, где сеть изначально была недостаточного размера.

Заключение

Как показало экспериментальное исследование, наиболее эффективными оказались алгоритм Xavier для инициализации сетей с нелинейными функциями активации и алгоритм Kaiming для инициализации сетей с кусочно-линейной функцией активации ReLU. Третьим по эффективности после алгоритмов Xavier и Kaiming оказался алгоритм инициализации константным значением с учетом размера слоя, позволив значительно улучшить качество обучения сети. Следующими по эффективности оказались алгоритмы, инициализирующие параметры сети случайными значениями с нормальным и равномерным распределением. Самыми неэффективными оказались алгоритмы инициализации константным значением уступив алгоритму инициализации нулями.

Таким образом, наиболее эффективным для инициализации начального приближения для полносвязных однородных нейронных сетей в задачах классификации можно считать алгоритмы Xavier и Kaiming.

Литература

1. Hristev R. M. *Matrix Techniques in Artificial Neural Networks*: дис. ... канд. мат. Canterbury, 2000.

2. Deng L. *The mnist database of handwritten digit images for machine learning research* // IEEE Signal Processing Magazine. 2012. №29(6). С. 141–142.

3. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms* // arxiv.org URL: https://arxiv.org/abs/1708.07747 (дата обращения: 10.09.2021).

4. Krizhevsky A., Hinton G. *Learning multiple layers of features from tiny images* // Master's thesis. Toronto: Department of Computer Science, 2009.

5. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning* // arxiv.org URL: https://arxiv.org/abs/1811.03378 (дата обращения: 10.09.2021).

6. Xavier G., Yoshua B. *Understanding the difficulty of training deep feedforward neural networks* // Proceedings of the thirteenth international conference on artificial intelligence and statistics. Sardinia: PMLR, 2010. С. 249-256.

7. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* // arxiv.org URL: https://arxiv.org/abs/1502.01852 (дата обращения: 10.09.2021).

8. *Numpy* // numpy.org URL: https://numpy.org (дата обращения: 10.09.2021).

9. *Understanding the exploding gradient problem* // arxiv.org URL: https://arxiv.org/abs/1211.5063 (дата обращения: 10.09.2021).

10. Günter K., Thomas U., Andreas M., Sepp H. *Self-Normalizing Neural Networks* // Advances in Neural Information Processing Systems. 2017. №30. С. 971-980.

11. Kevin M. *Machine Learning: A Probabilistic Perspective.* ISBN 978-0262018029 изд. MIT Press, 2012.

12. *Adam: A Method for Stochastic Optimization* // arxiv.org URL: https://arxiv.org/abs/1412.6980 (дата обращения: 10.09.2021).

13. *An overview of gradient descent optimization algorithms* // arxiv.org URL: https://arxiv.org/abs/1609.04747 (дата обращения: 10.09.2021).

Синицын А.А., Степанов Е.П., Писковский В.О.

Высокоуровневая система разработки приложений для программно-конфигурируемых сетей под управлением контроллера RunOS v2.0

Введение

В данной статье рассмотрен опыт реализации приложения Maple [1] для программно-конфигурируемых сетей под управлением контроллера RunOS v2.0 [2]. Приложение Maple позволяет использовать высокоуровневый язык программирования для разработки ПКС приложений, что позволяет описывать сетевые политики, эффективно устанавливать правила на коммутаторы, оптимизировать количество обращений к контроллеру. Это способствует унификации программирования приложений для ПКС сетей, а также и ускоряет их разработку: система берет на себя работу с низкоуровневыми деталями протокола OpenFlow [3]. Формализованы принципы и подходы к написанию сетевых служб и приложений, которые управляют логикой работы сети. В статье также описаны различия двух архитектур контроллера RunOS v0.6 и v2.0, а также их программная организация с точки зрения реализации приложения Maple. Работа является первым шагом в реализации систем разработки приложений маршрутизации потока fNetKAT для программно-конфигурируемых сетей под управлением RunOS v2.0.

1. Обзор контроллера RunOS

Проект RunOS [4] был направлен на разработку сетевой операционной системы, программного интерфейса для сетевых приложений, а также реализацию сетевых служб и основных функций интерфейса.

В ходе проекта RunOS было разработано три варианта архитектуры контроллера (см. рис. 1) [5].

Первый вариант был реализован полностью в пространстве пользователя (user-space) ОС Linux. Такой подход чреват накладными расходами на сетевое взаимодействие и работу с

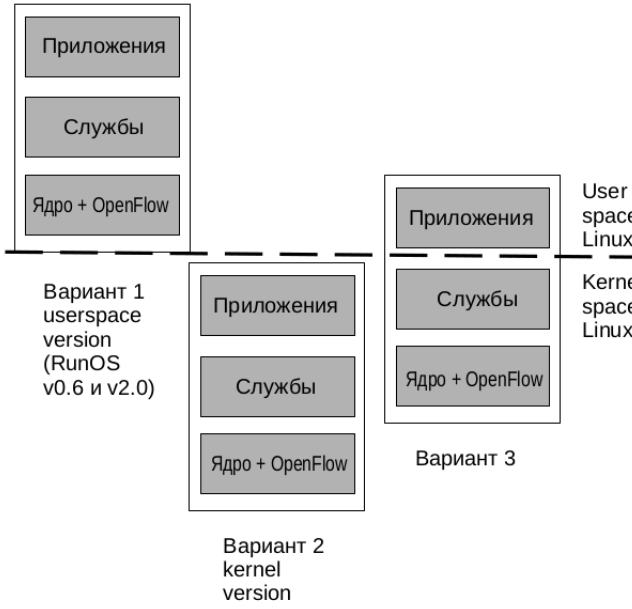


Рис. 1: Варианты архитектуры контроллера RunOS

памятью системы. Из достоинств, это быстрое и удобное создание сетевых приложений. Как оказалось, так организованы большинство существующих ПКС-контроллеров [5]. К данному варианту контроллеров относится RunOS v0.6 и v2.0. На данный момент RunOS v2.0 это последняя версия контроллера.

Второй вариант архитектуры контроллера полностью реализован в пространстве ядра (kernel-space) ОС Linux. Такой подход позволил избежать накладных расходов на взаимодействие с сетью и памятью, обеспечивая тем самым, самую высокую производительность. Однако разработка в ядре операционной системы Linux требует больше усилий на разработку в связи с:

- **Отсутствием библиотек** разработки в ядре Linux, написанных на высокоуровневом языке программирования;
- **Сборкой**: возникнет нужда пересобирать все ядро Linux, а не только модуль;
- **Отладкой**: нельзя просматривать работу ядра ОС по одной инструкции и сложнее следить за потреблением памяти;

- **Внедрением** и **поддержкой**: требуется обновление ядра ОС.

Для объединения достоинств обоих подходов разработан третий вариант архитектуры контроллера, который соединил в себе высокую производительность и удобство разработки приложений. В этом варианте в user-space была вынесена вся логика приложений, а в kernel-space осталась поддержка OpenFlow сессий с коммутаторами, обработка и анализ OpenFlow сообщений и выполнение таких сервисов, как, например, построение топологии. Такой подход к организации работы контроллера позволил наряду с высокой функциональностью и удобством по разработке приложений достичь высокой производительности.

Подробное описание архитектуры контроллера RunOS можно найти в [5]. Контроллер RunOS использует в своей работе проект libfluid [6] для реализации OpenFlow-контроллера [2]. Этот проект также используется в функциональных тестах контроллера. Проект libfluid состоит из двух независимых библиотек:

- **libfluid_base** — библиотека, которая содержит набор классов для создания OpenFlow-сервера. Библиотека libfluid_base основана на использовании стандартной клиент-серверной архитектуры, в которой контроллер исполняет роль сервера, а коммутаторы — клиентов. С помощью этой библиотеки сервер устанавливает соединение с устройствами по протоколу OpenFlow и обрабатывает события;

- **libfluid_msg** — библиотека, которая содержит набор классов для создания, анализа и обработки OpenFlow-сообщений.

Эти библиотеки могут быть скомпилированы и использованы независимо друг от друга. Соответственно libfluid_base может использоваться с любой другой библиотекой обмена OpenFlow-сообщениями, а libfluid_msg может быть интегрирована в любую другую реализацию контроллера.

2. Службы и приложения контроллера Runos v.2.0

Следует различать понятия сетевой службы и сетевого приложения:

- **Сетевые службы** ПКС-контроллера — отдельные обязательные модули контроллера, позволяющие реализовать определенную функциональность;

- **Сетевые приложения** ПКС-контроллера — отдельные необязательные модули контроллера, позволяющие реализовать определенную функциональность.

Сетевые службы обязательны для базового функционирования контроллера, сетевые приложения опциональны и пишутся под отдельные задачи. Службы контроллера позволяют вынести функциональность в отдельные модули, что упрощает создание новых приложений. Некоторые службы обязательны для работы всех приложений (например, загрузка приложений Loader и интерфейс для взаимодействия между ними), без них приложение даже не запустится. Другие службы не обязательны, но могут использоваться в работе некоторых приложений, например служба Topology предоставляет информацию об изменении топологии сети. Службы контроллера могут быть реализованы как в виде отдельных загружаемых модулей, так и непосредственно в ядре контроллера. Реализованная архитектура позволит в будущем запускаться модулю сразу вместе с контроллером, в случае если служба представляет собой отдельный модуль, так и динамически загружаться во время работы контроллера.

С точки зрения фунциональности службы контроллера можно разделить условно на пять групп:

1) Службы, обеспечивающие взаимодействие с коммутаторами и сбор статистики по коммутаторам (OFServer, OpenFlow Message Sender, Controller, Dpid Checker, Switch Manager, Switch Ordering, Stats Bucket Manager, Stats Rules Manager);

2) Службы для загрузки модулей, обеспечения взаимодействия между ними и управление службами извне (Loader, Rest-Listiner, Command Line, Database Connector);

3) Службы для работы с топологией сети (Topology, Link Discovery);

4) Служба, предназначенная для обеспечения отказоустойчивости служб и приложений (Recovery Manager);

5) Прочие службы (Log, Pcap).

Реализации данных служб в контроллере RunOS v.2.0 находятся в ядре контроллера.

С точки зрения фунциональности приложения контроллеров можно разделить условно на четыре группы:

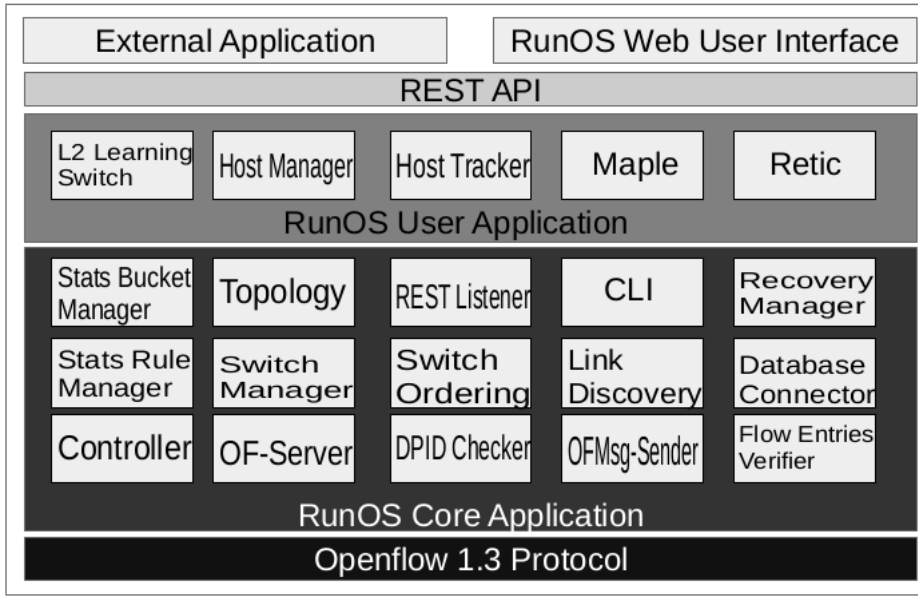


Рис. 2: Архитектура контроллера RunOS v.2.0

1) Приложения, реализующие простейшие правила коммутации и маршрутизации пакетов в сети (Maple, Retic, Hub, L2 Learning Switch, Static Flow Pusher, SPAN);

2) Приложения для работы с распространенными сетевыми протоколами (ARP, DNS, DHCP, Authentication);

3) Приложения для управления трафиком и сбора статистики в региональной сети передачи данных (Firewall, Load Balancer, NAT);

4) Приложения для поддержки виртуализации сетей и управления трафиком в ЦОД (Virtualization, Enterprise, C2-Cube).

Реализации данных приложений закрыты, кроме приложений L2 Learning Switch [7] и Host Manager [8], которые открыты в образовательных целях. Основываясь на их структуре, можно написать свое приложение.

На рис. 2 представлена архитектура контроллера RunOS v2.0. На нем обнозначено пространство пользовательских приложений, в

котором размещены сетевые приложения (L2 Learning Switch, Host Manager, Maple, etc), а ниже обозначено пространство ядра контроллера, в котором реализованы сетевые службы (Controller, OF-Server, Topology, etc). Они связаны через северный интерфейс, а южный интерфейс соединяет пространство ядра с протоколом OpenFlow 1.3, то есть для связи с устройствами в контуре передачи сообщений, а также управление сетевыми устройствами [9].

Так как работа ядра контроллера RunOS v2.0 достаточно оптимизирована, то для написания приложения в пространстве ядра контроллера нужно убедиться, что это не будет мешать его функционированию. Рекомендуется реализовывать свои приложения в пространстве пользовательских приложений, потому что реализуя приложение в пространстве ядра, к нему будут иметь доступ другие приложения как к сетевой службе, что не всегда нужно.

3. Написание приложения для контроллера RunOS v2.0

Структура любого приложения имеет вид (см. рисунок 3) [10].

Директория docs нужна для документации и содержит файл app.rst для автоматического создания документации в Sphinx. Директории include и src содержат исходные коды приложения: заголовочные файлы (*.hpp) и файлы с реализацией (*.cc). Содержимое директории include будет видно для других приложений (является интерфейсом, который можно предоставить для других сетевых приложений контроллера). Содержимое директории src относится только к рассматриваемому сетевому приложению. CMakeLists.txt содержит описание правил сборки на формальном языке CMake, conanfile.py нужен для управления зависимостями C++. README.md может содержать описание работы приложения и инструкции для запуска, settings.json содержит настройки конфигурации приложения.

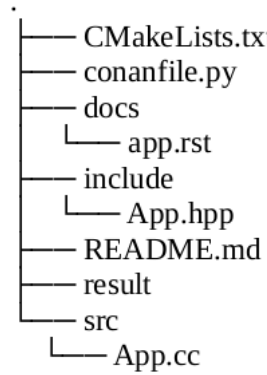


Рис. 3: Структура приложения в контроллере RunOS v.2.0

3.1 Основные методы

Основными методами для любого приложения в контроллере RunOS v2.0 являются:

- REGISTER_APPLICATION(your_class, list_of_dependeces) — регистрация приложения, где your_class — имя вашего класса приложения и list_of_dependeces — список приложений, от которых зависит данное приложение;
- void init(Loader *loader, const Config &config) — инициализация приложения. В этом методе можно извлекать настройки из settings.json. Пользователь может определять значения параметров вызова сетевых служб, которые загружены через settings.json;
- void startUp(class Loader *) — запуск приложения. Данный метод вызывается, когда все приложения проинициализированы;
- LOG() — вывод отладочных сообщений контроллера. Возможны следующие аргументы: INFO, WARNING, ERROR.

Минимальное количество методов для запуска приложения это два: REGISTER_APPLICATION() и void init() или void startUp().

3.2 Пример регистрации приложения

Листинг 1: Пример регистрации приложения

```
namespace runos {

REGISTER_APPLICATION(Maple, {"controller", ""})

// Methods implementation of Maple class

} // namespace runos
```

3.3 Пример использования методов init, starUp и LOG

Листинг 2: Пример использования методов init, startUp и LOG

```
// Maple.hpp
#pragma once
#include "Application.hpp"
#include "Loader.hpp"
#include "Controller.hpp"

namespace runos {

class Maple : public Application {
    Q_OBJECT
    SIMPLE_APPLICATION(Maple, "maple")
public:
    void init(Loader* loader, const Config& config) override;
    void startUp(Loader * loader) override;
}
} // namespace runos

// Maple.cc
void Maple::init(Loader* loader, const Config& root_config)
{
    LOG(INFO) << "This is Maple";
}

void Maple::startUp(Loader*)
{
```

```
    LOG(INFO) << "Registered packet-in handlers: ";
}

LOG(INFO) << "Information message";
LOG(WARNING) << "Warning";
LOG(ERROR) << "Error";
```

После каждой правки кода вашего приложения, нужно пересобирать контроллер.

4. Реализация Maple в контроллере RunOS v2.0

Разработка приложений, которые напрямую управляют коммутаторами с помощью протокола OpenFlow, чрезвычайно трудное занятие: программистам нужно тратить больше времени на работу с низкоуровневыми деталями протокола, чем на решении поставленной задачи. Поэтому создаются системы, которые берут на себя низкоуровневые детали по настройке OpenFlow коммутаторов и позволяют программисту выразить логику своего приложения по управлению сети с помощью высокоуровневых абстракций. Одной из таких систем является Maple [3].

Данная система реализована в виде приложения в контроллере RunOS v2.0 [1], написанном на языке программирования C++17, с использованием библиотек Boost, QT, LibFluid. RunOS v2.0 поддерживает протокол OpenFlow 1.3.

4.1 Пример приложения с использованием абстракции Maple

Maple предоставляет следующий интерфейс для взаимодействия с пакетом:

- прочитать поле пакета: метод *load*;
- проверить поле пакета на соответствие заданному константному значению: метод *test*;
- изменить поле пакета: метод *modify*;
- выбрать действие, которое необходимо произвести над пакетом: переслать пакет на заданный порт или порты, отбросить пакет, отправить по заданному маршруту в сети.

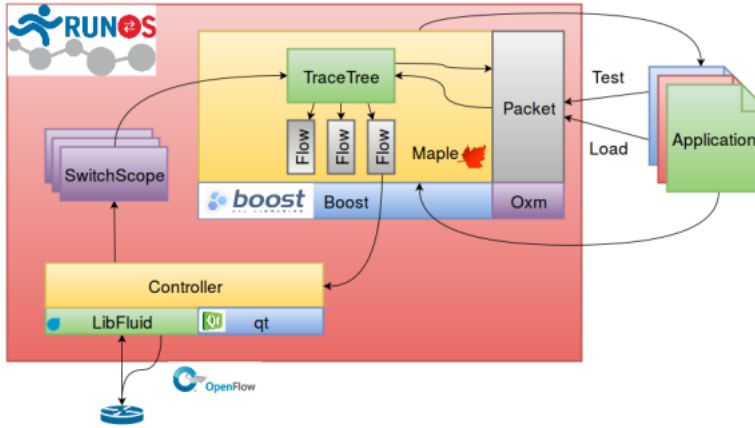


Рис. 4: Структура работы Maple в контроллере RunOS v2.0

Листинг 3: Приложение L2-Learning Switch с абстракцией Maple

```
def pkt_handler(pkt):
        in_port = pkt.load (inPort)
        src = pkt.load (ethSrc)
        dst = pkt.load (ethDst)
        sw = pkt.load (switchId)
        database.learn({sw, in_port}, src)
        route = database.find_route(src, dst)
        if route is found:
                return route
        else:
                return broadcast(timeout=0)
```

В отличии от реализации с использованием абстракции OpenFlow здесь нет необходимости устанавливать правила для каждого отдельного коммутатора, также не надо указывать приоритеты и поля соответствия, они будут выставлены нижележащей системой Maple. В системе Maple реализованы политики генерации правил на коммутаторах. Архитектура Maple и схема работы с контроллере показана на рисунке 4. Схема работы выглядит следующим образом: когда *PacketIn* сообщение приходит на контроллер, оно проходит через все приложения в указанном порядке, во время прохода собирается трасса выполнения функций — последовательность вызовов методов *Test* и *Loa*d, последним элементом трассы является список всех действий, которые надо

выполнить над пакетом. Эта трасса накладывается на внутреннее представление приложений *TraceTree* [2], и далее с помощью алгоритма инкрементального обновления таблицы потоков добавляются новые OpenFlow правила в таблицы коммутаторов.

4.2 Развитие системы: fNetKAT

В работе [11]Результаты тестирования алгоритмов инициализации предпринята успешная попытка совместить системы NetKAT [12] и Maple в систему fNetKAT, сочетающую их преимущества. Существующие системы (NetKAT и Maple) предоставляют высокоуровневый интерфейс разработки приложений, однако имеют недостатки: NetKAT обладает чрезмерными накладными расходами и не интуитивно-понятным синтаксисом, Maple не обладает достаточной мощью для реализации независимых приложений. Предложенный подход позволяет программисту разрабатывать приложения в понятном программисту императивном стиле и гибко управлять композицией независимых приложений. Разработанная система позволяет сократить накладные расходы на пересылку служебных сообщений на 10% по сравнению с системой Maple и на 98% по сравнению с системой NetKAT.

Реализация fNetKAT представлена в контроллере drunos [13]. Контроллер drunos содержит два основных приложения: Maple и Retic и они взаимосвязаны друг с другом: Retic представляет собой реализацию fNetKAT, который использует функционал Maple и NetKAT. Без успешной работы Maple не будет функционировать Retic. Следовательно, первой задачей был перенос Maple в актуальную версию контроллера RunOS v2.0. Основным изменением этой версии является разделение контроллера на уровень пользовательских приложений (APP layer) и уровень ядра контроллера (CORE layer). Ранее все приложения и зависимости с ядром находились в одном месте, соответственно задача была в переносе функциональности максимально без вмешательства в работу ядерных приложений. В результате все перенесено в пространство пользовательских приложений в виде Maple Library и Maple Application. Основной сложностью было отслеживание зависимостей с ядерными приложениями и переписыванием связанных методов.

Заключение

Данная работа посвящена обзору различий контроллеров RunOS v0.6 и v2.0 с точки зрения реализации приложения Maple на текущей, второй версии контроллера и тестирования работы полученного комплекса. В качестве тестирования было использовано приложение SimpleLearningSwitch, которое представляет реализацию L2-LearningSwitch и использует в своей работе абстракции Maple. Создана топология без циклов с помощью Mininet [14] и запущена команда pingall. Были проверены правила на коммутаторах: они соответствуют тем, которые поставил Maple. В ходе работы были описаны принципы написания приложений в контроллере RunOS v2.0 в применении к реализации fNetKAT и изучена реализация Maple в контроллере RunOS v0.6.

Литература

1. Voellmy, J. Wang, Y. R. Yang, B. Ford, P. Hudak *Maple: simplifying SDN programming using algorithmic policies* ACM SIGCOMM Computer Communication Review. — 2013. — Т. 43, No 4. — С. 87—98.

2. *RUNOS SDN/OpenFlow Controller.* [Электронный ресурс]. URL: https://github.com/ARCCN/runos (дата обращения 08.10.2021).

3. *OpenFlow: enabling innovation in campus networks* / N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner // ACM SIGCOMM Computer Communication Review. — 2008. — Т. 38, № 2. — С. 69—74.

4. A. Shalimov, S. Nizovtsev, D. Morkovnik, R. Smeliansky // *The Runos OpenFlow Controller* Software Defined Networks (EWSDN), 2015 Fourth European Workshop on. — IEEE. 2015. — С. 103—104.

5. Р.Л. Смелянский, В.А. Антоненко. *Концепции программного управления и виртуализации сетевых сервисов в современных сетях передачи данных: учеб. пособие* — Москва: КУРС, 2020. — 160 с. — (Современные сетевые технологии).

6. *Libfluid: The ONF OpenFlow driver. Documentation.* [Электронный ресурс].

https://opennetworkingfoundation.github.io/libfluid/ (дата обращения 08.10.2021).

7. *L2 Learning Switch Application for RUNOS Controller* [Электронный ресурс]. https://github.com/ARCCN/l2-learning-switch (дата обращения 08.10.2021).

8. *Host Manager Application for RUNOS 2.0 Controller* [Электронный ресурс]. https://github.com/ARCCN/host-manager (дата обращения 08.10.2021).

9. Скобцова Ю.А., Пашков В.Н. *Исследование и разработка сетевого приложения для распределенной платформы управления в программно-конфигурируемых сетях* Программные системы и инструменты: Тематический сборник № 16, Том: 16, 2016, С. 7—18.

10. *RUNOS Developer Guide* [Электронный ресурс]. http://arccn.github.io/runos/docs-2.0/eng/13_RUNOS_DeveloperGuide.html (дата обращения 08.10.2021).

11. Швецов Д. А. *Высокоуровневая система разработки приложений для программно-конфигурируемых сетей* Ломоносов-2019: Материалы XXVI Международной научной конференции студентов, аспирантов и молодых ученых: секция Вычислительная математика и кибернетика. Сборник тезисов/Сост. Атамась Е.И., Мальцева А.В., Шевцова И.Г. — Издательский отдел факультета ВМК МГУ Москва, 2019. — С. 198—200. — (МГУ имени М.В. Ломоносова, факультет ВМК).

12. C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, D. Walker *NetKAT: Semantic foundations for networks* Acm sigplan notices. — 2014. — Т. 49, No 1. — С. 113—126.

13. *drunos* [Электронный ресурс]. https://github.com/dShvetsov/drunos (дата обращения 08.10.2021).

14. *Mininet: An Instant Virtual Network on your Laptop (or other PC)* [Электронный ресурс]. http://mininet.org/ (дата обращения 08.10.2021).

Танкаев И. Р., Костенко В. А.

Федеративное обучение с использованием схемы простого голосования

Введение

Задача федеративного обучения [1-4] заключается в обучении локальных классификаторов на независимых выборках данных клиентов. Выборки однородны по типам объектов, т.е. объекты, описываются одинаковым набором характеристик (атрибутов) объекта. Объекты выборок не могут передаваться между клиентами, т.е. каждый локальный классификатор может обучаться только на данных одной выборки. Для обучения глобального классификатора используется сервер, который агрегирует параметры локальных классификаторов, полученных от клиентов, а также отправляет им новые параметры глобального классификатора.

Технология федеративного обучения актуальна при работе с закрытыми базами данных, большими базами данных и для периферийных вычислений.

Федеративное обучение уже применялось компанией Google для автодополнения текста набираемого на Google клавиатуре на смартфонах и планшетах [5].

В настоящий момент разрабатывается единая медицинская база данных всей Европы, в которой планируется использование федеративного обучения.

Основной проблемой промышленного применения федеративного обучения является безопасность технологии. Большинство известных алгоритмов обучения [1-2, 6-8] не обладают устойчивостью к перехвату обучения [9]. Под перехватом обучения понимается ситуация, в которой глобальный классификатор делает предсказания, необходимые злоумышленнику. В работе предлагается способ построения глобального классификатора с использованием схемы простого голосования. Приводятся результаты экспериментального исследования его точности в зависимости от числа локальных классификаторов, на которых перехвачено обучение.

1. Постановка задачи и описание алгоритма

Имеется N различных баз данных (БД). БД распределены по клиентам. БД однородны по хранимой информации, но информация между ними передаваться не может. Каждый объект в БД характеризуется набором характеристик (атрибутов) объекта и принадлежит к одному или нескольким классам. Значения атрибутов объекта могут или изменятся со временем, или быть постоянными. Классы, к которым принадлежит объект, также изменяются в зависимости от значений атрибутов, которые могут изменяться. Глобальный классификатор после изменения значений атрибутов объекта должен определять его принадлежность к классам. Также глобальный классификатор должен определять классы новых объектов.

Предложенный алгоритм основывается на использовании схемы простого голосования. Клиенты для определения классов используют глобальную модель, полученную с сервера.

Локальные классификаторы могут быть представлены любыми моделями. Глобальный классификатор является списком из локальных классификаторов.

Параметрами алгоритма являются: n, m – пороги изменения БД при достижении которых, локальные классификаторы обучаются заново на предоставленных объектах, k – число изменившихся классификаторов в списке глобального классификатора, при котором он заново рассылается всем клиентам. Значения n, m, k задаются перед работой алгоритма.

Алгоритм состоит из последовательно выполняемых шагов:

1. Инициализируются локальные классификаторы. На сервере инициализируется пустой список.

2. Локальные классификаторы обучаются на предоставленных данных. Параметры, описывающие локальные классификаторы, отправляются на сервер.

3. Сервер добавляет в список глобального классификатора локальные классификаторы, полученные от новых клиентов, и обновляет локальные классификаторы, полученные от клиентов, чьи классификаторы уже представлены в списке. Если глобальный классификатор достаточно изменился, то есть в сумме было добавлено или изменилось k новых

классификаторов в списке, то он заново отправляется всем клиентам, а k становится равным 0.

4. При появлении нового объекта или изменении параметров у старого объекта, клиент выбирает класс, основываясь на глобальной модели, полученной с сервера. Каждый классификатор из списка делает предсказание относительно объекта, все предсказания суммируются. Объекту присваивается класс, которому отдало предпочтение большинство классификаторов из списка.

5. Если база данных достаточно изменилась, то есть было добавлено n новых объектов или изменилось m старых, то локальный классификатор заново обучается и отправляется на сервер. Переход к шагу 3.

2. Экспериментальное исследование точности алгоритма

При тестировании параметры алгоритма n, m, k принимались за 1. В качестве метрики качества использовалась точность классификации (accuracy), то есть отношение числа правильно распознанных объектов к числу всех объектов. Во время обучения использовался оптимизатор Adam с шагом обучения 0.001 [10]. Функция потерь - перекрёстная энтропия между предполагаемыми и реальными классами объектов.

Локальные классификаторы представлены нейронными сетями, имеющими следующую структуру: свёрточный слой с 32 каналами на выходе и ядром размера 5x5, выделение максимума (max pooling) с ядром 2x2, свёрточный слой с 64 каналами на выходе и ядром размера 5x5, выделение максимума с ядром 2x2, полносвязанный слой из 512 нейронов, слой линейной активации и многомерная логистическая функция в конце. Одна сеть всего имеет 593342 параметров. Свёрточный слой уменьшает размер изображения, частично сохраняя информацию об изображении [11]. Слой выделения максимума уменьшает размер изображения, оставляя максимумы из небольших областей изображения [12]. Обучение сети производилось с помощью обратного распространения ошибки [13]. Похожие архитектуры были представлены в работах [1, 14] и показали хорошие результаты в распознавании изображений, поэтому была выбрана описанная архитектура.

Данные для тестирования были независимо и одинаково распределены (independent and identically-distributed — IID). Для тестирования используется библиотека MNIST [15]. MNIST database (Modified National Institute of Standards and Technology database) – открытая большая база данных рукописных цифр и букв. Содержит 60'000 тренировочных изображений и 10'000 тестовых. Изображения имеют размер 28x28 пикселей и чёрно-белый формат. В IID выборках для тестирования каждый объект имел одинаковый шанс попасть в выборку. Из-за ограниченного объёма тестовых данных есть предел на количество локальных классификаторов, но в экспериментах количество локальных классификаторов ниже, чем этот предел.

Для тестирования глобального классификатора была подготовлена отдельная выборка данных, не связанная с локальными тренировочными и локальными тестирующими выборками. Глобальный классификатор тестируется следующим образом: изначально список состоит только из незахваченных локальных классификаторов, обученных на реальных данных. Далее на каждом шаге в глобальном классификаторе один незахваченный локальный классификатор меняется на один захваченный, обученный на данных полученных от злоумышленника, и оценивается точность на глобальной тестовой выборке получившегося глобального классификатора.

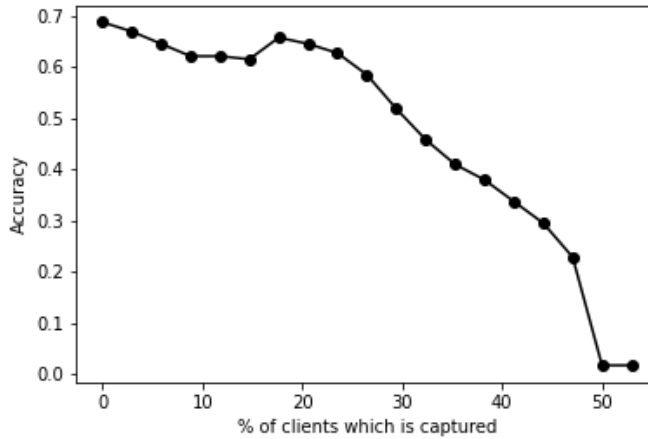


Рис. 1: Всего 34 клиента. Данные IID.

На графике по вертикальной оси отсчитывается точность, по горизонтальной – процент захваченных классификаторов от общего числа классификаторов в списке глобального классификатора.

Из рис. 1 видно, что при IID данных злоумышленнику необходимо захватить около четверти классификаторов, чтобы понизить точность глобального классификатора на 0.1, и половину, чтобы точность глобального классификатора стала почти нулевой. В данной ситуации классификаторы дают примерно похожие прогнозы для одинаковых данных из-за схожести обучающих выборок, соответственно, необходимо большое количество неверных предсказаний, чтобы взять большинство в простом голосовании и захватить глобальный классификатор.

Заключение

Проведённые эксперименты показали, что алгоритм устойчивее других известных алгоритмов [1,2,6,7,8] к атакам направленным на захват обучения локальных классификаторов или на передачу по сети параметров захваченных классификаторов вместо реальных параметров классификаторов. Чтобы получить контроль над глобальным классификатором, злоумышленнику необходимо захватить более половины, в случае IID данных, от общего числа локальных классификаторов и переобучить их на одинаково смещённых выборках.

Результаты экспериментов показывают, что предложенный алгоритм не показывает значительной прибавки точности глобального классификатора относительно локальных классификаторов. При приближении числа классификаторов с перехваченным обучением к числу обычных классификаторов алгоритм теряет точность из-за того, что преимущество в схеме голосования теряется.

Литература

1. H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, Blaise Agüera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data // Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, 2017, Vol. 54, pp. 1273-1282.

2. Jakub Konečný, H. Brendan McMahan, Daniel Ramage, Peter Richtárik. Federated Optimization: Distributed Machine Learning for On-Device Intelligence // CoRR, 2016, abs/1610.02527.

3. Q. Yang, Y. Liu, Y. Cheng, Y. Kang, T. Chen, and H. Yu. Federated learning // Synthesis Lectures on Artificial Intelligence and Machine Learning, 2019, Vol. 13, No. 3, pp. 1–207.

4. Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet. Advances and Open Problems in Federated Learning// Foundations and Trends in Machine Learning, 2021, Vol. 14, No. 1-2, pp. 1-210.

5. Federated Learning: Collaborative Machine Learning without Centralized Training Data, [Электронный ресурс], URL: `https://ai.googleblog.com/2017/04/federated-learningcollaborative.html`.

6. Felix Sattler, Klaus-Robert Müller, Wojciech Samek. Clustered Federated Learning: Model-Agnostic Distributed Multi-Task Optimization under Privacy Constraints // Computational Intelligence Magazine IEEE, 2021, Vol. 16, No. 1, pp. 49-62.

7. Qinbin Li, Zeyi Wen, Bingsheng He. Practical Federated Gradient Boosting Decision Trees // Proceedings of the AAAI Conference on Artificial Intelligence, 2020, Vol. 34, No. 04, pp. 4642-4649.

8. Yuqing Zhu, Xiang Yu, Yi-Hsuan Tsai, Francesco Pittaluga, Masoud Faraki, Manmohan chandraker, Yu-Xiang Wang. Voting-based Approaches For Differentially Private Federated Learning // arXiv preprint arXiv:2010.04851, 2021.

9. Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, Vitaly Shmatikov. How To Backdoor Federated Learning // Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics, 2020, Vol. 108, pp. 2938-2948.

10. Diederik P. Kingma, Jimmy Ba. Adam: A Method for Stochastic Optimization // CoRR, 2015, abs/1412.6980.

11. Convolutional neural network, [Электронный ресурс], URL: `https://en.wikipedia.org/wiki/Convolutional_neural_network#Convolutional_layers`.

12. Max-pooling layer, [Электронный ресурс], URL: `https://computersciencewiki.org/index.php/Max-pooling_/_Pooling`.

13. Уоссермен Ф. Нейрокомпьютерная техника // М.: Мир, 1992.

14. Y. Lecun, L. Bottou, Y. Bengio, P. Haffner. Gradient-based learning applied to document recognition // Proceedings of the IEEE, 1998, Vol. 86, No. 11, pp. 2278-2324.

15. The MNIST database of handwritten digits, [Электронный ресурс], URL: `http://yann.lecun.com/exdb/mnist/`.

Аннотации

Абрамов А. В., Чупахин А. А. Построение расписания выполнения работ в гетерогенной облачной вычислительной среде // Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

В данной работе рассматривается проблема планирования вычислений в гетерогенной вычислительной системе, в которой каждый вычислитель обладает определенным ресурсом, а для каждой поступающей в очередь планирования работы задана оценка времени выполнения и необходимый для ее выполнения ресурс.

Ил.: 6 рис., 3 табл. Библиогр.: 9.

Бодров А. О., Бахмуров А. Г. Построение масштабируемой платформы сбора медицинской телеметрии // Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

В работе решается задача построения платформы для сбора телеметрии, масштабируемой горизонтально при росте числа клиентов. Для достижения масштабируемости используется контейнерная виртуализация. А именно, в один контейнер упаковывается экземпляр брокера MQTT и ранее созданная серверная часть платформы, которая обеспечивает запись в БД. Для распределения клиентов по запущенным экземплярам серверной части использовано ПО HAProxy.

Ил.: 6 рис. Библиогр.: 10.

Галкина Е. В. Транспиляция программ на функциональных языках программирования: подходы и решения // Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

В работе рассмотрен один из путей совместного использования языков различных парадигм в рамках одного проекта —

транспиляция кода на дополнительном языке в код на базовом языке проекта. Описаны возможные подходы к транспиляции, выделены их преимущества и недостатки. Обоснована актуальность задачи транспиляции кода с языка Haskell в C++ и выбор подхода с сохранением структуры программы, описано подмножество языка Haskell, для которого реализуется транспиляция, приведено описание способа транспиляции для объявлений пользовательских типов данных языка Haskell.

Библиогр.: 18.

Кузьмин Я. К., Волканов Д. Ю. Разработка метода синхронизации состояния алгоритма обработки пакетов в сетевом процессорном устройстве // Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

В данной работе рассматривается сетевое процессорное устройство (СПУ) RuNPU, основанное на специализированных вычислительных ядрах, предназначенное для обработки пакетов в сетевых устройствах. В работе предлагаются модификации архитектуры СПУ, позволяющие хранить состояние алгоритма обработки пакетов и синхронизировать его между вычислительными конвейерами. Проводится экспериментальное исследование характеристик разработанной архитектуры.

Ил.: 5 рис. Библиогр.: 9.

Ларин А. В., Пашков В. Н. Применение нейронных сетей для прогнозирования нагрузки на контроллер в программно-конфигурируемых сетях // Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

В работе рассматривается проблема прогнозирования нагрузки на контроллер ПКС с целью предотвращения его перегрузки, как ключевого элемента управления ПКС сетью. Проводится краткий анализ применимости статистического подхода и методов машинного обучения для решения поставленной задачи. Предлагается метод прогнозирования нагрузки контроллера на основе нейронной сети. В рамках экспериментального исследования приводятся оптимальные параметры для обучения нейронной сети, приводится оценка точности прогноза разработанного метода.

Ил.: 1 рис., Библиогр.: 12.

Маркобородов А. А., Волканов Д. Ю. Трансляция групповой таблицы коммутатора ПКС в язык ассемблера сетевого

процессора RuNPU // Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

В работе рассматривается задача трансляции групповой таблицы протокола OpenFlow в программу на языке ассемблера программируемого сетевого процессора RuNPU. В работе предлагается три метода трансляции с промежуточным представлением групповой таблицы, которые позволяют разделить обработку пакета по группе действий на несколько проходов по вычислительному конвейеру. Для предложенных алгоритмов проводится экспериментальное исследование характеристик получаемой программы на языке ассемблера и времени обновления промежуточного представления групповой таблицы.

Ил.: 5 рис., Библиогр.: 6.

Никифоров Н. И., Волканов Д. Ю. Исследование применимости алгоритмов сжатия данных к таблицам классификации в сетевом процессорном устройстве // Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

В данной работе рассматривается задача представления таблиц потоков в рамках архитектуры сетевого процессорного устройства (СПУ). СПУ представляет из себя специализированную интегральную микросхему. В СПУ используется конвейерная архитектура, каждый конвейер состоит из восьми вычислительных блоков. Каждый вычислительный блок имеет доступ к устройству памяти объёмом 64K, в котором хранится программа обработки сетевых пакетов. Для идентификации сетевого пакета по его заголовку и поиска действий, которые СПУ должен выполнить над сетевым пакетом, требуются таблицы потоков, которые могут содержать до десятков тысяч правил. Поэтому занимаемый ими объём памяти может достигать до десятков мегабайт. Таблицы потоков представляются в виде программы обработки пакетов на языке ассемблера. Таким образом возникает задача разработки алгоритмов сжатия для применения к таблицам потоков. Экспериментальное исследование разработанных алгоритмов сжатия было проведено на имитационной модели сетевого процессора.

Ил.: 3 рис., Библиогр.: 12.

Пантюхин Л. К., Антоненко В. А. Разработка метода борьбы с атаками типа отказ в обслуживании при L4 балансировке

// Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

Современные пользовательские приложения представляют из себя набор микросервисов, которые разворачиваются на множестве серверов. Для балансировки клиентских запросов между этими серверами в реальном времени необходимы балансировщики нагрузки. Однако L4 балансировщики с сохранением состояния (stateful) могут быть уязвимы к DoS атакам, в частности к атаке SYN-flood. Цель данной работы реализовать прототип балансировщика, способного эффективно противостоять такой атаке. В данной работе проведён обзор основных методов борьбы с SYN-flood атакой в контексте балансировщика нагрузки. Предложен метод, основанный на методе SYN-cookies. Проведена реализация предложенного метода с использованием языка Python и модификации ядра Linux, а также собран экспериментальный стенд с использованием VirtualBox и Mininet. Проведено экспериментальное исследование с целью демонстрации работоспособности предложенного метода.

Ил.: 6 рис., 1 табл. Библиогр.: 18.

Рябченков В. М., Глонина А. Б. Разработка аналитического метода оценки времени отклика задач в системах интегрированной модульной авионики // Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

В данной работе предложен аналитический метод оценки времени отклика задач в системах интегрированной модульной авионики в случае интервальной неопределенности длительнности выполнения работ и длительности передачи сообщений. Доказана корректность предложенного алгоритма. Проведено экспериментальное исследование предложенного метода на искусственно сгенерированных и реальных данных с целью сравнения его с другим существующим методом по точности между собой, а также с целью сравнения получаемых предложенным алгоритмом верхних оценок с гарантированно достижимыми нижними оценками. Экспериментальное исследование показало высокую точность предложенного алгоритма.

Ил.: 1 рис., 2 табл. Библиогр.: 8.

Селезнев Л. Е., Костенко В. А. Влияние способа задания начального приближения на эффективность обучения нейронных сетей // Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

В работе приводятся обзор и результаты сравнения эффективности применения различных алгоритмов инициализации начального приближения в том числе для сетей с различными функциями активации.

Ил.: 1 таб., Библиогр.: 13.

Синицын А. А., Степанов Е. П., Писковский В. О. Высокоуровневая система разработки приложений для программно-конфигурируемых сетей под управлением контроллера RunOS v2.0 // Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

В данной статье рассмотрен опыт реализации приложения Maple для программно-конфигурируемых сетей под управлением контроллера RunOS v2.0.

Ил.: 4 рис., 3 лист. Библиогр.: 11.

Танкаев И. Р., Костенко В. А. Федеративное обучение с использованием схемы простого голосования // Программные системы и инструменты. Тематический сборник № 21, М.: Изд-во факультета ВМК МГУ, 2021.

В работе рассматривается задача федеративного обучения классификаторов. Предложен алгоритм обучения классификаторов с использованием схемы простого голосование (комитета большинства). Приведены результаты исследования свойств алгоритма: точность и устойчивость к перехвату обучения.

Ил.: 1 рис. Библиогр.: 15.

Software systems and tools: Tematic collection / Ed. by R. L. Smelyansky.— Moscow: Publishing Department of the Faculty of Computational Mathematics and Cybernetics (license ID № 05899 from 24.09.2001); MAKS Press, 2021.— № 21.— 164 p.

ISBN 978-5-89407-625-5 (CMC MSU)

ISBN 978-5-317-06733-5 (MAKS Press)

Theese proceedings consists of student's works, who have received the recommendation of the departments Automation of Computer Systems and Algorithmic Languages. This edition continues the tradition of publishing in memory of Korolev L.N.. The proceedings contains articles devoted to problems of modern computer networks, methods and tools for organizing and managing cloud computing, tools for medical telemetry, network processor unit architecture, training and use of neural networks and program transpilation.

These papers will be of interest to students, graduate students and professionals in the development of application software systems.

Keywords: information telecommunication technology, software-defined networking, task scheduling, cloud computing, cloud environment, genetic algorithms, evolutionary algorithms, platform for medical research, Internet of Things, virtual resources, network functions virtualization, cloud platform, program transpilation, functional programming languages, network processor unit, state synchronization, neural networks, OpenFlow Controller, load prediction, classification tables, data compression, Denial of Service, load balancing, Response Time Estimation, Integrated Modular Avionics, federated learning, Neural Networks, simple vote, classifier.

Научное издание

ПРОГРАММНЫЕ СИСТЕМЫ И ИНСТРУМЕНТЫ

Тематический сборник

№ 21

Под общей редакцией чл.- корр. РАН,
профессора Р. Л. Смелянского

Издательский отдел
Факультета вычислительной математики и кибернетики МГУ
имени М.В. Ломоносова
Лицензия ИД N 05899 от 24.09.01 г.

119992, ГСП-2, Москва, Ленинские горы,
МГУ имени М.В. Ломоносова,
2-й учебный корпус

Издательство «МАКС Пресс»
Главный редактор: *Е. М. Бугачева*

Напечатано с готового оригинал-макета
Подписано в печать 22.12.2021 г.
Формат 60x90 1/16. Усл.печ.л. 10,25.
Тираж 50 экз. Заказ 194.

Издательство ООО “МАКС Пресс”
Лицензия ИД N 00510 от 01.12.99 г.
119992, ГСП-2, Москва, Ленинские горы,
МГУ им. М. В. Ломоносова,
2-й учебный корпус, 527 к.
Тел. 8(495)939–3890/91. Тел./Факс 8(495)939–3891.

Отпечатано в полном соответствии с качеством
предоставленных материалов в ООО «Фотоэксперт»
109316, г. Москва, Волгоградский проспект, д. 42,
корп. 5, эт. 1, пом. I, ком. 6.3-23Н