Анализ и исследование структур данных для поиска в таблицах классификации в архитектуре сетевого процессора без выделенного ассоциативного устройства

***Аннотация*—В данной работе рассматривается проблема классификации пакетов в рамках архитектуры сетевого процессора, без выделенного ассоциативного устройства. Под классификацией понимается процесс идентификации пакета по его заголовку. Для этапа классификации требуется реализация структур данных для хранения таблиц классификации. В рассматриваемом сетевом процессоре присутствуют некоторые ограничения, из-за которых невозвожно использование TCAM памяти. На основе ограничений рассматриваемой архитектуры были выбраны структуры данных для дальнейших исследований. На основе выбранных структур данных были реализованы адаптированные под рассматриваемую архитектуру сетевого процессора деревья. Экспериментальное исследование, реализованных структур данных, было проведено на имитационной модели сетевого процессора, Использование адаптированного АВЛ дерева позволило сократить количество тактов сетевого процессора на обработку одного пакета и уменьшить использование памяти вычислительных блоков конвейера.**

***Index Terms*—**

I. Введение

В настоящее время активно развивается технология программно-конфигурируемых сетей [10], в которых требуются высокопроизводительные коммутаторы [4]. Возникает задача разработки программируемого сетевого процессора, являющегося основным функциональным элементом коммутаторов. Сетевой процессор представляет из себя интегральную микросхему, специализированную для обработки сетевых пакетов, которая выполняет следующие функции: получение пакета с физического порта, выделение заголовка, классификация пакета по его заголовку, принятие решения о дальнейшем пути следования пакета, отправка пакета на физический порт [6]. В настоящее время активно ведётся разработка программируемых сетевых процессоров. Под программируемым сетевым процессором понимается сетевой процессор, который позволяет менять программу обработки пакетов и набор различаемых полей заголовков, что позволяет быстро подстраиваться под новые протоколы, и использовать коммутатор в ПКС сетях [9].

Исходя из функций сетевого процессора, целесообразно рассматривать архитектуру, основанную на наборе конвейеров, которая позволяет с фиксированной задержкой обрабатывать каждый пакет. Конвейер в сетевом процессоре состоит из вычислительных блоков. В данной работе рассматривался этап классификации пакетов. Под классификацией понимается процесс идентификации сетевого пакета по его признакам, определяемыми текущим протоколом. Таблица классификации – набор правил, содержащих в себе признаки, по которым идентифицируется группа пакетов, и действия, которые сетевой процессор выполняет над данной группой пакетов. Таким образом, для выполнения классификации сетевой процессор должен включать в себя ассоциативное устройство. Для реализации этого устройства естественным будет использование ассоциативной памяти. Однако единственный контроллер ассоциативной памяти будет являться узким местом, так как к нему должны иметь доступ все стадии всех конвейера. Соответственно, возникает потребность усложнения архитектуры, например, путём добавления в неё нескольких контроллеров ассоциативной памяти. Чтобы избежать сложной организации памяти, в архитектуре можно отказаться от использования ассоциативной памяти. В таком случае одно из решений – совместить память команд и данных, и разместить память на кристалле сетевого процессора. Таким образом, возникает задача разработки структур данных для поиска в таблицах классификации в сетевом процессоре без выделенного ассоциативного устройства.

II. Архитектура сетевого процессора

В рассматриваемом сетевом процессоре используется конвейерная архитектура, каждый конвейер состоит из 10 вычислительных блоков.

Каждый вычислительный блок имеет доступ к участку памяти, в котором располагаются микрокод и данные. Существует ограничение на количество тактов, которое один пакет может обрабатываться на вычислительном блоке, оно соответствует 25 тактам. Данное ограничение обусловлено требованием к производительности сетевого процессора, а именно требованием фиксированного времени обработки одного

пакета на сетевом процессоре. Также один вычислительный блок имеет доступ к 2 мегабайтам памяти. Из-за особенностей микроархитектуры, отсутствует отдельная область памяти, в которой хранятся данные. Поэтому микрокод содержит в себе все данные, необходимые для классификации пакетов.

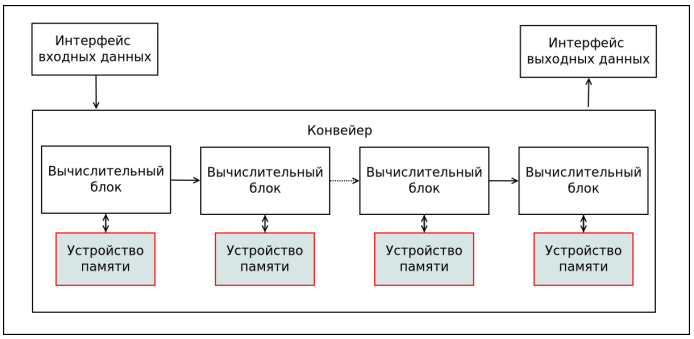

Рис. 1: Архитектура рассматриваемого сетевого процессора

A. Язык ассемблера сетевого процессора

Для описания программ обработки сетевых пакетов в рассматриваемой архитектуре сетевого процессора используется язык ассемблера. В рассматриваемом языке присутствуют следующие классы инструкций:

- Инструкции для работы с регистром.
- Инструкции арифметических операций.
- Инструкции битовых операций.
- Инструкции для условного и безусловного перехода на метку.
- Инструкция записи в регистр выходного порта.

III. Проделанная работа

В литературе представлены решения поставленой задачи. В силу особенностей архитектуры сетевого процессора, а именно отсутствия адресуемой памяти, которая требуется для не древовидных структур данных будут рассматриваться только древовидные структуры данных.

В рассматриваемых решениях будут рассматриваться следующие аспекты: **асимптотическая сложность поиска** – позволяет оценить использование ресурсов сетевого процессора для поиска в структуре данных, **универсальность структуры данных** – используемая структура данных должна поддерживать поиск произвольных битовых строк, не превосходящих по длине 128 бит, **необходимость использования адресуемой памяти** – некоторым рассматриваемым структурам данных требуется адресуемая память для их реализации, соответствующей их асимптотической сложности, **Количество вершин**, которое необходимо посетить для поиска в случае хранения битовых строк длиной до 48 бит, **Необходимость изменения микроархитектуры** вычислительных блоков конвейера сетевого процессора, **Оценка объёма памяти**, занимаемой структурой данных – рассматривается объём памяти, занимаемый структурой данных, для 64000 вхождений префиксов IPv4.

A. Метод подсчёта объёма памяти занимаемый рассматриваемой структурой данных

Так как в рассматриваемой архитектуре сетевого процессора совмещена память для команд и данных, объём памяти занимаемой структурой данных для конкретного набора битовых строк соответствует объёму памяти, занимаемому программой, которая реализует рассматриваемую структуру данных.

Для реализуемой структуры данных вычисляется количество вершин для хранения 64000 битовых строк – N, затем вычисляется среднее количество инструкций на одну вершину – S. Тогда искомый объём памяти занимаемый структурой данных будет рассчитываться по формуле $M = instruction\_size * N * S$, где $instruction\_size = 128$, в силу рассматриваемой архитектуры сетевого процессора.

B. бинарное однобитное дерево

Наиболее распространенная [2] структура данных для поиска по наибольшему совпадению. Строится дерево по заданным префиксам так, что каждому биту префикса соответствует своя вершина в дереве. Поиск осуществляется спуском в глубину по битам элемента, для которого выполняется поиск. Поиск заканчивается тогда, когда достигнута пустая вершина, результатом поиска считается последний встретившийся префикс [6].

C. Бинарное сжатое дерево

Рассматриваемая структура данных – оптимизация бинарного однобитного дерева [8]. Для построения данного дерева необходимо построить бинарное однобитное дерево, затем провести процедуру сжатия, а именно все вершины, у которых только один лист, сокращаются, и в следующую вершину заносятся данные о количестве пропущенных вершин. Таким образом, построенное дерево не имеет вершин с одним листом. Благодаря описанной оптимизации, данное дерево занимает меньше памяти, чем бинарное однобитное дерево. Это обусловлено отсутствием проходных вершин. Однако, уменьшается количество затраченных инструкций на поиск лишь для префиксов, перед которыми были однолистные вершины. Для худшего случая, когда для префикса есть все его более короткие версии, количество вершин, которые нужно посетить для поиска, аналогично бинарному однобитному дереву [8].

D. Мультибитное сжатое дерево

Данное дерево – оптимизация бинарного сжатого дерева [3]. Используется другая структура деревьев, когда в каждой вершине может быть максимум не два

листа, а 2^h, где h — это максимальная глубина поддерева данной вершины. При использовании рассматриваемой структуры данных в рамках архитектуры процессора общего назначение, количество операций для поиска ограничивается глубиной дерева, которая равна $\frac{W}{K}$, где W — длина максимального префикса, а K — количество уровней в нашем дереве. В рамках архитектуры сетевого процессора будет рассматриваться реализация данного дерева, в которой используется линейный поиск в каждой вершине по дочерним вершинам [3].

E. Мультибитное сжатое дерево с изменением микроархитектуры

При реализации микроинструкции, которая по заданной метке перехода и маске переходит на инструкцию находящуюся на метке + значение регистра с применённой маской, есть возможность реализовать структуру данных, в которой алгоритм поиска будет быстрее. Рассмотрим эту структуру данных в рамках архитектуры с добавлением указанной микроинструкции.

F. Бинарный поиск по длинам префиксов

Структура данных основана на построении специальных таблиц для префиксов определённой длины [7]. Пусть максимальная длина префикса W, тогда строятся таблицы $h_1, \ldots, h_w$. В каждой из них хранятся префиксы длины соответствующие номеру этой таблицы. Предполагается, что в каждой такой таблице реализована своя хеш-функция, которая быстро позволяет найти вхождение префикса в данную таблицу. Таким образом мы можем выполнить бинарный поиск по длине префиксов. В рамках рассматриваемой архитектуры сетевого процессора, реализация таких таблиц возможна только с использованием древовидных структур данных [7].

G. АВЛ дерево

Представление префиксов как скалярных префиксов позволяет использовать больший набор структур данных [2]. В качестве примера рассмотрим АВЛ дерево, основной особенностью которого является правило его построения: у каждой вершины разность глубины левого и правого поддерева не превосходит 1, что даёт асимптотическую сложность поиска $O(1 + \log_2 N)$, где N — количество префиксов в нашей структуре данных. Из этого следует, что время поиска не зависит от длины искомых данных, а значит с помощью данной структуры данных эффективно выполнять поиск префиксов IPv6 [1].

H. Сравнение структур данных

У каждой рассмотренной структуры данных есть свои достоинства и недостатки, рассмотрим их:

1) **Бинарное однобитное дерево** — данная структура проста в реализации, но занимает много памяти и поиск требует прохождения 48 вершин.
2) **Бинарное сжатое дерево** — занимает меньше памяти, чем двоичное однобитное дерево, но поиск требует прохождения 48 вершин. Соответственно использование данного дерева предпочтительнее, чем двоичного однобитного дерева.
3) **Мультибитное сжатое дерево** — занимает много памяти, но поиск требует прохождения сильно меньшего количества вершин. Из-за проблем с реализацией в рамках рассматриваемой архитектуры, эта структура данных не может быть реализована.
4) **Бинарный поиск по длинам префиксов** — занимает много памяти, и может быть использован только для поиска наиболее длинного префикса. Также из-за проблем с реализацией на рассматриваемой архитектуре, данная структура не подходит для решения проблемы.
5) **АВЛ дерево** — занимает меньше всего памяти, и при этом, поиск требует прохождения малого количества вершин, которое зависит от количества вхождений в структуру данных, а не от конкретного префикса.

Таким образом на основе обзора для дальнейшей реализации целесообразно выбрать две структуры данных: АВЛ дерево с алгоритмом представления префиксов как скалярных величин и бинарное сжатое дерево. Также в качестве эталона выбрано бинарное однобитное дерево.

I. Выводы

В данном разделе были рассмотрены следующие структуры данных: двоичное однобитное дерево, двоичное сжатое дерево, мультибитное сжатое дерево, бинарный поиск по длинам префиксов и АВЛ дерево. Обзор проводился с целью выбора структур данных для дальнейшей адаптации и усовершенствования в рамках рассматриваемой архитектуры сетевого процессора, и дальнейшей реализации полученных структур данных. По итогам проведённого обзора были выбраны следующие структуры данных: АВЛ дерево и бинарное сжатое дерево.

IV. Разработка структур данных

В данной главе представлены этапы разработки выбранных в рамках обзора структуры данных для рассматриваемой архитектуры сетевого процессора. А также описаны адаптации предложенные для реализации выбранных структур данных. Данные адаптации обусловлены ограничениями описанными в главе **??**.

В описании алгоритмов использовалась структура данных *Node*, которая содержит в себе следующие поля:

- *Node.left* — ссылка на левого сына вершины.
- *Node.right* — ссылка на правого сына вершины.
- *Node.rule* — действие, соответствующее префиксу данной вершины.
- *Node.prefix* — префикс в текущей вершине.
- *Node.bit* — значащий бит в текущей вершине.

И структура *prefix*, которая содержит в себе следующие поля:

- *prefix.bit_string* — битовая строка, задающая префикс.
- *prefix.length* — длина префикса. При поиске точного совпадения длина префикса равна длине битовой строки.

A. Бинарное однобитное дерево

Рассмотрим алгоритм построения бинарного однобитного дерева, являющийся базовым для бинарнго сжатого дерева. Для добавления вершин в структуру данных определим процедуру *Add* (Листинг 1).

Входные данные: текущая вершина *Node*, добавляемый префикс *prefix*, правило для текущего префикса *rule*, и текущий бит *bit*.

Выходные данные: обновлённая текущая вершина, а именно объект структуры *Node*.

```
1  procedure Add(Node, prefix, rule, bit)
2      if Node is empty then
3          Node = new Node()
4          Node.bit = bit
5      endif
6      if prefix.length == bit then
7          Node.prefix = prefix
8          Node.rule = rule
9          Node.bit = bit
10         return Node
11     endif
12     if prefix.bit_string[bit] == 1 then
13         Node.left = Add(Node.left, prefix, rule,
       bit + 1)
14     else
15         Node.right = Add(Node.right, prefix, rule
       , bit + 1)
16     endif
17     return Node
```

Листинг 1: Процедура добавления вершины в бинарное однобитное дерево.

Таким образом, для построения бинарного однобитного дерева необходимо последовательно добавить все правила из таблицы классификации, используя процедуру *Add*.

B. Бинарное сжатое дерево

Рассмотрим алгоритм построения бинарного сжатого дерева. Для упрощения программной реализации сначала строилось бинарное однобитное дерево, процедура построения которого описана в пункте IV-A. Для получения из построенного бинарного однобитного дерева бинарного сжатого дерева, необходимо удалить все вершины, которые имеют только одного сына, и не имеют префикса.

1) Процедура удаления незначимых вершин: Рассмотрим процедуру удаления *Remove* незначимых вершин из бинарного однобитного дерева, для получения бинарного сжатого дерева.

Входные данные: текущая вершина *Node*.

Выходные данные: обновлённая текущая вершина *Node*. Алгоритм удаления не важных вершин может быть представлен в виде псевдокода процедуры *Remove* (Листинг 2)

```
1  procedure Remove(Node):
2      if Node.prefix is empty then
3          if Node.left is empty and Node.right is
       not empty then
4              return Node.right
5          endif
6          if Node.left is not empty and Node.right
       is empty then
7              return Node.left
8          endif
9      else
10         return Node
11     endif
```

Листинг 2: Процедура удаления незначимых вершин.

Данный алгоритм позволяет построить бинарное сжатое дерево, на базе бинарного однобитного дерева, путём удаления незначимых вершин из бинарного однобитного дерева.

Для данного дерева необходимо было разработать процедуру описывающую вершину рассматриваемой структуры данных. Данная процедура должна позволять выполнять поиск по построенной структуре данных. Рассмотрим процедуру описывающую вершину двоичного сжатого дерева в рамках рассматриваемой архитектуры сетевого процессора:

```
1  setmask (1 << port)
2  cmpj lable, prefix.bit_string, prefix.length
```

Листинг 3: Процедура описывающая вершину бинарном сжатом дереве.

В данной процедуре: *setmask 1 « port* — является опциональной инструкцией и присутствует только тогда, когда в текущей вершине содержится префикс. *cmpj label, prefix.bit_string, prefix.length* — сравнение в текущей вершине, то происходит переход на метку *label*, иначе просто продолжается выполнение программы. Метка *label* — может быть перед левым сыном, если он есть иначе метка *finish*, которая завершает программу и тем самым прекращает поиск по структуре данных.

C. АВЛ дерево

1) Алгоритм представления префиксов как скалярных величин: Данный алгоритм необходим для сохранения и поиска префиксов в АВЛ дерева, так как изначально в вершинах дерева могут хранится только скалярные величины. Алгоритм позволяет сравнивать префиксы между собой [1], а именно:

- Если длины двух префиксов совпадают, то они сравниваются как две скалярные величины.
- Если длина префикса q меньше длины префикса p, то как скалярные величины сравниваются первые $len(q) - 1$ бит у обоих префиксов.

Данный алгоритм может быть описан процедурой *PrefixCompare* (Листинг 4).
Входные данные: префикс *prefix1* и префикс *prefix2*.
Выходные данные: *True*, если первый префикс больше второго, иначе *False*.

```
1  procedure PrefixCompare(prefix1, prefix2):
2      if prefix1.length == prefix2.length then
3          return Int(prefix1.bit_string) > Int(
       prefix2.bit_string)
4      else
5          minimal_length = Min(prefix1.length,
       prefix2.length) - 1
6          return Int(prefix1.bit_string[0:
       minimal_length]) > Int(prefix2.bit_string[0:
       minimal_length])
7      endif
```

Листинг 4: Процедура сравнения префиксов как скалярных величин.

2) Алгоритм построения дерева: Рассмотрим алгоритм построения АВЛ дерева, а именно добавления в него новых вершин. Данный алгоритм может быть описан процедурой *Add* (Листинг 5). Для построения АВЛ дерева используется структура *Node*, которая идентична структуре описанной в пункте IV-A.
Входные данные:

- *Node* — объект структуры вершины АВЛ дерева.
- *prefix* — текущий префикс.
- *rule* — правило для текущего префикса.

Выходные данные: обновлённая текущая вершина *Node*.
Также в процедуре *Add* используется процедура *BalanceNode*, которая выполняет операцию балансировки АВЛ дерева.

```
1  procedure Add(Node, prefix, rule):
2      if Node.prefix is empty then
3          return new Node(prefix, rule, depth=1)
4      endif
5
6      if prefixCompare(prefix, Node.prefix) then
7          Node.right = Add(Node.right, prefix, rule
       )
8      else:
9          Node.left = Add(Node.left, prefix, rule)
10     endif
11
12     return BalanceNode(Node)
```

Листинг 5: Процедура добавления вершины в АВЛ дерево.

Данная процедура позволяет построить АВЛ дерево, последовательно добавляя правила из таблицы.

V. Описание реализованных программных средств

A. Имитационная модель сетевого процессора

Имитационная модель сетевого процессора — программное средство написанное на языке программирования python, которая позволяет имитировать работу сетевого процессора, а именно получать пакеты на любой из 24-х портов, обрабатывать пакет, получая информацию о любой стадии обработки, и отправлять пакет на выходные порты. Имитационная модель сетевого процессора позволяет оценить количество тактов затраченных на обработку пакета, а также объём памяти, затраченный на структуру данных.

Передача пакетов на порты сетевого процессора осуществляется с помощью файлов *pcap*, в которых можно описать поступление пакетов на каждый порт в определённые моменты времени. В данной работе, не ограничивая общности, будет рассматриваться только один порт имитационной модели сетевого процессора. Так как для каждого входного порта используется соответствующий конвейер. В конвейере имитационной модели сетевого процессора реализован один вычислительный блок, этого достаточно для проведения экспериментального исследование, так как конвейер сетевого процессора имеет статическое расписание.

Исходная программа для обработки пакетов принимается в виде кода, написанного на языке ассемблера, который используется в рассматриваемой архитектуре сетевого процессора. Имитационная модель преобразует программу, раскрывая директивы *#include* и *#define*, и начинает имитацию работы сетевого процессора по этой программе.

B. Описание программной реализации

Для проведения дальнейшего экспериментального исследования необходимо разработать программную реализацию на языке python. Программная реализация должна включать в себя следующие модули:

- Модуль загрузки таблиц классификации из файла.
- Модуль построения АВЛ-дерева.
- Модуль построения Бинарного сжатого дерева.
- Модуль построения Бинарного однобитного дерева.
- Модуль преобразования построенного дерева в язык ассемблера.

Схема взаимодействия модулей представлена на рисунке 2.

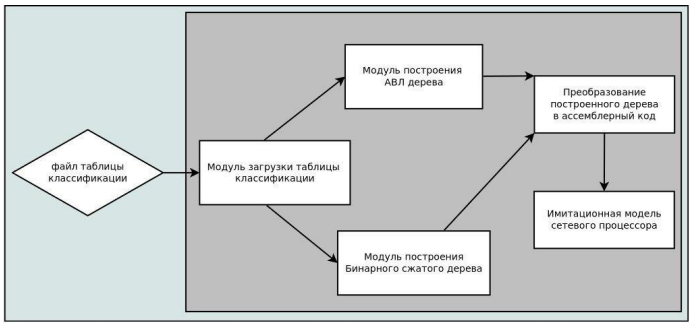

Рис. 2: Схема программной реализации

1) Модуль загрузки таблиц классификации: В данном модуле реализуется считывание файлов таблиц классификации и преобразование их в словарь где ключом является битовая строка, по которой будет происходить поиск, а значением номер порта коммутатора. Данные словари отправляются в модули построения структур данных. Модуль позволяет использовать существующие таблицы классификации для проведения экспериментального исследования. Пример входных таблиц классификации представлен в листинге 6

```
{IPv4}
192.168.31.0 24 1
192.168.43.0 24 2
```

Листинг 6: Пример таблиц классификации

2) Модули построения структур данных: В данных модулях реализуется построение описанных деревьев, а именно, бинарного сжатого дерева, бинарного однобитного дерева и АВЛ дерева. На вход модули принимают считанные из файла и преобразованные таблицы классификации. Затем каждое правило последовательно добавляется в структуру данных, алгоритм добавления описан в главе IV, для каждой из рассматриваемых структур данных. После построения структура данных передаётся в модуль преобразования дерева в язык ассемблера использующийся в сетевом процессоре в виде корневой вершины.

3) Модуль построения преобразования построенного дерева на язык ассемблера: Для каждого дерева модуль преобразования строит определённый ассемблерный код, подробно описанный в главе IV. Преобразование происходит рекурсивно проходя древовидную структуру данных, вершины последовательно записываются в массив, и затем для каждой вершины генерируется собственный ассемблерный код. Сгенерированный ассемблерный код записывается в указанный файл. Затем файл используется при экспериментальном исследовании на имитационной модели сетевого процессора. Пример преобразования бинарного сжатого дерева из пункта III-C представлен в листинге 7.

```
cmpj l_1, 1, 1
setmask (1 << 3)
cmpjn finish, 1, 2
setmask (1 << 1)
j finish

l_1:
cmpj l_2, 11, 2
cmpjn finish, 101, 3
setmask (1 << 5)
cmpjn finish, 1011, 4
setmask (1 << 6)
j finish

l_2:
setmask (1 << 4)
cmpjn finish, 1100, 4
setmask (1 << 2)
j finish

finish:
```

Листинг 7: Пример бинарного сжатого дерева на языке ассемблера.

C. Вспомогательные программные средства

Для проведения экспериментального исследования в рамках данной работы, был разработан генератор сетевого трафика, который реализует следующую функциональность:

- Возможность генерации L2 и L3 трафика, а именно трафика использующего, например, протоколы ARP, и IPv4.
- Возможность использовать заданную базу префиксов или MAC адресов для генерации трафика.
- Возможность использовать для генерации трафика заданное количество случайных префиксов или MAC адресов.
- Возможность сохранить базу использованных префиксов или MAC адресов.
- Возможность задать распределения времени поступления пакетов.
- Возможность сохранять сгенерированный трафик в *pcap* файлы.

Для реализации используется язык программирования *python*, так как существует открытая библиотека *scapy* [5], которая позволяет работать с L2 и L3 трафиком, а так же позволяет сохранять сгенерированный трафик в *pcap* файлы, что необходимо, для дальнейшего использования в имитационной модели сетевого процессора.

VI. Методика экспериментального исследования

Для проведения экспериментального исследования были найдены таблицы коммутации префиксов различной длины. Экспериментальное исследование будет проводится для таблиц классификации различного размера, а именно: 1000, 8000, 16000, 32000, 64000

битовых строк. Будут использоваться битовые строки различной длины, а именно: 32, 48 бита. Для каждой структуры данных будут проведены исследования с трафиком покрывающим загруженную таблицу в имитационную модель сетевого процессора. Также будет проводится исследование с поиском наиболее длинного префикса и точного совпадения.

Для проведения экспериментального исследования необходимо выполнить следующие действия:

1) Подготовить входные данные для программной реализации, а именно привести таблицы классификации к виду описанному в пункте V-B1.
2) Для каждой из рассматриваемых таблиц классификации построить разработанные структуры данных и преобразовать их в ассемблерный код.
3) Провести имитацию для каждой структуры данных на имитационной модели сетевого процессора.
4) Зафиксировать полученные данные при имитации, а именно объём затраченной памяти конвейера сетевого процессора, и среднее количество инструкций на классификацию пакета.

Для бинарного сжатого дерева ожидается получить значительное уменьшение объёма памяти и уменьшение среднего количества тактов на один пакет по сравнению с бинарным однобитным деревом. Для АВЛ дерева ожидается уменьшение среднего количества тактов затраченных на поиск одного пакета по сравнению с бинарным сжатым деревом и не сильное уменьшение затраченного объёма памяти на хранение структуры данных.

VII. Результаты экспериментального исследования

В данной главе представлены результаты экспериментального исследования разработанных структур данных для таблиц классификации размера от 1000 до 64000, и признаками различной длины.

A. Бинарное однобитное дерево

Рассмотрим график зависимости среднего количества инструкций, потраченных на поиск по структуре данных, для пакета от количества правил в таблице классификации (Рис. 3). Как видно из представленного графика, среднее количество затраченных инструкций практически не изменяется в зависимости от размера таблицы классификации. В среднем количество составляет 48 инструкций на пакет для битовых строк длины 32 бита, и в среднем 41 инструкцию для битовых строк длины 48 бит. Теперь рассмотрим график зависимости затраченного объёма памяти от количества правил в таблице классификации (Рис. 4). Из представленного графика видно, что бинарное однобитное дерево занимает в несколько раз больше памяти, чем в представленных в главе **??** ограничениях. Что не позволяет использовать использовать данную структуру данных в рассматриваемой архитектуре сетевого процессора.

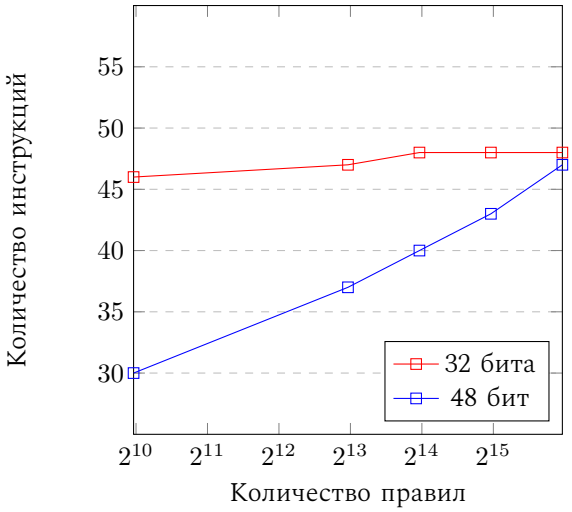

Рис. 3: Зависимость затраченных инструкций от количества правил в таблице классификации.

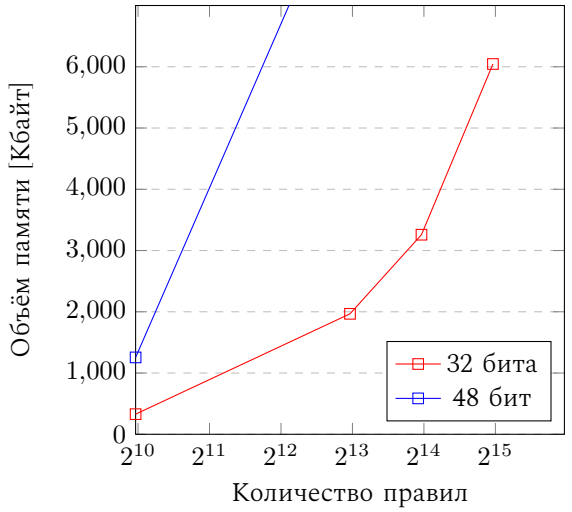

Рис. 4: Зависимость затраченного объёма памяти от количества правил в таблице классификации.

B. Бинарное сжатое дерево

Рассмотрим график зависимости среднего количества инструкций, потраченных на поиск по структуре данных, для пакета от количества правил в таблице классификации (Рис. 5). Как видно из представленного графика, среднее количество затраченных инструкций практически не изменяется в зависимости от размера таблицы классификации. В среднем количество составляет 39 инструкций на пакет для битовых строк длины 32 бита, и в среднем 31 инструкцию для битовых строк длины 48 бит. Теперь рассмотрим график зависимости затраченного объёма памяти от количества правил в таблице классификации (Рис. 6). Из

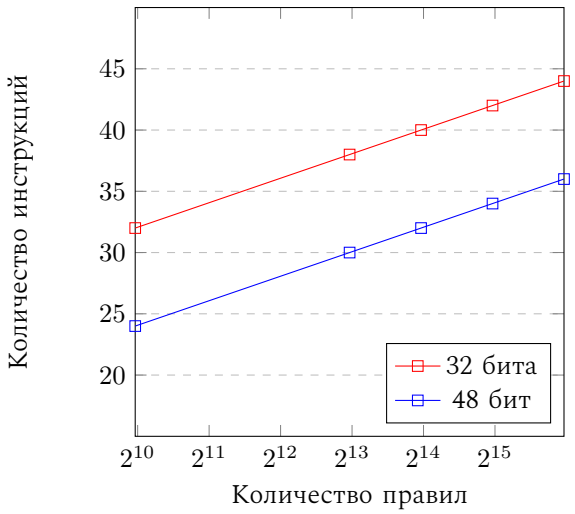

Рис. 5: Зависимость затраченных инструкций от количества правил в таблице классификации.

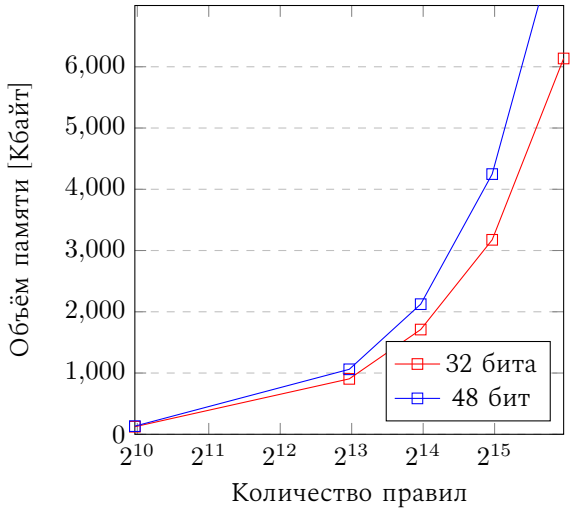

Рис. 6: Зависимость затраченного объёма памяти от количества правил в таблице классификации.

представленного графика видно, что бинарное сжатое дерево занимает больше памяти, чем в представленных в главе **??** ограничениях, только при 32000 правил в таблицах классификации.

C. АВЛ дерево

Рассмотрим график зависимости среднего количества инструкций, потраченных на поиск по структуре данных, для пакета от количества правил в таблице классификации (Рис. 7).

Как видно из представленного графика, среднее количество затраченных инструкций практически не изменяется в зависимости от размера таблицы классификации. В среднем количество составляет 30 инструкций на пакет для битовых строк длины 32 бита, и в среднем 23 инструкцию для битовых строк длины 48 бит.

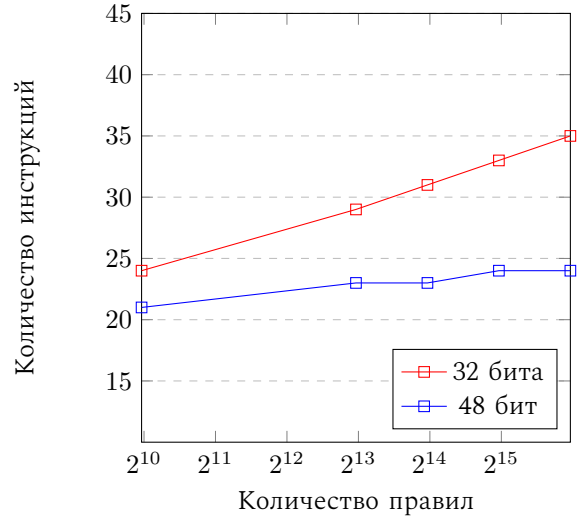

Рис. 7: Зависимость затраченных инструкций от количества правил в таблице классификации.

Теперь рассмотрим график зависимости затраченного объёма памяти от количества правил в таблице классификации (Рис. 8).

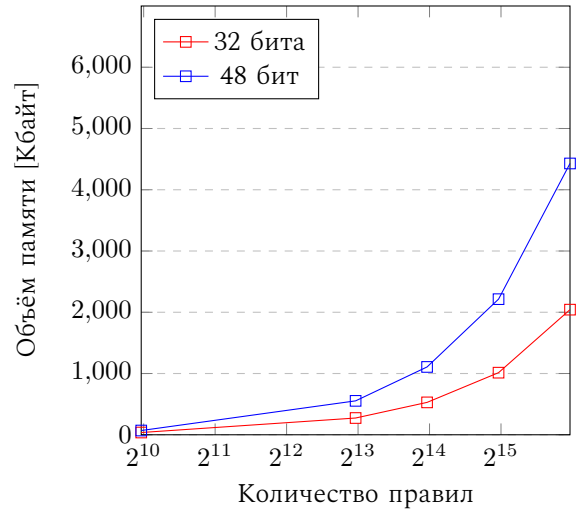

Рис. 8: Зависимость затраченного объёма памяти от количества правил в таблице классификации.

Из представленного графика видно, что АВЛ дерево укладывается в представленные в главе **??** ограничения, за исключением случая, когда в таблице классификации 64000 правила, с признаками длиной 48 бит.

D. Сравнение структур данных

В данном пункте представлено сравнение реализованных структур данных. Рассмотрим сравнительные графики всех реализованных структур данных. На первом графике (Рис. 9) представлено сравнение реализованных структур данных по

среднему количеству инструкций на один пакет. Видно, что АВЛ дерево показывает лучший результат из всех реализованных структур данных.

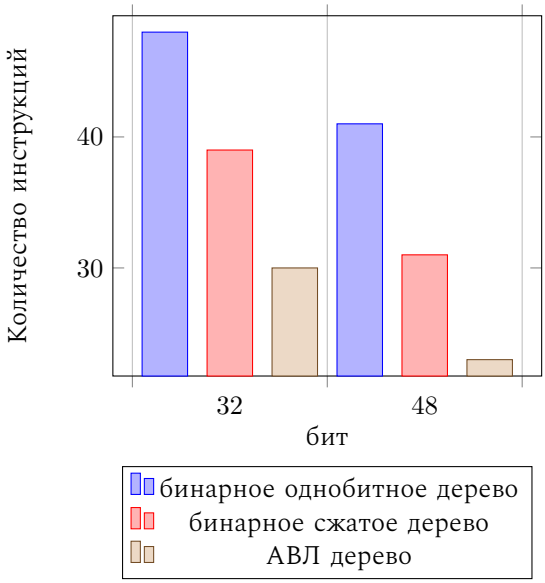

Рис. 9: Сравнение среднего количества инструкций, затраченного на обработку одного пакета.

На следующем графике представлено сравнение реализованных структур данных по максимальному объёму памяти занимаемому структурой данных (Рис. 10). Видно, что только АВЛ дерево удовлетворяет ограничением представленным в пункте **??**, и занимает значительно меньше памяти, чем остальные структуры данных.

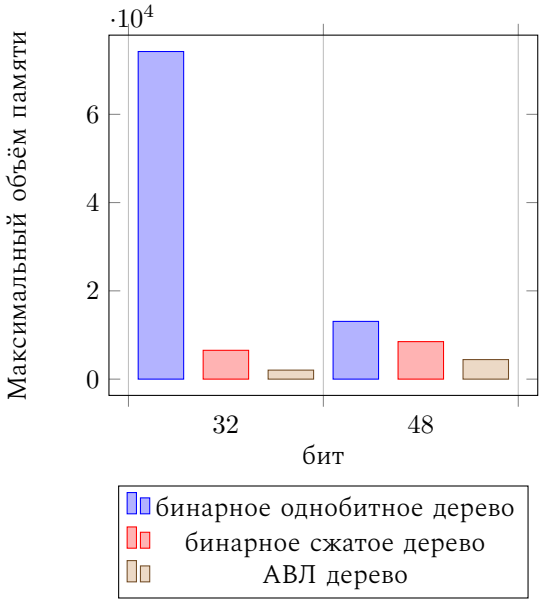

Рис. 10: Сравнение максимального объёма памяти, занимаемого структурами данных.

VIII. Выводы

По результатам экспериментального исследования, только АВЛ дерево удовлетворяет поставленным ограничениям, и может быть использовано в рассматриваемой архитектуре сетевого процессора.

IX. Заключение

В данной работе была рассмотрена проблема классификации пакетов в рассматриваемой архитектуре сетевого процессора. Были разработаны структуры данных для классификации пакетов в рассматриваемой архитектуре. В рамках данной работы были достигнуты следующие результаты:

- Был проведён обзор существующих решений, целью которого было выбрать структуры данных для дальнейшей адаптации и реализации в рамках рассматриваемой архитектуры сетевого процессора. На основе обзора были выбраны АВЛ дерево и бинарное сжатое дерево.
- Выбранные структуры данных были адаптированы под рассматриваемую архитектуру сетевого процессора.
- Адаптированные структуры данных были реализованы с использованием языка программирования *python*.
- Также были разработаны дополнительные программные средства, для проведения экспериментального исследования.
- Было проведено экспериментальное исследование, в результате которого были получены характеристики исследуемых структур данных. Для АВЛ дерева дерева — максимальный объём памяти, занимаемый структурой данных, не превосходит 2 МБ, а среднее количество инструкций на классификацию одного пакета равно 30 и 23 для битовых строк длины 32 и 48 бита соответственно. Для Бинарного сжатого дерева — максимальный объём памяти, занимаемый структурой данных, не превосходит 8.5 МБ, а среднее количество инструкций на классификацию одного пакета равно 39 и 31 для битовых строк длины 32 и 48 бита соответственно.
- Из полученных данных при проведении экспериментального исследования был сделан вывод, что только АВЛ дерево удовлетворяет ограничениям, поставленным в данной работе.

В качестве направления дальнейших исследований, можно указать исследование возможности дальнейшей оптимизации реализованных структур данных, для уменьшения объёма занимаемой памяти структурами данных, а также рассмотрение изменений ар-

хитектуры сетевого процессора и структур, которые требуют изменений в рассматриваемой архитектуре.