Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Кафедра автоматизации систем вычислительных комплексов

Вотинцев Алексей Константинович

Разработка и реализация структуры данных с использованием деревьев поиска для таблиц классификации сетевого процессорного устройства

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:

к. ф.-м. н., доцент Волканов Дмитрий Юрьевич

2022

Аннотация

В работе рассматривается задача классификации пакетов в рамках архитектуры отечественного сетевого процессорного устройства(СПУ).

В рамках работы проведен обзор структур данных, используемых для классификации структур данных. На основе выбранных структур данных были реализованы адаптированные под рассматриваемую архитектуру СПУ деревья. Экспериментальное исследование реализованных структур данных, было проведено на имитационной модели СПУ. В качестве вычислительных ядер конвейра СПУ использовались ядра архитектуы RISC-V.

Содержание

Введение

Стремительный рост объемов трафика и изменение его структуры, необходимость обслуживания увеличивающегося в геометрической прогрессии количества пользователей мобильными и беспроводными технологиями, формирование высокопроизводительных кластеров для обработки Больших Данных и хорошо масштабируемых виртуализированных сред для предоставления облачных сервисов — все это серьезно изменило требования к сетевым средам.

Развитие микропроцессорной техники и телекоммуникаций привело к тому, что сейчас на каждого человека приходится в среднем около 40 чипов, однако появляются все новые сетевые устройства, внесение любых изменений в их существующие конфигурации трудоемко, затратно и практически невозможно без привлечения производителя. Нельзя гарантировать, что программно-аппаратные средства производителя содержат только ту функциональность, которая описана в документации, а в сетях си туация может быть еще сложнее — такая функциональность может быть распределенной. Средства построения сетей сегодня проприетарны, их основной функционал реализован аппаратно и за крыт для изменений со стороны владельцев сетей [17].

Поэтому за последние несколько лет компьютерные сети превратились в ограничивающий фактор развития вычислительной инфраструктуры. Главная проблема состоит в том, что традиционные сети слишком статичны и потому не соответствуют динамике, свойственной современному рынку. Сегодня в области компьютерных сетей мир переживает смещение акцентов с аппаратного уровня на программный [14].

В программно-конфигурируемых сетях(ПКС) уровни управления сетью и передачи данных разделяются за счет переноса функций управления (маршрутизаторами, коммутаторами и т. п.) в приложения, работающие на отдельном сервере (контроллере). Заинтересованность ИТ-компаний в ПКС вы звана тем, что такие технологии позволяют повысить эффективность сетевого оборудования на 25–30%, снизить на 30% затраты на эксплуатацию сетей, превратить управление сетями из искусства в инженерию, повысить безопасность и предоставить пользователям возможность программно создавать новые сервисы и оперативно загружать их в сетевое оборудование [17].

Одним из основных устройств передачи данных в сетях является коммутатор, а основным устройством в коммутаторе является СПУ. СПУ – это программируемый процессор, архитектура которого оптимизирована для использования в сетевых устройствах в целях обеспечения устойчивого режима обработки пакетов (packet processing), а точнее для получения пакета с физического порта, выделение его заголовка, последующая классификация заголовка, и отправка пакета в соответствующий порт.

Для отправки пакета на нужный порт коммутатор производит поиск по таблице классификации СПУ. В таблице классификации хранятся маршруты и метрики, связанные с этими маршрутами. Механизм поиска IP-адресов является ключевым вопросом при проектировании IP-маршрутизаторов. Поиск LPM - это поиск самого длинного префикса среди тех, которые соответствуют начальной подстроке данного сетевого объявления. Поиск EM - поиск по точному соответствию ключа поиска и аргумента поиска.

Поскольку количество памяти СПУ сильно ограничено, основным критерием анализа соответствующих структур будет объем памяти, необходимый для их хранения. Второстепенными критериями будет время поиска и время обновления дерева.

Данная работа посвящена разработке структур данных для поиска в таблицах классификации в рамках архитектуры, используемого в ПКС коммутаторе СПУ. В главе 1 приводится задача поиска по древовидным структурам. 2я глава содержит формальную и неформальную постановки задач, а также объяснение задач поиска и введение в используемые виды поиска по древовидным структурам. В главе 3 описаны программно-конфигурируемые сети, протокол OpenFlow, архитектура и функции СПУ. Глава 4 показывает существующие и подходящие по задаче решения древовидных структур, а в главе 5 представлены предложения и улучшения данных структур. В главе 6 кратко описаны реализованные функции трех структур данных и приведен текст заголовочных файлов. 7я глава содержит сравнения данных реализациий: аналитическое, на локальном компьютере и в архитектуре RISC-V при помощи эмулятора QEMU и итог. Глава 8 - заключение работы и глава 9 - список литературы.

1 Цель работы

Целью работы является разработка структуры данных для поиска в таблицах классификации в рамках архитектуры СПУ. Для достижения данной цели необходимо решить следующие подзадачи:

- Изучить существующие структуры данных для поиска в таблицах классификации пакетов для использования в рассматриваемой архитектуре сетевого процессорного устройства.
- Адаптировать выбранные структуры данных для реализации в рамках рассматриваемой архитектуры сетевого процессора.
- Реализовать адаптированные структуры данных и алгоритмы поиска.
- Провести экспериментальное исследование реализованных структур данных.

2 Постановки задачи

2.1 Описание разных типов поиска

2.1.1 Основные понятия

Задача поиска в таблице заключатпеся в поиске одной из N записей. Каждая запись содержит поле ключ, по которому мы можем однозначно идентифицировать данную запись. В общем случае нам требуется N различных ключей, чтобы идентифицировать N различных записей. Алгоритм поиска имеет так называемый аргумент K и суть поиска заключается в нахождении записи в базе данных, для которой ключ равен K. Результатом поиска является либо нужная запись, либо информация, что запись не найдена.

Процесс поиска обычно является самым "времяемким" этапом программы, поэтому важно знать разные типы поиска и понимать, как и в каких случаях их использовать.

Существуют следующие разбиения поиска на внутренний и внешний, статический и динамический, на сравнение ключей и сравнение числовых свойств ключей, использование ключей действительных и преобразованных[12].

Самые распространенные способы поиска - линейный, бинарный, поиск Фибоначчи, интерполяционный, поиск по префиксу.

В работе рассматриваются алгоритмы поиска по максимальному соответствию префикса(LPM) и поиска по точного значению(EM).

2.1.2 Описание LPM поиска

Комбинация адреса назначения и длины маски называется префиксом [2][4][6]. Префикс, который наилучшим образом соответствует адресу назначения пакета, обычно называют самым длинным совпадением префикса. Поиск совпадения с самым длинным префиксом (Longest prefix match) обычно называется поиском совпадения с самым длинным префиксом. Один префикс может иметь произвольную длину (если он не превышает длину сетевого адреса) и состоять из битовой строки, указанной начальной подстроки сетевого адреса. Никакие две записи не содержат одного и того же префикса. Для примера, префикс 00* определяет диапазон адресов от 00000000 до 00111111.

Пример поиска - IP-адрес 01000001 имеет самый длинный префикс, совпадающий с префиксом 010*, полученным из двух совпадающих префиксов 01* и 010*.

Говорят, что два различных префикса перекрываются тогда и только тогда, когда один является правильным префиксом другого. В противном случае они, как говорят, не связаны. Набор префиксов считается не пересекающимся тогда и только тогда, когда любые 2 префикса в наборе не перекрываются друг с другом. Эффективность памяти определяется как объем памяти (в байтах), необходимый для хранения одного байта префикса. Промежуточный узел представляет собой значение следующего прыжка, если он является конечным какой-то приставки. Внутренний узел называется фиктивным узлом, если он не имеет префикса, заканчивающегося на нем [9].

Для IP-префиксов существует два отношения диапазона: принадлежащие и не пересекающиеся. В следующем объяснении под left понимается минимальное значение, покрываемое данным префиксом, под right - максимальное.

Предположим, что существует два префикса X и Y:

- Если $left(X) \leq left(Y) \cap right(X) \geq right(Y)$, $X \supseteq Y$.
- Если $left(X) \geq left(Y) \cap right(X) \leq right(Y)$, $X \subseteq Y$.
- Если $left(X) > right(Y) \cup right(X) < left(Y)$, X и Y не пересекаются.

Свойство LPM поиска 1: Даны два префикса, PA и PB, если |PA| = |PB| тогда PA и PB не перекрываются [9].

Свойство LPM поиска 2: Учитывая префикс PA с длиной n, PA можно представить как объединение 2 префиксов PB и PC длины (n + 1) путем добавления 0 и 1 к PA.

2.1.3 Описание EM

Поиск по точному соответствию проверяет на полное совпадение ключа с аргументом. Поиск по точному соответствию можно осуществлять линейно, бинарно, используя числа Фибоначчи, с применением каких-то знаний расстановки в базе данных (интерполяционный способ) [12].

Пример наивного использования интерполяциооного способа - поиск в словаре, когда человек знает, на какую букву начинается слово и может предположить, какую часть книги желательно открыть.

Точный поиск имеет очень большое применение и создано много структур данных, дабы его поддерживать. Такими структурами данных являются avl tree, rb tree, B tree, оптимальные деревья (вершины которых размещены по принципу частоты запросов поиска), деревья Фибоначчи и так далее.

Основной стратегией данного поиска является разбиение на диапазоны на каждой вершине, где все вершины слева от рассматриваемой имеют ключи меньше, справа - больше. Поэтому, используя такой вид поиска можно быть точно уверенным, что будет получен какой-то результат.

Есть также примеры деревьев, в каждой вершине которых хранится несколько ключей. Они придуманы для выполнения различных задач. Например, B tree, и все его модификации созданы для хранения больших объемов данных. A Interval tree и Segment tree - для поиска по диапазонам данных величины, характеризующей время, которая должна попасть в промежуток между точкой начала и точкой конца какого-то промежутка времени.

2.2 Неформальная постановка задачи

Необходимо разработать древовидную структуру данных для поиска в таблицах классификации в рамках архитектуры сетевого процессора RuNPU. Разрабатываемая структура данных должна удовлетворять следующим требованиям (для таблиц, не превышающих 64 тысячи записей).

- Размер структуры не должен превышать 640 Кб.
- Должны использоваться одинаковые алгоритмы для IPv4 и IPv6.

2.3 Формальная постановка задачи

Требуется разработать метод классификации пакетов на для простой таблицы OpenFlow версии 1.3.

Введены следующие обозначения:

1. n — количество полей в дескрипторе пакета.
2. $D = \{f_1, \ldots, f_n\}$ — дескриптор пакета (конкатенация полей заголовка пакета и метаданных). f_i — i-е поле дескриптора пакета. Поле представляет собой битовую строку.
3. $Match_k = \{v_{k1}, \ldots, v_{kn}, m_{k1}, \ldots, m_{kn}\}$ — структура, определяющая соответствие дескриптора пакета k-му правилу. v_{ki} содержит требуемые значения i-го поля дескриптора пакета. m_{ki} содержит битовую маску, задающую значимые биты.
4. Mr_k — множество всех возможных дескрипторов пакетов, соответствующих $Match_k$.
5. $Priority_k$ — целое число, задающее приоритет k-го правила.
6. $Action_k$ — идентификатор действия, выполняемого над пакетом согласно k-му правилу.
7. $Rule_k = \{Match_k, Priority_k, Action_k\}$ — k-е правило таблицы классификации.
8. q — количество правил в таблице классификации.
9. $Table = \{Rule_1, \ldots, Rule_q\}$ — таблица классификации.

Дескриптор пакета состоит из следующих полей:

1. MAC адрес отправителя (48 бит).
2. MAC адрес получателя (48 бит).
3. IPv4/IPv6 адрес отправителя (32 бита/128 бит).
4. IPv4/IPv6 адрес получателя (32 бита/128 бит).
5. Порт отправителя (16 бит).
6. Порт получателя (16 бит).

7. Номер входного порта (8 бит).

Задача классификации C:

- **Дано:** таблица классификации $Table$ и дескриптор пакета D.
- **Найти:** s: $s \in \{1, \ldots, q\}$, $D \in Mr_s$ и $Priority_s = max_{i:D \in Mr_i} Priority_i$.

Требуется разработать древовидную структуру данных для решения задачи C. Разрабатываемая структура данных должна удовлетворять следующим требованиям (для таблиц, не превышающих 64 тысячи записей).

- Размер структуры не должен превышать 640 Кб.
- Должны использоваться одинаковые алгоритмы для IPv4 и IPv6.

3 Описание предметной области

3.1 Программно-конфигурируемые сети

Программно-конфигурируемые сети ПКС/SDN (Software Defined Networking) – это концепция строительства и эксплуатации сетей передач данных и центров обработки данных, путь к автоматизации, программируемости и открытости сетей.

В традиционных коммутаторах и маршрутизаторах эти процессы неотделимы друг от друга. Поскольку эти задачи – разные, и подходы к решению каждого должно быть – разными. Согласно концепции ПКС, вся логика управления выносится в так называемые контроллеры, которые способны отслеживать работу всей сети. Можно говорить о четырех основных областях применения ПКС: коммутация, контроллеры, виртуализация облачных приложений и виртуализация средств безопасности сетевых решений.

3.2 ПКС Контроллер

Сетевая операционная система (или ПКС контроллер, далее по тексту — контроллер) в OpenFlow является центральным звеном программно-конфигурируемых сетей (ПКС), в котором сосредотачивается весь функционал управления ПКС сетями и осуществляется логически централизованное управление сетевой инфраструктурой и потоками данных в сети. Контроллер – это программное обеспечение, которое запускается на сервере с некоторой установленной операционной системой общего назначения. Совокупность серверов и контроллеров, запущенных на этих серверах и использующих их вычислительные ресурсы, образуют программно-аппаратный комплекс.

В ПКС сети весь «интеллект» управления сетью сосредоточен на контроллере. С контроллером по унифицированному программному интерфейсу взаимодействуют разнообразные приложения, реализующие сетевые сервисы, такие как маршрутизация, балансировка нагрузки, разнообразные протоколы, шлюзы, сетевые экраны, шифрование и другие.

Сервисы контроллера позволяют вынести функциональность, часто используемую различными приложениями, в отдельные модули. Данные сервисы не являются обязательными, однако упрощают создание новых приложений для контроллера. Наличие некоторых сервисов обязательно для нормального функционирования приложений. Другие сервисы могут предоставлять необходимую информацию для некоторых приложений, однако они не требуются для работы всех приложений. Сервисы контроллера могут быть реализованы как в виде отдельных загружаемых модулей, так и непосредственно в ядре контроллера.

Приложение контроллера реализует произвольную функциональность, необходимую пользователю. Приложения могут представлять собой как модуль контроллера, так и самостоятельный процесс, взаимодействующий с контроллером через некоторый интерфейс (например, RESTful). Реализация приложения в виде модуля контроллера обеспечивает высокую скорость обмена сообщениями как между ядром контроллера и приложением, так и между различными приложениями. Однако использование стандартного интерфейса для взаимодействия между контроллером и приложением позволяет создавать приложения на любом языке программирования.

Некоторые приложения могут входить в состав контроллера, набор таких приложений может различаться для различных контроллеров.

3.3 ПКС Коммутатор

Коммутатор состоит из одной или нескольких таблиц классификации потоков и таблицы групп, которые определяют порядок обработки и пересылки пакетов. ПКС контроллер управляет коммутатором.

Просмотр таблиц начинается с первой таблицы классификации потоков и может продолжаться в следующих дополнительных таблицах потоков. Записи о потоках данных просматриваются в приоритетном порядке, с первой соответствующей пакету записи в каждой таблице. Если соответствующая запись найдена в таблице, то выполняются инструкции, связанные с этой записью. Если соответствующая запись не найдена в таблице потоков, то в зависимости от конфигурации коммутатора:

- пакет может быть перенаправлен в контроллер по защищенному каналу;
- сброшен;
- поиск может быть продолжен в следующей таблице.

Инструкции конвейера обработки позволяют пересылать пакеты в последующие таблицы для дальнейшей обработки и позволяют передавать информацию (в виде метаданных) между таблицами.

Запись о потоке может предписывать переслать пакет в определённый порт. Зарезервированные виртуальные порты могут определять общие действия пересылки, такие, как отправка контроллеру, широковещательная (лавинная) рассылка, или «обычная» обработка коммутатором. Виртуальные порты, определенные коммутатором, могут точно определять группы агрегирования каналов, туннели или интерфейсы с обратной связью. Записи о потоках могут также указывать на группы, в которых определяется дополнительная обработка. Группы представляют собой наборы действий для широковещательной рассылки, а также наборы действий пересылки с более сложной семантикой (например, multipath, быстрое изменение маршрута, агрегирование каналов). Механизм групп позволяет эффективно изменять общие выходные действия для потоков. Таблица групп содержит записи о группах; каждая групповая запись содержит список наборов действий над пакетом. К пакету, отправленному в группу, применяются действия одного или нескольких наборов действий из этого списка (в зависимости от типа группы).

3.4 Общее описание СПУ

Основным устроством передачи данных в коммутаторе является СПУ. По определению сетевое процессорное устройство – это программируемый процессор, архитектура которого оптимизирована для использования в сетевых устройствах и обеспечения устойчивого режима обработки пакетов (packet processing).

RuNPU включает в себя 10 вычислительных узлов (decision engine, DE), каждый содержит по 64 кб памяти, которая разделена между данными и действиями. Таблицы классификации с данными тоже распределены между DE, как показано на рисунке 1. В качестве ядер DE используются ядра с архитектуры RISC-V.

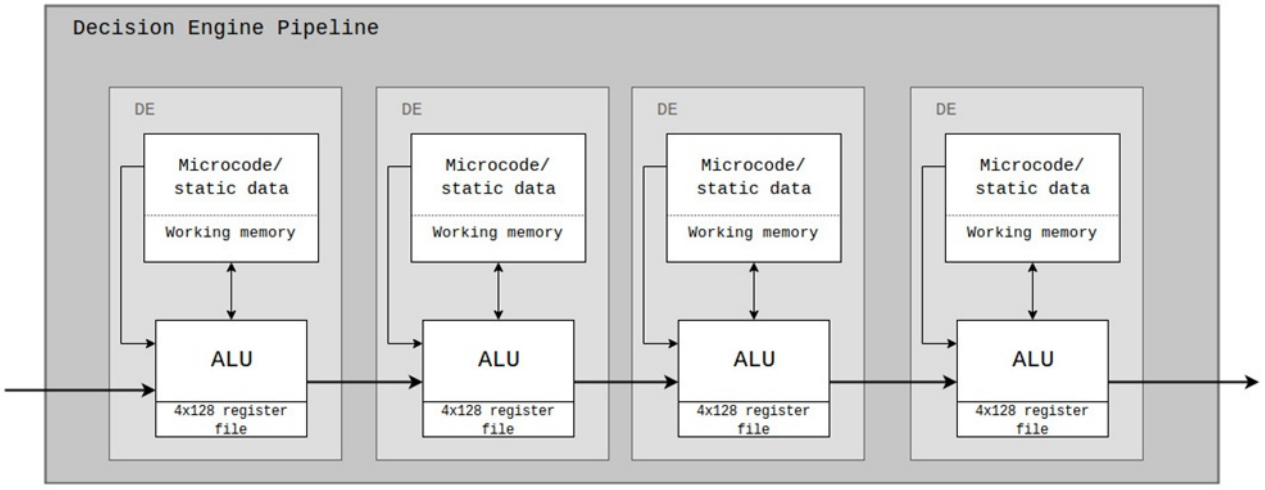

Рисунок 1 - Архитектура конвейера СПУ

Сетевой процессор состоит из процессорного ядра (их может быть и несколько), исполняющего программы, процессоров обработки пакетов и аппаратных ускорителей, разгружающих ЦП от рутинных функций, таких, как вычисление контрольных сумм. Кроме того, в него входят блоки интерфейса с ОЗУ и интерфейсы с высокоскоростной шиной. При этом СПУ оптимизируется под задачу, которую он выполняет, или, как сейчас говорят, под сетевую функциональность.

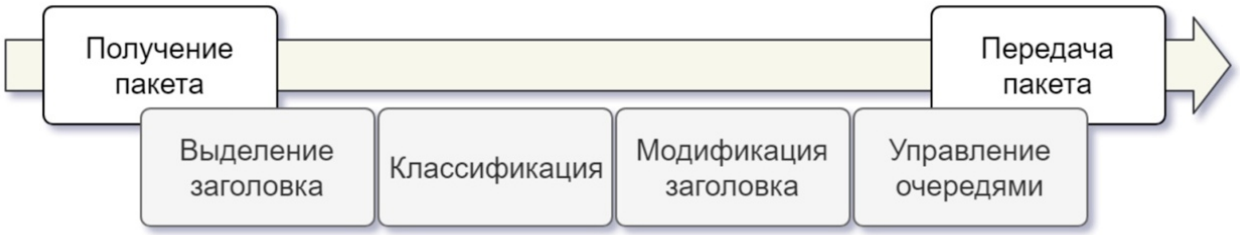

Рисунок 2 - Функции СПУ

На рисунке 2 показана общая схема работы с пакетами в СПУ.[15] Получение пакета с физического уровня включает преобразование сигналов физической среды в биты и дальнейшая группировка битов в пакеты. Передача пакета на физический уровень, аналогично, подразумевает преобразование пакета в сигналы.

Выделение заголовка (или разбор заголовка) означает выделение групп битов (полей) заголовка пакета, значения которых необходимы для обработки в рамках реализуемых устройством протоколов передачи данных.

Классификация пакета означает его идентификацию. Иными словами, должны быть определены действия, которые нужно произвести с пакетом, соответствующим выделенному заголовку (или некоторому подмножеству его полей). Это могут быть действия по модификации заголовка или действия над пакетом, такие как отправка на порт или сброс. Обычно классификация реализуется как поиск заголовка пакета в таблице классификации коммутатора. Таблица классификации состоит из правил – записей, содержащих ключ и соответствующий ему набор действий над пакетом. Формат ключа одной таблицы фиксирован и может состоять из нескольких полей заголовка. Таблиц классификации в устройстве может храниться несколько, они могут работать с ключами разных форматов. Исходя из целей данной работы, наш маршрутизатор работает как на канальном уровне, так и на сетевом уровне. Соответсвенно, и сама таблица и поиск по ней должны быть универсальны для двух данных уровней. На канальном уровне хранится таблица классификации, на сетевом - маршрутизации. Поскольку мы рассматриваем оба этих уровня, в работе будут применяться оба этих названия, но означать одно и тоже. Модификация пакета означает выполнение действий над классифицированным пакетом, полученным из таблицы классификации. Если подходящих записей в таблице классификации найдено не было, выполняется действие по умолчанию (например, сброс пакета или отправка на ЦПУ).

Обычно операция модификации выполняется после поиска в одной таблице. Если требуется классификация по нескольким таблицам, последовательность операций классификации и модификации выполняется несколько раз.

Управление очередями означает дифференцированную обработку пакетов в зависимости от класса обслуживания. Это требует реализации нескольких очередей для каждого из портов и особых алгоритмов планирования их загрузки.

3.5 Таблица классификации

Сетевой адрес	Маска	Адрес шлюза	Интерфейс	Метрика
127.0.0.0	255.0.0.0	127.0.0.1	127.0.0.1	1
0.0.0.0	0.0.0.0	198.21.17.7	198.21.17.5	1
56.0.0.0	255.0.0.0	213.34.12.4	213.34.12.3	15
116.0.0.0	255.0.0.0	213.34.12.4	213.34.12.3	13
129.13.0.0	255.255.0.0	198.21.17.6	198.21.17.5	2
198.21.17.0	255.255.255.0	198.21.17.5	198.21.17.5	1
198.21.17.5	255.255.255.255	127.0.0.1	127.0.0.1	1

Рисунок 3 - Таблица маршрутизации

Таблица классификации (пример на рисунке 3) — электронная таблица (файл) или база данных, хранящаяся на маршрутизаторе или сетевом компьютере, которая описывает соответствие между адресами назначения и интерфейсами, через которые следует отправить пакет данных до следующего маршрутизатора. Является простейшей формой правил маршрутизации.

Таблица классификации обычно содержит:

- Адрес сети или узла назначения, либо указание, что маршрут является маршрутом по умолчанию
- Маску сети назначения (для IPv4-сетей маска /32 (255.255.255.255) позволяет указать единичный узел сети)
- Шлюз, обозначающий адрес маршрутизатора в сети, на который необходимо отправить пакет, следующий до указанного адреса назначения

- Интерфейс, через который доступен шлюз (в зависимости от системы, это может быть порядковый номер, GUID или символьное имя устройства; интерфейс может быть отличен от шлюза, если шлюз доступен через дополнительное сетевое устройство, например, сетевую карту)
- Метрику — числовой показатель, задающий предпочтительность маршрута. Чем меньше число, тем более предпочтителен маршрут (интуитивно представляется как расстояние).

В таблице классификации, поскольку происходит работа с масками подсети, важен порядок записей в таблице. Бывают случаи, когда маска одной записи может быть охвачена маской другой записи, в этом случае приведенные записи нужно хранить в таблице в правильном порядке. Ведь, если мы сначала прочитаем строку с короткой маской и обнаружим, что можем сделать переход по ней, то, не сможем уже проверить строку с длинной маской, которая, в зависимости от конкртетных значений, может быть нужной, и отправим пакет не туда. Из-за такой ошибки на маршрутизатор с длинной маской вообще не будут приходить пакеты. Поэтому на сетевом уровне в таблицах очень важен порядок записей. В работе рассматривается таблица классификации OpenFlow 1.3.

OpenFlow реализует концепцию разделения плоскостей управления передачей данных (ПКС-контроллер) и непосредственно самой передачи (OpenFlow-коммутатор). Протокол OpenFlow позволяет управлять, программировать работу OpenFlow коммутаторов в контуре передачи данных сети.

OpenFlow разделяется на простые и групповые таблицы, в работе же рассматриваются только простые таблицы.

4 Деревья поиска

4.1 Основные понятия

В общем случае дерево называется связный граф, не имеющий циклов. Связный граф - граф, где все вершины связаны, то есть для любых двух вершин существует цепь, соединяющая их. Ориентированный граф G = (V, E) называют пару множеств, где V -множество вершин, E - множество дуг [5][12].

Веткой будем называть отрезок, связывающий две вершины. Корнем дерева называют вершину, в которую не ведут ребра, такая вершина в деревьях может быть только одна. Концевыми вершинами или листьями являются вершины, из которых не исходят дуги. Уровнем узла называют длину (количество вершин) пути. Детьми вершины будем называть вершины, в которые непосредственно входят дуги из этого узла. Потомками вершины будем называть все вершины, расположенные ниже данной вершины. Порядком узла назовем количество его потомков. Родителем узла будем звать узел, из которого в этот узел непосредственно входят дуги. Соответственно, у листьев нет детей, а у корня нет родителя.

В рамках данной работы будем рассматривать деревья с описанной выше структурой. Поддерево — часть древовидной структуры, которая может быть представлена в виде отдельного дерева. Для любого узла поддерева либо должен быть путь в корневой узел поддерева, либо сам узел должен быть корневым. Высота дерева определяется, как его максимальный уровень, длина самого длинного пути от корня к внешнему узлу.

Путем назовем чередующуюся последовательность вершин и не повторяющихся дуг. Длиною пути к узлу является количество ветвей, которые нужно обойти, чтобы дойти от корня до данного узла.

В элементарном случае (бинарном дереве поиска):

- слева от каждого узла находятся узлы, ключи которых меньше или равны ключу данного узла.
- справа от каждого узла находятся узлы, ключи которых больше или равны ключу данного узла.

Особенность использования деревьев поиска — при одном сравнении отсекается примерно половина оставшихся элементов (если дерево сбалансировано, рассмотрим ниже). Количество операций сравнения в этом случае пропорционально log2N, т. е. алгоритм имеет асимптотическую сложность O(log2N). Конечно, нужно учитывать, что предварительно дерево должно быть построено. Поэтому такой алгоритм выгодно применять в тех случаях, когда данные меняются редко, а поиск выполняется часто (например, в базах данных).

4.2 Виды структур поиска

В общем случае алгоритмы поиска IP-адресов можно разделить на следующие категории: подходы на основе trie, скалярного дерева, дерева диапазонов и хэша, и они могут быть реализованы на программном или аппаратном обеспечении, или на том и другом. Аппаратное решение, такое, как TCAM может искать содержимое параллельно, в то время, как программные решения могут извлечь выгоду из низкой стоимости, гибкости и масштабируемости. У каждого есть свои преимущества и недостатки [5][8][10][12].

- Исходная двоичная trie организует префиксы с битами префиксов, чтобы направлять ветвление. В подходах, основанных на trie, IP-поиск выполняется простым обходом trie в соответствии с битами в IP-адресе. Эти конструкции имеют компактную структуру данных, но большое количество узлов trie, следовательно имеем умеренную эффективность памяти. Кроме того, задержка этого решения пропорциональна длине префикса, что делает ее менее привлекательной для протокола IPv6. Также это решение очень затратно по памяти, а для устройств с ограниченной встроенной памятью трудно использовать внешнюю память из-за большого количества этапов конвейера и ограниченного количества контактов ввода-вывода.
- Дерево диапазонов - это подход поиска по значениям, который избегает измерения длины, выполняющего сравнения с расширенными префиксами (полными адресами). Предлагаемый подход позволяет достичь оптимального времени поиска двоичного поиска и может быть обновлен за логарифмическое временя. Еще одним преимуществом этих древовидных подходов является то, что задержка поиска не зависит от длины префикса, а пропорциональна логарифму числа префиксов; поэтому они очень подходят и для IPv6.
- В хешировании используются хеш-функции. Хэш имеет выдающуюся особенность почти O(1) времени выполнения и является широко используемым методом поиска IP-адресов. До сих пор многие сетевые процессоры также имеют встроенный хэш-блок. Тем не менее, этот подход включает в себя много предварительных вычислений, приводящих к трудному обновлению.

- В скалярно-древовидных подходах рассматривают каждый префикс, как число без какой-либо кодировки. Достаточно преобразовать его в префикс и сравнивать с другими префиксами, как скалярные числа. Такой подход позволяет сократить количество префиксных ключей при удвоении пропускной способности памяти. Особенность этого метода также в том, что есть возможность разбить таблицу маршрутизации на несколько деревьев поиска. Это позволяет достичь низкой затраты памяти за счет высокой пропускной способности.

4.3 Подход к выбору структур данных

В известных книгах Дональда Кнута "Искусство программирования"[12], а имеено в 3 томе данной серии, названном "Сортировака и поиск"[12], и совместной работе Томаса Кормена, Чарльза Лейзерсона, Рональда Ривеста и Клиффорда Штайна "Алгоритмы.Построение и анализ"[13] описано множество различных древовидных структур данных. Каждая древовидная структура особенна и уникальна, у каждой есть свои преимущетсва и недостатки. Некоторые созданы, чтобы сократить высоту уже существубщих деревьев, другие ускорили процесс обновления структуры дерева, пожертвовал скоростью поиска. Из всего многообразия структур будут рассмотрены лишь те, которые удовлетворяют требованиям сразу, либо модификации которых могут удовлетворять требованиям поставленной задачи, а именно возможность как поиска по точному соответствию, так и соответствию по максимальному префиксу.

Для осуществления нужного нам поиска обычно используют подход хранения префикса в вершине. Это префискное дерево и его частный вид скалярное дерево. Данные типы деревьев и их некоторые модификации будут рассмотрены в главе 4.4. Способом поиска по такому дереву являестя LPM, описанный во 2.2.2 главе.

Способ поиска по деревьям диапазонов является EM, он описан во 2.2.3 главе. У данного подхода есть тоже много различных модификаций, например часто используемые avl, rb, B деревья. Но, удовлетворяющих условию задачи есть только 1 - дерево интервалов. Это дерево будет описано в главе 4.5.

В следующих главах 4.4 и 4.5 приведено описание упомянутых деревьев поиска. Для сравнения этих деревьев необходимо провести различные тесты, аналитически оценить занимаемую деревьями память довольно сложно.

В работе также приведены авторские модификации префиксного дерева, описанные в главах 5.1 и 5.2, и интервального дерева, содержащиеся в главе 5.3. Глава 6 содержит сравнение деревьев на основанни проведенных экспериментов и вывод, исходя из данных исследований.

4.4 Префиксные деревья (Trie)

Префиксное дерево (Бор) (рисунок 4) [6][12] — структура данных, позволяющая хранить ассоциативный массив, ключами которого являются строки. Префиксное дерево содержит данную строку-ключ тогда и только тогда, когда эту строку можно прочитать на пути из корня до некоторого (единственного для этой строки) выделенного узла. Также каждая вершина префиксного дерева содержит длину хранимого в ней префикса.

Нагруженное дерево отличается от обычных n-арных деревьев тем, что в его узлах не хранятся ключи. Вместо них в узлах хранятся метки, а ключом, который соответствует некоему узлу, является путь от корня дерева до этого узла, а точнее - строка составленная из меток узлов, повстречавшихся на этом пути. В таком случае корень дерева, очевидно, соответствует пустому ключу.

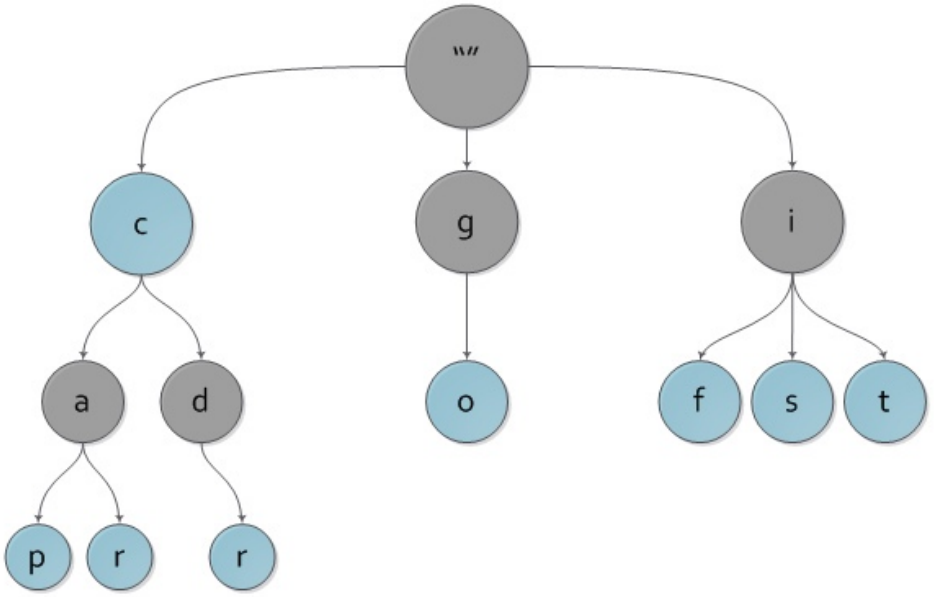

Рисунок 4 - Бор

Структура данного дерева на языке C выглядит так:

```
const int MAX_TRIE_CHILDREN = 26; //количество букв в латинском алфавите
struct trie {
    struct trie* children[size_t MAX_TRIE_CHILDREN] = 0;
    char prefix;
    char key;
}
```

Ключ, соответствующий узлу — конкатенация меток узлов, содержащихся в пути от корня к данному узлу. Из этого свойства естественным образом следует алгоритм поиска ключа (как, впрочем, и алгоритмы добавления и удаления).

Пусть дан ключ Key, который необходимо найти в дереве. Будем спускаться из корня дерева на нижние уровни, каждый раз переходя в узел, чья метка совпадает с очередным символом ключа. После того, как обработаны все символы ключа, узел, в котором остановился спуск, и будет искомым узлом. Если в процессе спуска не нашлось узла с меткой, соответствующей очередному символу ключа, или спуск остановился на промежуточной вершине (вершине, не имеющей значения), то искомый ключ отсутствует в дереве.

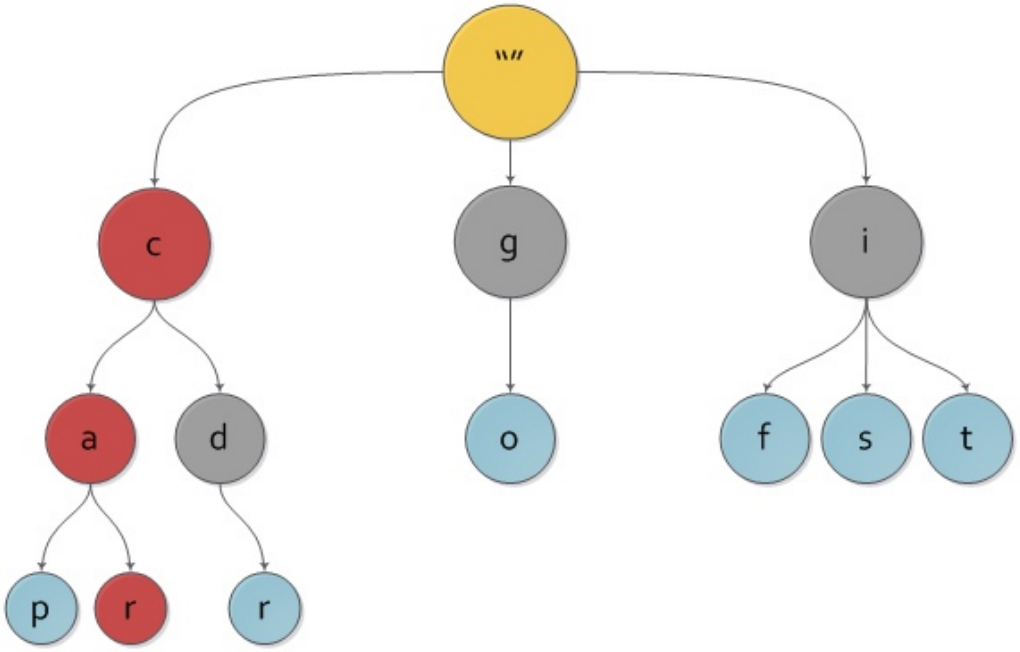

Рисунок 5 - Поиск в префиксном дереве

Пример поиска ключа car в префиксном дереве изображен на рисунке 5.

Поскольку алфавит (или набор возможных элементов в последовательностях) ограничен, то отдельный уровень Trie можно сортировать за линейное время относительно количества дочерних узлов и размера алфавита O(n + K) при помощи алгоритма сортировки подсчетом (Counting Sort).

4.4.1 Сжатые деревья

Представляет собой корневое дерево, каждое ребро которого помечено каким-то символом так, что для любого узла все рёбра, соединяющие этот узел с его сыновьями, помечены разными символами. Оно преобразуется из обычного префиксного дерева путем объединения вершин по данному алгоритму: если у вершины только 1 дочерняя вершина, то они объединяются в одну вершину(рис. 6).

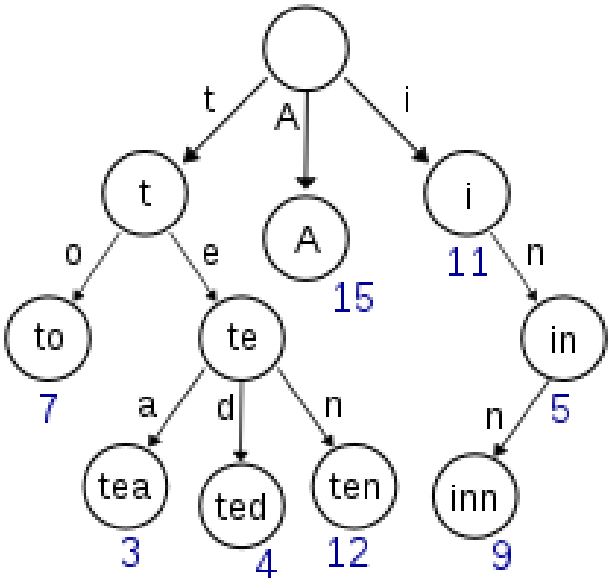

Рисунок 6 - Сжатое префиксное дерево

Структура сжатого префиксного дерева на языке C выглядит так:

```
const int MAX_TRIE_CHILDREN = 26; //количество букв в латинском алфавите
struct trie_compressed {
    struct trie_compressed* children[size_t MAX_TRIE_CHILDREN] = 0;
    char* prefix;
    char key;
}
```

4.4.2 Скалярное дерево

Скалярное дерево (рисунок 7) [7][8][9][10][11] — префиксное дерево, что хранит в вершине префикс в скалярном виде.

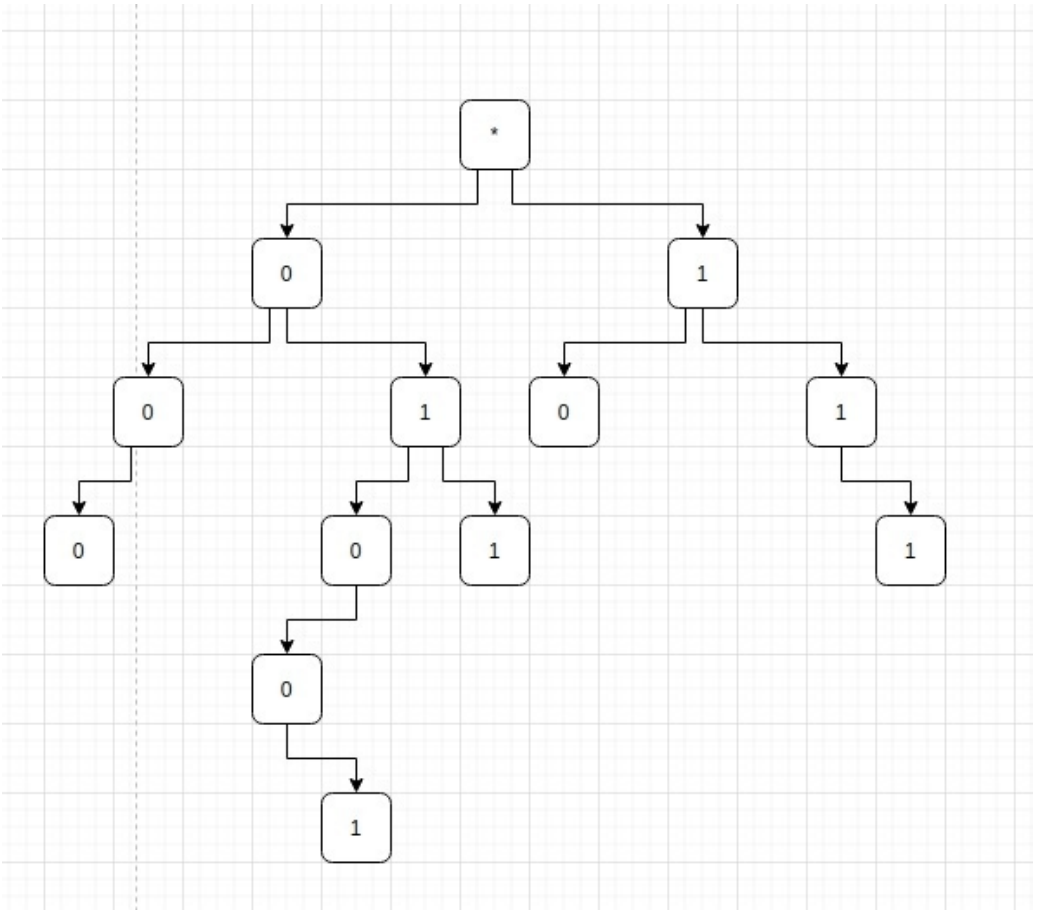

Рисунок 7 - Числовое префиксное дерево

Структура однобитного префиксного дерева на языке C выглядит так:

```
struct trie_bit {
    struct trie_bit* left_child;
    struct trie_bit* right_child;
    char prefix;
    char key;
}
```

Обычное однобитное дерево поиска содержит по 1 биту на каждой своей вершине.

Для его построения достаточно выделять первый бит из оставшейся строки, записывать его, смотреть последующий за ним бит и, в зависимости от этого, переходить в левое поддерево или правое поддерево. Высота такого дерева соответствует длине самой длинной строки.

Процесс поиска в общем случае для скалярных деревьев:

- Если данная вершина — лист, и соответственно есть доступ к искомому полю — вернуть значение данного поля.

- Делать сдвиг вправо числом, которым осуществляем поиск lengthNumber - lengthPrefix раз, сделать сравнение получившегося значения с prefixTop.
- Если совпадают — уменьшить число lengthNumber на lengthTop, узнать значение бита, идущего в числе, сразу после этого префикса, перейти на соответствующую вершину. Перейти к шагу 1.
- Если не совпадают — число не найдено, вернуть значение ошибки.

4.5 Интервальные и сегментные деревья

Интервальные и сегментные деревья по своей сути похожи на avl и rb деревья, только в каждой вершине вместо одного ключа для поиска они хранят три [3][12][13].

4.5.1 Общая концепция диапазонных деревьев

Перед тем как описать дерево интервалов нужно показать АВЛ и красно-черные деревья.

АВЛ-дерево (представлено на рисунке 3)[12] — это двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм.

Структура АВЛ дерева на языке С выглядит так:

```
struct avl {
    struct avl* left_child;
    struct avl* right_child;
    int elem;
    char key;
}
```

Особенность заключается в том, что в бинарном АВЛ дереве высота левого поддерева любого узла отличается не более чем на 1 от высоты правого поддерева этого узла.

Он требует лишь новых 2х битов на узел и никогда не использует более O(log n) операций поиска по дереву или вставки элемента. Эти деревья называются сбалансированными. Также сбалансированные деревья хороши при наличии большого количества элементов.

Если после выполнения операции добавления или удаления, коэффициент сбалансированности какого-либо узла АВЛ-дерева становиться равен 2, т. е. |h(Ti, R)-h(Ti, L)|=2, то необходимо выполнить операцию балансировки. Она осуществляется путем вращения (поворота) узлов – изменения связей в поддереве. Вращения не меняют свойств бинарного дерева поиска, и выполняются за константное время. Всего различают 4 их типа:

1. Малое правое вращение.
2. Большое правое вращение.
3. Малое левое вращение.
4. Большое левое вращение.

Свойства красно-черных деревьев:

1. Каждый узел окрашен либо в красный, либо в черный цвет (в структуре данных узла появляется дополнительное поле – бит цвета).
2. Корень окрашен в черный цвет.
3. Листья (так называемые NULL-узлы) окрашены в черный цвет.
4. Каждый красный узел должен иметь два черных дочерних узла. Нужно отметить, что у черного узла могут быть черные дочерние узлы. Красные узлы в качестве дочерних могут иметь только черные.
5. Пути от узла к его листьям должны содержать одинаковое количество черных узлов (это черная высота).

Вставка элементов в красно-черное дерево (удаление вершины из красно-черного дерева) начинаются со вставки (удаления) элемента, как в обычном бинарном дереве поиска. Только здесь элементы вставляются в позиции NULL-листьев. Вставленный узел всегда окрашивается в красный цвет. Далее идет процедура проверки сохранения свойств красно-черного дерева 1..5. Вставка требует до 2 (при удалении до 3) поворотов.

Красно-черное дерево является сбалансированным деревом порядка 2[12]. По сравнению с avl-tree, r-b tree будет иметь меньшее количество балансировок, значит и процессы добавления и удаления элементов в среднем будут проходить быстрее. Но скорость поиска по этой же причине будет уступать.

Рисунок 8 демонстрирует обычные виды АВЛ дерева.

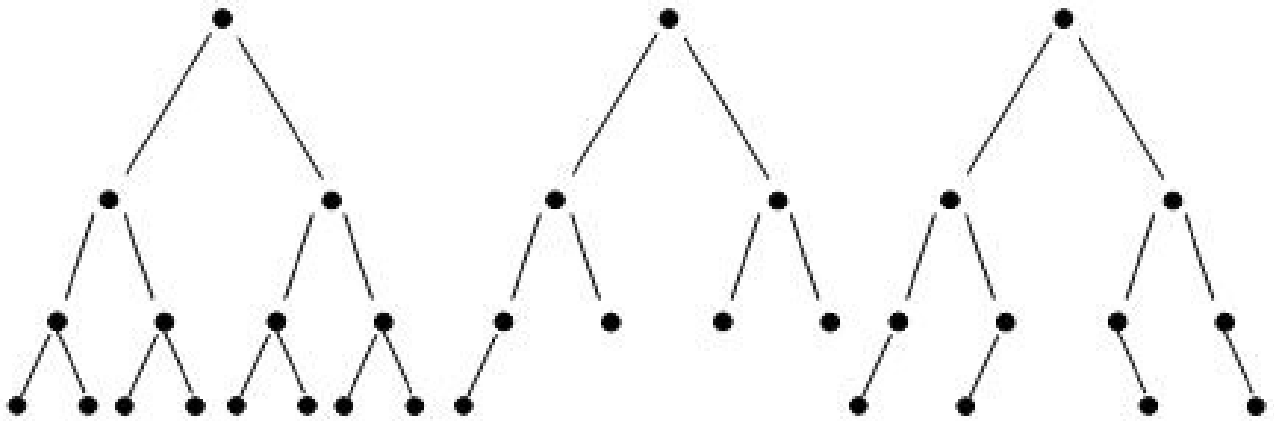

Рисунок 8 - АВЛ дерево

4.5.2 Интервальное дерево

Деревья интервалов: двоичное дерево, хранящее 3 значения - левую и правую границу диапазона данной вершины, максимальное значение поддерева, начинающегося в данной вершине [13].

Структура интервального дерева на языке С выглядит так(если интервальное дерева на основе АВЛ дерева - length, на основе RB дерева - color):

```
struct interval {
    interval* left_child;
    interval* right_child;
    int low_border;
    int upper_border;
    int max;
    char key;
    unsigned char color_or_length;
}
```

Процесс удаления и добавления вершин совпадает с АВЛ деревом, но в дереве интервалов используют в качестве ключа только нижнюю вершину диапазона. Соответственно, все вершины, у которых начало диапазона меньше находятся слева от рассматриваемой вершины, больше - справа. Тем самым получается сбалансированное по левым концам диапазона каждой вершины дерево.

Поиск в классическом дереве интервалов осуществляется так - если вершина слева не нулевая и максимальное значение в вершине слева больше, чем искомое значение - ищем в левом поддереве, иначе в правом. Поиск заканчивается, когда мы находим отрезок, который содержит искомое значение или спускаемся по нулл указателю.

Поскольку все действия данного дерева на каждой вершине не отличаются от АВЛ дерева, то скорость поиска, скорость добавления и скорость удаления составляет O(log n).

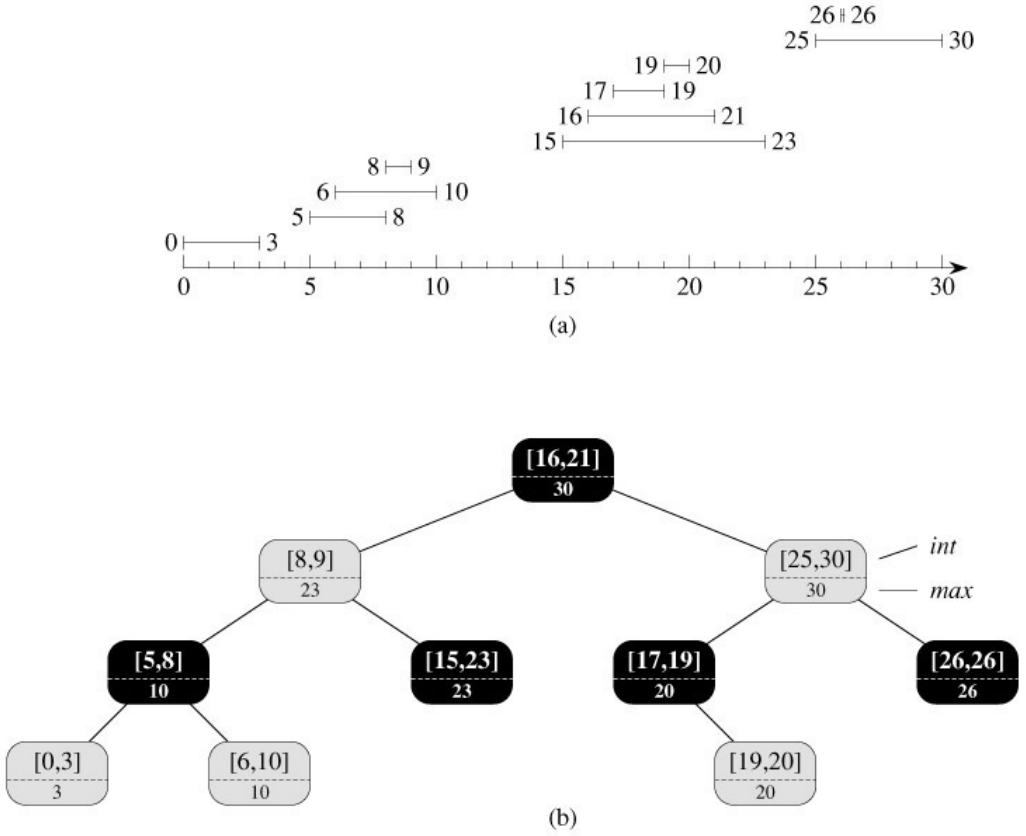

Рисунок 9 - Интервальное дерево

На рисунке 9 показано интервальное дерево, полученное из rb дерева. Такое дерево, только из-за базовой структуры, будет также иметь по всем показателям O(log n).

4.6 Вывод

В данном разделе представлены структуры данных, которые могут быть используемы в СПУ. Для более эффективного использования памяти СПУ необходима модификация данных структур, которая будет представлена в следующем разделе.

5 Предложенные структуры данных

5.1 Сжатое однобитное дерево

Применяя сжатие (рисунок 6) на однобитных деревьях (рисунок 7), получаем сжатые однобитные деревья (рисунок 10).

В таком дереве поиска нужно в каждой вершине обязательно выделять память для хранения целого ip-адреса и длины значащей части данного адреса. При обходе каждая вершина в сжатом дереве полностью хранит префикс, поэтому скорость поиска, как в простом trie, равна O(k).

Любая вершина сжатого однобитного дерева(как и у обычного сжатого дерева), имеющая лишь 1 потомка, объединяется с потомком.

Поэтому каждая вершина сжатого дерева имеет либо 2 потомка, либо не имеет их вовсе, что позволяет максимально сократить количество вершин. Это делает сжатое дерево более эффективным для небольших наборов строк (особенно если сами строки достаточно длинные), и также для наборов, имеющих небольшое количество длинных префиксов.

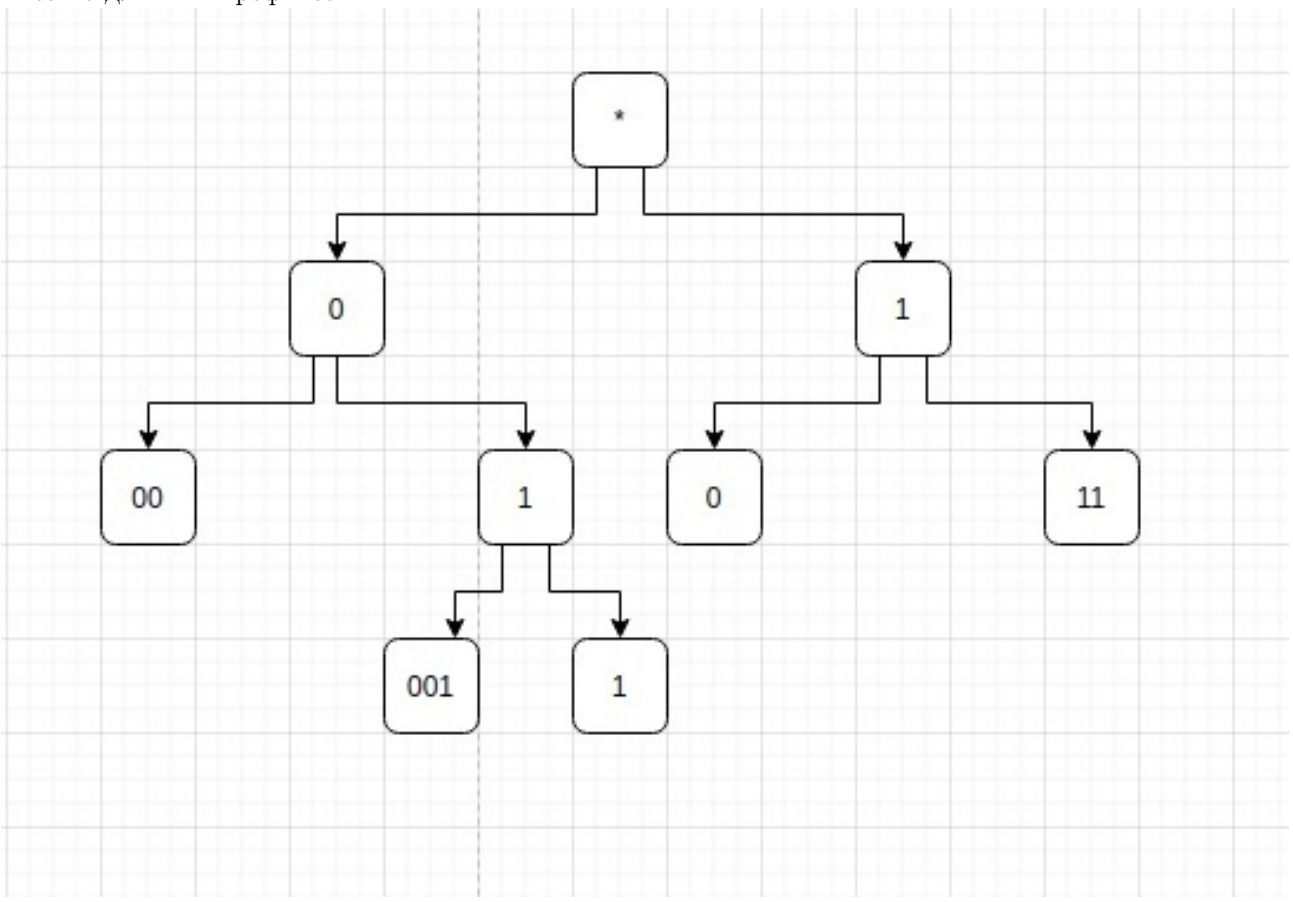

Рисунок 10 - Сжатое числовое дерево

В общем случае метод построения сжатого дерева является таким. Начальным шагом является построение префиксного дерева из заданной таблицы маршрутизации. Вторым шагом будет преобразование этого дерева к виду, когда каждая не листовая вершина имеет 2 дочерних вершины. Это осуществляется путем объединение тех вершин, что имеют только 1 дочернюю вершину со своей дочерней вершиной.

5.2 Предложенная версия скалярного дерева

При объединении вершин однобитного префиксного дерева в сжатое бинарное можно сделать ограничение на длину префикса, хранимого в вершине. Наименьшая память, занимаемая различными типами данных - 8 бит(1 байт).

Вершина модифицированного скалярного дерева содержит поле префикса из 8 бит. Помимо префикса должно быть поле значащей длины префикса.

Структура вершины выглядит так:

struct scalar
    scalar* left;
    scalar* right;
    unsigned char keyNode;
    char length;
    char key; ;

	Сжатое дерево	Скалярное дерево
Префикс меньше 9 бит	40 бит	16 бит
Префикс от 9 до 19 бит	40 бит	32 бита
Префикс от 19 до 25 бит	40 бит	48 бит
Префикс от 25 до 32 бита	40 бит	64 бит

При обходе каждая вершина в сжатом дереве полностью хранит префикс, поэтому скорость поиска, как в простом trie, равна O(k).

Поскольку длина адреса в IPv4 составляет 32 бита, то при хранении больших объемов данных, скорость поиска будет соответствовать сжатому дереву.

При переходе с IPv4 на IPv6 каждая вершина в сжатом дереве будет содержать 128 бит, отводимых на хранение префикса, и 8 бит для длины. В дереве с ограниченной длиной префикса также вершина будет содержать 16 бит.

Процесс добавления вершины в модифицированное скалярное дерево:

1. Корень ничего не хранит в себе, проверить первый бит и перейти в левое или правое поддерево.

2. Если вершина пустая, нужно создать количество вершин, записав в первую из них наибольшее количество значащих битов, которые пока еще не записаны в дереве, и длину данного префикса, если префикс больше 8 бит - создаем по такому же принципу еще нужное количество вершин из оставшихся значений ключа поиска и длины, преобразуем их в дерево.

3. Если не пустая вершина - смотрим длину префикса, сравниваем нужные биты в ключе поиска с битами, которые содержит вершина поиска.

4. Если они совпадают — находим бит в ключе поиска, что идет сразу после последнего бита префикса, совпадающего с данной вершиной. Если он 1 — переходим в правое поддерево, если 0 - в левое.

5. Если же префикс не совпадает, то оставляем совпадающую часть ключа в вершине и его длину. Находим несовпадающую часть ключа вершины, объединяем ее со всеми ключами потомков, спускаясь вниз, пока у потомка не будет 2-е дочерних вершины или она не будет листовой. Фиксируем нужную информацию для листовой вершины — указатель на структуру, которая содержит данные, иначе - на вершину, чтобы дальше работать с указателями на дочерние вершины.

6. Создаем дерево по пункту 2 с оставшейся частью ключа и его длины. Создаем второе дерево с найденными в предыдущем абзаце значениями. В зависимости от их первых значащих в префиксе битов, делаем из этих деревьев левое и правое поддерево в ранее несовпадающей по префиксу вершины.

Процесс удаления вершины:

1. Если вершина пуста — вернуть значение несовпадения.

2. Если ключ вершины является префиксом ключа поиска и эта вершина не листовая - спускаемся на дочернее поддерево, по указанному следующему биту ключа поиска. На своем пути поиска фиксируем последнюю и предпоследнюю вершину, на которой было разбиение на поддеревья.

3. Если же вершина является листовой, префикс также совпал, удаляем все данное поддерево его зафиксированного предка последнего разбиения.

4. У предка предпоследнего разбиения форматируем поддерево, содержащее предка последнего разбиения.

5.3 Модификация интервального дерева

К сожалению, классический вид интервального дерева не подходит для решения поставленной задачи. Нужны дополнения в функции поиска. Ведь бывают случаи, даже показанные на рисунке, когда один интервал лежит в другом (в нашем случае только так и может быть). Классический подход к написанию интервального дерева прекратит свой поиск, когда встретит первую на своем пути вершину, диапазон которой покрывает аргумент поиска. Поскольку все действия данного дерева на каждой вершине не отличаются от АВЛ дерева, то скорость поиска, скорость добавления и скорость удаления составляет O(log n)

Дополнения, необходимые для решения задачи:

- Не останавливать поиск, как только нашелся первый попавшийся интервал.
- Поиск с переходом в левое поддерево остается, но пункт иначе - убирается.
- Если аргумент поиска больше, чем левая граница диапазона, нужно также переходить и в правое поддерево.
- При поиске, помимо нужной информации, возвращать длину диапазона, тогда, если найдется 2 или больше подходящих диапазонов - перейти по самому короткому диапазону.
- Тогда каждая вершина выбирает, какую информацию передать наверх в корень - с левого поддерева, с правого или с самой вершины.

Данные дополнения позволяют проводить правильный поиск по интервальному дереву, узлы которого хранят записи таблицы маршрутизации сетевого процессорно устройства.

6 Реализация

6.1 Общее описание

Предложенные в предыдущем разделе структуры были реализованы в виде 3 программ на языке C. Общая схема работы следующая:

1. Реализованные на C древовидные структуры данных скомпилированы в ассемблер архитектуры RISC-V
2. С помощью эмулятора QEMU проведены тесты на архитектуре RISC-V

Полная реализация, а также ассемблерный код ассемблера локальной машины и архитектуры RISC-V, а также текст данной работы представлены здесь [20]. Все 3 структуры(Сжатое дерево, Скалярное дерево и Интервальное дерево на языке C) вместе содержат примерно 1000 строк кода. Далее опишем подробней архитектуру RISC-V и эмулятор QEMU, использоавнный в работе.

6.2 Описание RISC-V

Систему команд RISC-V с 2010 года разрабатывают в Университете Беркли (Калифорния, США), где решили, что необходим новый единый и открытый набор команд для всех типов систем — от микроконтроллеров до высокопроизводительных систем. К 2010 году грамотного стандарта для системы команд CPU как программно-аппаратного интерфейса не было. Кроме того, закрытость популярных систем команд (а на тот момент почти все были закрытыми, кроме SPARC) создала коммерческие предпосылки для развития открытого стандарта [16].

Система команд RISC-V имеет ряд отличительных особенностей.

1. Отсутствие неявных внутренних состояний. Результат любой операции (кроме команд перехода) всегда помещается в регистр общего назначения, так как в RISC-V нет, например, флагов состояний, которые устанавливает инструкция cmp в x86. Вместо этого результат команды сравнения помещается в один из регистров общего назначения.
2. Отсутствие предикатных инструкций, так как в скалярном коде выигрыш от них небольшой.
3. Компактный базовый набор инструкций, всего их 39.
4. 32 регистра общего назначения. Это вдвое больше, чем у RISC-аналогов.
5. Поддержка ослабленной модели памяти (Relaxed Memory Model) в базовом наборе инструкций.
6. Наличие спаренных (fused) операций «сравнение + переход», уменьшающих размер программного кода.
7. Инструкции ускорения вызова функций и выполнения операторов.

Применение таких открытых технологий, как RISC-V, играет ключевую роль при решении проблемы обеспечения импортонезависимости, поскольку отказ от использования и разработки закрытых решений позволит как сохранить инвестиции, так и обеспечить работоспособность критически важных систем независимо от политической конъюнктуры. При наличии множества производителей оборудования, поддерживающих стандарт RISC-V, теряет смысл установка любых закладок в процессоры — такой процессор всегда можно заменить на другой.

6.3 Описание QEMU

QEMU(Quick Emulator) - это программа, которая используется для эмуляции программного обеспечения разных платформ. Она распространяется бесплатно и имеет открытый исходный код. Работает во всех популярных операционных системах - Microsoft Windows, Linux, MacOS, а также ее можно запускать на Android. Помимо эмуляции, поддерживает технологии аппаратной виртуализации (Intel VT и AMD SVM) на x86-совместимых процессорах Intel и AMD.

Программа QEMU имеют следующие преимущества и особенности:

1. Поддерживает два режима эмуляции: пользовательский режим [User-mode] и системный режим [System-mode].

   (a) Пользовательский режим эмуляции позволяет процессу, созданному на одном процессоре, работать на другом (выполняется динамический перевод инструкций для принимающего процессора и конвертация системных вызовов Linux).
   (b) Системный режим эмуляции позволяет эмулировать систему целиком, включая процессор и разнообразную периферию.

2. Может сохранить и восстановить состояние виртуальной машины со всеми запущенными программами. Гостевой операционной системе не нужно патчей для запуска внутри QEMU.

3. Может эмулировать сетевые карты (разных моделей), которые разделяют подключения принимающей системы, делая трансляцию сетевых адресов, что позволяет эффективно гостю использовать ту же сеть, как хозяину. Виртуальные сетевые карты также могут подключаться к сетевым картам других экземпляров QEMU или к локальным TAP интерфейсам.

4. Объединяет несколько сервисов, чтобы обеспечить связь хостовой и гостевой систем. Она также может загружать ядра Linux без загрузчика.

5. Не зависит от наличия графических методов вывода на хост-системе. Вместо этого он может позволить получить доступ к экрану гостевой ОС с помощью интегрированного сервера.

6. Не требует прав администратора для запуска.

7. Виртуальная машина может взаимодействовать со многими типами физических аппаратных средств хоста. К ним относятся: жесткие диски, диски CD-ROM, сетевые карты, аудио интерфейсы и USB-устройств. USB-устройства могут быть полностью эмулированы.

8. Виртуальные образы дисков могут быть сохранены в специальном формате qcow или qcow2, которые занимают столько дискового пространства, сколько гостевая ОС на самом деле использует. Таким образом, эмулируемый 100 GB диск может занимать всего несколько сотен мегабайт на хосте. Формат qcow2 также позволяет создавать наложения образов, которые записывают разницу из другого файла базового (немодифицированного) образа. Это дает возможность возвращать содержимое эмулируемого диска в предыдущее состояние.

6.4 Реализация Сжатого дерева

Реализованы основные функции **insert_in_compressed_tree**, **delete_from_compressed_tree** и **search_in_compressed_tree** для добавления, удавления вершины из древовидной структуры и поиска элемента в ней. Задачи удаления и добавления довольно объемны, поэтому основная логика этих задач разбита на несколько функций, поэтому присутствуют функции **add_in_compressed_tree** и **del_from_compressed_tree**. Также функция полного удаления дерева **delete_compressed_tree**. Поиска длины общего префикса в вершине и элементе **length_prefix_in_compressed_tree**. Функция построения вершины или дерева, для хранения 1 адреса - **create_compressed_top**. И функцию перестройки дерева при изменении элементов корневой вершины - **remake**. Далее приведен полный список основных используемых функций:

- void delete_compressed_tree(struct compressed** root);
- struct compressed* create_compressed_top(unsigned int number, char key, char length);
- struct compressed* build_top(struct data info, struct compressed* add_ver);
- struct compressed* remake(struct compressed* tmp);
- unsigned char length_prefix_in_compressed_tree(struct compressed* tmp, struct data info, char* flag);
- void break_top(struct compressed** tmp, struct data info, char length, char bit);
- void add_in_compressed_tree(struct compressed** head, struct data info);
- void insert_in_compressed_tree(struct compressed** head, struct data info);
- char search_in_compressed_tree(struct compressed* head, struct data info);
- void del_from_compressed_tree(struct compressed* head, struct data info);
- void delete_from_compressed_tree(struct compressed* head, struct data info);

6.5 Реализация Скалярного дерева

Реализованы основные функции **insert_in_scalar_tree**, **delete_from_scalar_tree** и **search_in_scalar_tree** для добавления, удавления вершины из древовидной структуры и поиска элемента в ней. Задачи удаления и добавления довольно объемны, поэтому основная логика этих задач разбита на несколько функций, поэтому присутствуют функции **add_in_scalar_tree** и **del_from_scalar_tree**. Также функция полного удаления дерева **delete_scalar_tree**. Поиска длины общего префикса в вершине и элементе **length_prefix_in_scalar_tree**. Функция построения вершины или дерева, для хранения 1 адреса - **create_scalar_top**. И функцию перестройки дерева при изменении элементов корневой вершины - **remake_scalar_tree**. Далее приведен полный список основных используемых функций:

- void delete_scalar_tree(struct scalar** root);
- struct scalar* create_scalar_top(unsigned char number, char key, char length);
- struct scalar* build_scalar_top(struct data info, struct scalar* add_ver);
- struct scalar* remake_scalar_tree(struct scalar* tmp);
- unsigned char length_prefix_in_scalar_tree(struct scalar* tmp, struct data info, char* flag);
- void break_top_scalar_tree(struct scalar** tmp, struct data info, char length, char bit);
- void add_in_scalar_tree(struct scalar** head, struct data info);
- void insert_in_scalar_tree(struct scalar** head, struct data info);
- char search_in_scalar_tree(struct scalar* head, struct data info);
- void del_from_scalar_tree(struct scalar** head, struct data info);
- void delete_from_scalar_tree(struct scalar** head, struct data info);

6.6 Реализация Интервального дерева

Реализованы основные функции **insert_in_range_tree**, **delete_from_range_tree** и **search_in_range_tree** для добавления, удавления вершины из древовидной структуры и поиска элемента в ней. Задачи удаления и добавления довольно объемны, поэтому основная логика этих задач разбита на несколько функций, поэтому присутствуют функции **add_in_range_tree**, **del_from_range_tree** и **find_in_range_tree**. Данное дерево основано на базе AVL дерева, поэтому есть функции малых и больших левых и правых вращений: **small_left_rotation**, **small_right_rotation**, **big_left_rotation**, **big_right_rotation**, также функции анализа, какое вращение применять для балансирвоки - **fixheight**, **balanced_range_tree**. И для поддержки свойств Интервального дерва и корректирвоки значения max - **fixmax**. Далее приведен полный список основных используемых функций:

- struct range* small_left_rotation(struct range* head);
- void fixmax(struct range** head);
- void fixheight(struct range** head);
- struct range* small_right_rotation(struct range* head);
- struct range* big_left_rotation(struct range* head);
- struct range* big_right_rotation(struct range* head);
- void balanced_range_tree(struct range** head);
- void add_in_range_tree(struct range** head, struct range* new_elem);
- void insert_in_range_tree(struct range* head, struct data elem);
- int find_in_range_tree(struct range* head, int number, char* key);
- int search_in_range_tree(struct range* head, struct data info);

- void delete_elem(struct range** head, struct range** new_elem);
- void del_from_range_tree(struct range** head, struct range* elem);
- void delete_from_range_tree(struct range** head, struct data elem);

7 Экспериментальное сравнение

7.1 Сравнение структур без учета особенностей СПУ на архитектуре x86

Нынешние представление древовидных структур для хранения таблиц маршрутизации сетевых процессорных устройствах - либо обычное сжатое префиксное дерево(рисунок 6), либо однобитное скалярное(рисукнок 7).

В первом случае нужно в каждой веришне хранить динамический массив строк, где каждый байт представляет лишь 1 бит ip-адреса. Это позволяет объединить нужные вершины, уменьшить высоту дерева, но хранение динамического массива и постоянное его изменение при добавлении/удалении новых ключей делаеет задачу не оптимальной с точки зрения памяти. Во втором случае - в каждой вершине хранится лишь 1 символ, нет динамического массива. Но высота дерева ровно 32, также это порождает огромное количесво ненужных вершин.

Для аналитического сравнения с представленными в работе деревьями, начнем с сжатого скалярного дерева. Оно, также как и сжатое дерево, может хранить весь адрес. Но память, выделяемая на его хранение одинаковая - всегда 5 байт (4 байта для хранения ключа структуры и 1 байт для его длины). Тем самым мы избегаем динамическое выделение памяти. Но в нем также есть проблема, ведь при большом количестве записей длина префикса в конкретной вершине стремиться к 1 байту, а количество памяти на вершине также будет статично, 5 байт.

Модифицировав этот подход, мы можем ограничить количесво выделяемой памяти на вершину до 2х байтов (1 байт ключа структуры и 1 байт его длины). Проблема очень большого количества адресов в таблице маршрутизации, где большая часть вершин будет хранить по 1 биту адреса, все также не решена.

Дерево интервалов дает другой подход к решению поставленной задачи. В отличие от префиксных деревьев, дееревья диапазонов всегда имеют определенное количество вершин, равное количеству записей, то есть заранее известное. Да и обход такиих деревьев можно сказать имееет константное время. Но, хоть вершин в деревьях диапазона намного меньше, чем в деревьях префиксных, для нормального функционирования в каждой вершине нужно намного больше информации (13 байт).

Помимо этого, деревья интервалов, будучи простыми сбалансированными бинарными деревьями, можно перевести в массив(так называемую кучу или heap). Основным принципом при таком переходе является положить корень в нулевой элемент массива, его левого ребенка в 1, а правого во 2 элемент массива. И дальше, если мы переложили вершину в i-ый элемент массива, то его левого ребентка мы помещаем в 2*i+1 элемент, а правого в 2*i+2 элемент. Но это не получится сделать с префиксными деревьями, так как разница в высоте между двумя листями этого дерева может быть огромна. Тем самым можно избавиться от хранения всех указателей. Да и данный переход в массив подходит только для статичных таблиц маршрутизации, так как при частом изменении дерева придется сильно менять массив, но это сделать не получится из-зи отсутвия указателей и придется заново строить дерево, его изменять и только после этого перекладывать в массив.

	Compressed	Scalar	Range
10	544B - 640B	608B - 928B	480B
100	2.6 - 5Kb	2.7Kb - 7Kb	4.7Kb
1000	19Kb - 47Kb	22Kb - 50Kb	47Kb
4000	70Kb - 160 Kb	74Kb - 163Kb	188kb
10000	152Kb - 393Kb	207Kb - 550Kb	470Kb
20000	302Kb - 812Kb	380Kb - 1Mb	940Kb
32000	466Kb - 1Mb	496Kb - 1.5Mb	1.4Mb
64000	874Kb - 2.2Mb	923Kb - 4Mb	2.8Mb

Таблица 1 - память, выделяемая при помощи malloc

Компиляция программ выполяналась командной **gcc <nameProg>.c -o <nameFile>**.

Сравнение структур, результат которого приведен на таблице 1, проводилось на языке C с помощью **struct malinfo** и его значения **uordblks**(Total allocated space), вычитая данное значение, высчитанное до

построения дерева из значения после построения дерева. Где столбец слева - количество записей в таблице маршрутизации и соответственной количество различных аргуметнов поиска в дереве. Сверху указаны наименования деревьев, а в ячейках таблицы - занимаемая ими память.

Аналитическое сравнение с помощью подсчета количества вершин в дереве и умножения на занимаемую одной вершиной память представлено на рисунке 12. Также взяты средние значения количества вершин при данном количестве записей.

	Compressed	Scalar	Range	Heap
10	280B	341B	220B	140B
100	2.1Kb	2.2Kb	2.2Kb	1.4Kb
1000	17.4Kb	17Kb	21Kb	14Kb
4000	70Kb	72Kb	84Kb	56kb
10000	140Kb	136Kb	210Kb	136kb
20000	290Kb	270kb	420Kb	272Kb
32000	446Kb	370Kb	700Kb	437Kb
64000	957Kb	752kb	1.4Mb	875Kb

Таблица 2 - Аналитический подход

Произведен подсчет оперативной памяти, занимаемой выше указанными структурами данных. Таблица 2 демонстрирует средние показатели различных древовидных структур и количества записей таблицы маршрутизации.

	Compressed	Scalar	Range
10	-	-	-
100	-	-	-
1000	-	-	-
4000	260kb	272Kb	240Kb
10000	353kb	424Kb	400Kb
20000	650Kb	665Kb	692Kb
32000	1003Kb	1100Kb	1120Kb
64000	1480Kb	1600kb	1903kb

Таблица 3 - Выделенная оперативная память

Для получения RSS данных (таблица 3) во время выполениния программы вводилась терминальная команда **ps -aux –sort -rss | grep <nameFile>**

Все просмотренные процессы до начала построения дерева занимали память от 511 до 570 Kb. После, также, как и в примере выше, вычиталось одно значение из другого и средние показатели данных значений вносились в табличку. Разброс показателей до 200Kb, в зависимости от количества записей.

RSS - переводится как постоянное потребление памяти (Resident Set Size) и показывает сколько в момент вывода команды занято в оперативной памяти. Также стоит отметить что он показывает весь стек физически выделенной памяти на указанный процесс в данный момент.

7.2 Сравнение структур данных в архитектуре RISC-V

7.2.1 Компилятор RISC-V

В работе использовался комплилятор **riscv-gnu-toolchain**[18]. Это кросс-компилятор RISC-V C и C++. Он поддерживает два режима сборки: общий набор инструментов ELF / Newlib и более сложный набор инструментов Linux-ELF / glibc.

Для его уставновки использовались команды:

- **$ git clone https://github.com/riscv/riscv-gnu-toolchain**
- **$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev**
- **$ sudo yum install autoconf automake python3 libmpc-devel mpfr-devel gmp-devel gawk bison flex texinfo patchutils gcc gcc-c++ zlib-devel expat-devel**
- **$ sudo pacman -Syyu autoconf automake curl python3 libmpc mpfr gmp gawk base-devel bison flex texinfo gperf libtool patchutils bc zlib expat**
- **$ ./configure –prefix=/opt/riscv**
- **$ make linux**

Сборка по умолчанию нацелена на RV64GC (64-разрядную версию) с помощью glibc, даже в 32-разрядной среде сборки.

Поддерживаемые архитектуры - rv32i или rv64i плюс стандартные расширения (a)tomics, (m) умножение и разделение, (f) loat, (d) ouble или (g) eneral для MAFD. Поддерживаемые ABI: ilp32 (32-разрядный с плавающей запятой), ilp32d (32-разрядный с плавающей запятой), ilp32f (32-разрядный с одинарной точностью в регистрах и двойной в памяти, только для использования в нише), lp64 lp64f lp64d (то же самое, но с 64-разрядной длиной и указателями).

7.2.2 Настройка QEMU с RISC-V

Чтобы настроить эмулятор **QEMU** для запуска с архитектуров **RISC-V** были введены в терминале приведенные команды.

- **$ ./configure –target-list=riscv64-softmmu**
- **$ make -j $(nproc)**
- **$ sudo make install**

После чего создан линукс с **RISC-V** архитектурой командой

- **$ make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- defconfig**

И создано ядро командой

- **$ make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- -j $(nproc)**

BusyBox — набор UNIX-утилит командной строки, используемой в качестве основного интерфейса во встраиваемых операционных системах. Преимуществами этого приложения являются малый размер и низкие требования к аппаратуре. Оно представляет собой единый файл. Для его установки использовались команды[19].

- **$ CROSS_COMPILE=riscvbits-unknown-linux-gnu- make defconfig**
- **$ CROSS_COMPILE=riscvbits-unknown-linux-gnu- make -j $(nproc)**

7.2.3 Тестирование

	Compressed	Scalar	Range
10	-	-	-
100	-	-	-
1000	-	-	-
4000	244kb	272Kb	240Kb
10000	340kb	404Kb	380Kb
20000	605Kb	644Kb	660Kb
32000	962Kb	998Kb	1024Kb
64000	1372Kb	1456kb	1692kb

Таблица 4 - RSS RISC-V

Компиляция программ осущуствлялась командной **riscv-gcc <nameProg>.c -o <nameFile>**

Запуск командой **qemu-riscv64 <nameFile>**. Получал информацию RSS с помощью **ps -aux –sort -rss | grep <nameFile>**. Для компиляции использовался компилятор riscv64 **riscv-gcc**, а запуск производился на эмуляции архитектуры RISC-V в эмуляторе QEMU **qemu-riscv64**.

7.3 Результаты

Как видно на приведенных сравнительных таблицах, лучшие показатели изменяемых таблиц маршрутизации имеет сжатое скалярное дерево. При тестировании оно обошло скалярное дерево из-за одинакового количества памяти, выделяемого при создании объекта обеих структур. Но сами деревья имеют разное количество вершин, поэтому сжатое дерево и занимает меньше памяти.

Если же таблицы маршрутизации статичны и меняются очень редко, то лучшим решением будет создание кучи на основе интервального дерева. То есть, сначала создать интервльное дерево, поместить в него нужные записи и перенести информацию оттуда в кучу. Дальнейшее хранение дерева бесполезно. Тем самым имеем какое-то множество ограниченных массивов, которые создаются лишь раз и для их создания вообще не нужна динамическая память. Данный подход будет занимать наименьшее количество памяти по сравнению со всеми остальными.

8 Заключение

В данной работе рассматривалась задача разработки структуры данных на основе деревьев для поиска в таблицах классификации в рамках архитектуры СПУ RuNPU. В рамках её решения были выполнены следующие задачи:

- Изучены существующие древовидные структуры данных и выбраны несколько типов деревьев для последующего исследования.
- Разработаны новые версии скалярного и интервального деревьев, позволяющие выполнять поставленную задачу эффективнее.
- Программно реализованы скалярное сжатое дерево, модифицированное скалярное дерево, интервальное дерево.
- Проведен сравнительный анализ предложенных деревьев.

Как показало экспериментальное исследование, для изменяемых таблиц маршрутизации лучше всего подойдет сжатое скалярное дерево, для статических - куча, на основе интервального дерева.

В качестве направлений дальнейших исследований можно указать:

- Оптимизация приведенных реализации по количеству тактов.
- Проведение экспериментов с другими архитектурами ядер, например MIPS.
- Рассмотрение разделения структур данных между несколькими DE.
- Рассмотрения одновременного поиска в нескольких таблицах маршрутизации.

Список литературы

[1] 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, 2020. - с. 118–132.

[2] Kobayashi, M., Murase, T., Kuriyama, A. (2000, June). A longest prefix match search engine for multi-gigabit IP processing. In 2000 IEEE international conference on communications. ICC 2000. Global convergence through communications. Conference record (Vol. 3, pp. 1360-1364). IEEE.

[3] Suri, S., Varghese, G., Warkhede, P. R. (2001, November). Multiway range trees: Scalable IP lookup with fast updates. In GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No. 01CH37270) (Vol. 3, pp. 1610-1614). IEEE.

[4] Tzeng, H. Y. (1998, November). Longest prefix search using compressed trees. In Proceedings of IEEE Global Communication Conference.

[5] Ruiz-Sánchez, M. Á., Biersack, E. W., Dabbous, W. (2001). Survey and taxonomy of IP address lookup algorithms. IEEE network, 15(2), 8-23.

[6] I. Aoe, K. Morimoto, and T. Sato. An efficient implementation of trie structures. Software—Practice and Experience, 22(9):695–721, September 1992..

[7] Chuprikov, P., Kogan, K., Nikolenko, S. (2017, October). General ternary bit strings on commodity longest-prefix-match infrastructures. In 2017 IEEE 25th International Conference on Network Protocols (ICNP) (pp. 1-10). IEEE.

[8] Le, H., Prasanna, V. K. (2011). Scalable tree-based architectures for IPv4/v6 lookup using prefix partitioning. IEEE Transactions on Computers, 61(7), 1026-1039.

[9] Behdadfar, M., Saidi, H., Alaei, H., Samari, B. (2009, April). Scalar prefix search: A new route lookup algorithm for next generation internet. In IEEE INFOCOM 2009 (pp. 2509-2517). IEEE.

[10] Le, H., Prasanna, V. K. (2009, April). Scalable high throughput and power efficient ip-lookup on fpga. In 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines (pp. 167-174). IEEE.

[11] Sun, Q., Zhao, X., Huang, X., Jiang, W., Ma, Y. (2007, August). A scalable exact matching in balance tree scheme for IPv6 lookup. In ACM SIGCOMM (pp. 27-31).

[12] Кнут, Д. (2022). Искусство программирования. Том 3. Сортировка и поиск. Litres.

[13] Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. (2009). Алгоритмы. Построение и анализ:[пер. с англ.]. Издательский дом Вильямс.

[14] Уезеролл, Д., Таненбаум, Э. (2012). Компьютерные сети.

[15] Xu, B., Jiang, D., Li, J. (2005, March). HSM: A fast packet classification algorithm. In 19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers) (Vol. 1, pp. 987-992). IEEE.

[16] Waterman, A. S. (2016). Design of the RISC-V instruction set architecture. University of California, Berkeley.

[17] Смелянский, Р. Л. (2012). Программно-конфигурируемые сети. Открытые системы. СУБД, 9, 15-26.

[18] URL: https://github.com/riscv-collab/riscv-gnu-toolchain (дата обращения 10.04.2022)

[19] URL: https://RISC-V-getting-started-guide.readthedocs.io/en/latest/linux-qemu.html (дата обращения 10.04.2022)

[20] URL: https://github.com/Arrgentum/diplom (дата обращения 12.05.2022)