На правах рукописи

Филиппов Илья Викторович

Исследование и разработка систем программирования масштабируемых высокопроизводительных сетевых функций в облачных инфраструктурах

Специальность 05.13.11
«Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Автореферат
диссертации на соискание учёной степени
кандидата технических наук

Москва — 2019

Работа прошла апробацию на кафедре микропроцессорных технологий в интеллектуальных системах управления федерального государственного автономного образовательного учреждения высшего образования «Московский физико-технический институт (национальный исследовательский университет)».

Научный руководитель: Плоткин Арнольд Леонидович, д.т.н., профессор, заведующий кафедрой микропроцессорные технилогии в интеллектуальных системах управления МФТИ

Научный консультант: Мелик-Адамян Арег Фрикович, к.т.н.

Ведущая организация: публичное акционерное общество «Институт электронных управляющих машин им. И.С. Брука».
Защита состоится 28.06.2019 в 12:00 на заседании диссертационного совета ФРТК. 05.13.11.002 по адресу: 141701, Московская область, г. Долгопрудный, Институтский переулок, д.9.
С диссертацией можно ознакомиться в библиотеке и на сайте Московского физико-технического института (национального исследовательского университета):
https://mipt.ru/education/post-graduate/soiskateli-tekhnicheskie-nauki.php.

Работа представлена «16» апреля 2019 г. в Аттестационную комиссию федерального государственного автономного образовательного учреждения высшего образования «Московский физико-технический институт (национального исследовательского университет)» для рассмотрения советом по защите диссертаций на соискание ученой степени кандидата наук, доктора наук в соответствии с п. 3.1 ст. 4 Федерального закона «О науке и государственной научно-технической политике».

Общая характеристика работы

Актуальность темы. Исторически компьютерная сеть представляла из себя конечные устройства - хосты (hosts), на которых производилась обработка и собственно сеть, предоставляющую только передачу пакетов. Однако со временем всё больше вспомогательных задач выносилось с конечных устройств в инфраструктуру передачи пакетов. Развитие компьютерных сетей привело к тому, что сеть перестала состоять только из физического оборудования для передачи пакетов. Современная компьютерная сеть включает в себя как неотъемлемую часть обработку пакетов на промежуточных узлах – сетевые функции. Приведём примеры сетевых функций:

- NAT – Network Address Translation – трансляция сетевых адресов
- IPsec – IP security protocol – семейство протоколов шифрования и аутентификации на границе защищённого и не защищённого сегментов сети
- AntiDDOS – предотвращение атак «отказ в обслуживании»
- DPI – Deep Packet Inspection – глубокий анализ пакетов
- Billing – тарификация и выставление счетов

Маршрутизаторы различных уровней модели сети также являются сетевыми функциями, часть из них могут включать в себя вышеперечисленные примеры.

В большинстве случаев сетевая функция реализуется с помощью так называемых middle-box – программно-аппаратного обеспечения, специализированного под высокопроизводительное решение одного класса задач. Различные middle-box последовательно выстраиваются в цепочку - так называемый, bump-in-the-wire подход. В сетях интернет провайдеров бывает несколько десятков стоящих последовательно сетевых функций. Реализация сетевых функций с помощью middle-boxes имеет ряд недостатков:

1. отсутствие гибкости топологии сети: необходимо физически перемещать оборудование для изменений
2. сложность управления: необходимость ручной переконфигурации при изменениях топологии, функциональности, масштабирования, отказах оборудования
3. высокие эксплуатационные затраты на внесение изменений в сетевую функцию
4. высокие капитальные затраты на приобретение специализированного оборудования, количество поставщиков которого ограничено
5. высокие затраты на масштабирование и отказоустойчивость: дублирующее оборудование для пиковых мощностей и обеспечения отказоустойчивости большую часть времени простаивает

Для преодоления указанных недостатков была предложена технология виртуализации сетевых функций – Network Function Virtualization

– NFV [1]. Концепция сетевой архитектуры NFV предполагает создание сетевых функций как обычных программ (что решает проблему эксплуатационных расходов), исполняющихся на стандартном, не специализированном оборудовании, (что решает проблему капитальных расходов). При этом предлагается размещать функциональность каждого middle-box в отдельной виртуальной машине, находящейся на общем сервере, связанной с другими виртуальными машинами виртуальным коммутатором vSwitch, эмулирующим физическую сеть (что решает проблему гибкости топологии и автоматической переконфигурации и масштабирования). Функции, размещённые внутри виртуальных машин называются виртуальными сетевыми функциями - Virtual Network Function - VNF.

Подход NFV решил часть недостатков, однако привнёс новые проблемы. Значительно повысилась сложность общего администрирования аппаратуры, гипервизора, виртуального коммутатора, виртуальных машин и сетевых функций внутри виртуальных машин. Из-за сложности конфигурации виртуального коммутатора, его правила в большинстве случаев задаются вручную, что означает прежнюю привязку к топологии и сложность администрирования (проблемы 1 и 2). Методы разработки сетевых функций предполагают использование низкоуровневых примитивов и средств программирования, чтобы обеспечить необходимую производительность, что значительно повышает время и стоимость создания новых сетевых функций, что увеличивает эксплуатационные расходы (проблема 3). Это привело к тому, что сетевые функции создаются теми же поставщиками, которые ранее выпускали middle-box, что означает прежние высокие капитальные затраты (проблема 4). Кроме того, технология виртуализации не предполагает абстрагирования уровня работы с сетью, поэтому появились зависимости от оборудования, поддерживающего специальные возможности повышения производительности.

С другой стороны, происходит быстрое развитие облачных инфраструктур, предлагающих стандартизированный подход к управлению вычислительными мощностями. Возможность использования облачных инфраструктур для осуществления сетевых функций решило бы проблемы виртуальных машин, однако необходим инструментарий, обеспечивающий автоматизацию развёртывания, функционирования, конфигурируемости, отказоустойчивости сетевой функции в облачной модели выполнения.

В этих условиях актуальной проблемой становится упрощение, а значит удешевление, создания сетевых функций, при сохранении того же уровня производительности, в том числе, в облачных инфраструктурах.

Целью диссертационной работы является создание системы, упрощающей программирование и исполнение производительных сетевых функций.

Задачи, которые необходимо решить для достижения цели:

1. Разработка модели представления сетевых функций, учитывающей необходимость эффективного программирования и масштабирования в рамках существующих вычислительных ресурсов.
2. Разработка модели выполнения и масштабирования выполнения сетевых функций в облачных инфраструктурах
3. Разработка эффективного алгоритма масштабирования выполнения сетевых функций для достижения заданной пропускной способности при использовании минимальных необходимых ресурсов.
4. Разработка программного обеспечения, позволяющего создавать сетевые функции, на основе предложенных моделей. Сравнение с существующими аналогами.

Объектом исследования являются сетевые функции.

Предметом исследования является эффективная разработка высокопроизводительных сетевых функций.

Научная новизна результатов, предложенных в диссертационной работе заключается в следующем:

1. Предложена эффективная реализация модели представления сетевых функций в виде соединённых, предопределённых, конфигурируемых блоков, учитывающая масштабирование и высокую производительность
2. Разработан алгоритм масштабирования сетевых функций в многоядерной системе в условиях отсутствия предварительных данных о структуре входного потока пакетов и поведении пользовательской функции
3. Предложена методология использования сетевых функций в облачных инфраструктурах

Практическая значимость. На основе предложенных в диссертационном исследовании моделей и алгоритмов было создано программное обеспечение, использующееся или планируемое к использованию в коммерческих проектах следующих корпораций:

- Intel – разработка микроэлектроники
- SmartEdge – системный интегратор
- Glasnostic – системный интегратор
- XCloud – телекоммуникационные технологии
- Vectra – компьютерная безопасность

А также в одном из крупнейших интернет провайдеров США и в одном из крупнейших провайдеров кабельного телевидения США.

Методы исследования базируются на базовых принципах системного и сетевого программирования, правилах проектирования программного обеспечения, теории алгоритмов и теории графов.

На защиту выносятся следующие основные результаты

1. Алгоритм масштабирования сетевых функций в многоядерной системе в условиях отсутствия предварительных данных о структуре входного потока пакетов и поведении пользовательской функции
2. Эффективная реализация модели представления сетевых функций в виде соединённых, предопределённых, конфигурируемых блоков, учитывающая масштабирование и высокую производительность
3. Методология использования виртуальных сетевых функций в облачных инфраструктурах

Достоверность научных положений и эффективность предложенного решения проверялись с помощью разработанного программного обеспечения путём замеров производительности (пропускной способности и задержки откликов) созданных сетевых функций. Соответствие результатов необходимым целям обеспечивалось путём научно-технических консультаций с представителями отраслевой индустрии.

Апробация результатов работы. Разработанная модель и программное обеспечение были представлены на следующих конференциях и семинарах:

1. Software Engineering Conference Russia 2017 – крупнейшая конференция разработчиков в восточной Европе
2. Gophercon 2016, 2018 – крупнейшая конференция разработчиков на языке GO
3. Intel SWPC 2018 – конференция создателей программного обеспечения компании Intel
4. DPDK Summit 2018 – крупнейшая конференция разработчиков на платформе DPDK
5. Семинары корпорации Intel 2017, 2018, в том числе с представителями телекоммуникационной индустрии

Личный вклад автора заключается:

- В адаптации модели представления сетевых функций в виде соединённых, предопределённых, конфигурируемых блоков под потребности высокой производительности и масштабирования
- В разработке алгоритма масштабирования в условиях отсутствия предварительных данных о структуре входного потока пакетов и поведении пользовательской функции
- В разработке архитектуры созданного программного обеспечения
- В программной реализации ключевых компонент созданного программного обеспечения

Публикации. Основные результаты по теме диссертации изложены в 5 печатных изданиях, 2 из которых изданы в журналах, индексируемых SCOPUS, 1 — в журналах, индексируемых RSCI и 2 — в тезисах докладов.

Содержание работы

Во введении обоснована актуальность диссертационной работы, сформулирована цель и аргументирована научная новизна исследований, дан перечень задач, необходимых для достижения цели, показана практическая значимость полученных результатов, представлены выносимые на защиту научные положения.

В первой главе диссертационной работы дан обзор существующих решений и исследований для высокопроизводительной обработки сетевого потока пакетов.

В параграфе 1.1 даётся историческая справка проблемы высокопроизводительной обработки пакетов на стандартном оборудовании. Перечисляются недостатки ядра обычной, не специализированной под обработку пакетов, операционной системы (далее ОС) на примере ОС Linux, влияющие на производительность, даются этапы обработки пакеты сетевым стеком:

Таблица 1 — этапы приёма-передачи пакета - действия с памятью

NIC принимает пакет из сетевого провода	
NIC записывает пакет в кольцевой буфер приёма	NIC
Драйвер NIC инициирует прерывание "новый пакет"	
ОС реагирует на прерывание, запуская обработку пакета	
ОС инициализирует sk_buf структуру с информацией о пакете	
ОС перемещает пакет в буфер сокета в памяти ОС	DMA
Первичная обработка: подсчёт контрольных сумм, дешифровка	
Кэш промах - пакета нет в кэш. Пакет помещается в кэш	КЭШ
ОС обрабатывает пакет сетевым стеком, определяет протоколы	
Пользователь вызывает системный вызов "read"	
Переключение контекста с пользователя на ОС	
ОС копирует пакет в буфер пользователя	COPY
Переключение контекста с ОС на пользователя	
Пользователь может обрабатывать пакет	
Пользователь вызывает системный вызов "write"	
Переключение контекста с пользователя на ОС	
ОС копирует пакет в буфер сокета в памяти ОС	COPY
Драйвер NIC записывает пакет в кольцевой буфер передачи	DMA
NIC передаёт пакет в сетевой провод	NIC
Драйвер NIC инициирует прерывание "пакет послан"	
ОС реагирует на прерывание, переключение на пользователя	

В параграфе 1.2 анализируются способы повышения производительности обработки сетевых задач на не специализированном оборудовании - исключения этапов из таблицы 1

В качестве аппаратного ускорения рассматриваются технологии DDIO – Direct Data Input Output и DCA – Direct Cache Access, позволяющие записывать данные из PCIe устройств напрямую в LLC кэш, минуя оперативную память; RSS – Receive Side Scaling - масштабирование на стороне приёма, обеспечивающая распределение принимаемых сетевой картой пакетов по входным очередям драйвера карты; Offloading, позволяющая выполнять стандартные операции (например, подсчёт контрольных сумм) на сетевой карте; Hugepages - позволяющая оперировать страницами памяти большого размера.

В качестве системного ускорения приводятся Packet_MMAP - использование промежуточной памяти между ядром и пользовательским процессом для исключения копирования; NAPI - New Application Programm Interface использующаяся для перевода драйвера сетевой карты из режима прерываний в режим опрашивания для приёма пакетов с большой скоростью; BPF и eBPF - Berkeley Packet Filter, позволяющие выполнение пользовательской предобработки пакета до исполнения сетевого стека ОС. Рассказывается о новой технологии AF_XDP, которая использует возможности eBPF, чтобы передавать приходящие пакеты из драйвера сразу в пользовательское приложение без копирования и системных вызовов.

В качестве ускорения с помощью специализированных драйверов приводятся netmap и PF_Ring, позволяющие напрямую передать пакет в пользовательское приложение с помощью подмены системных вызовов. Упоминаются системы OpenOnload, Sniffer10G, Ixy. Отдельно анализируется система DPDK, включающая собственные драйвера, работающие в пользовательском пространстве, систему распределения памяти, основанную на заранее выделенных блоках, использующих большие страницы, кольцевые буферы, позволяющие одновременные чтение и запись без задержек.

В результате обзора делается вывод о том, что существующие программно-аппаратные решения способны обеспечить высокопроизводительную обработку пакетов на стандартном оборудовании. Однако, создание функций с помощью рассмотренных систем затруднено, так как необходимы глубокие знания системной архитектуры, много времени тратится на сопутствующие операции: взаимодействие потоков, выделение буферов, опрашивание устройств, а не на решение сетевых задач.

В параграфе 1.3 рассматривается использование высокоуровневых абстракций. Вводится модель представления функции как графа обработки пакетов - ориентированного графа без циклов, каждая вершина которого представляет собой блок - отдельную часть функциональности

по обработке пакета сетевой функцией, а каждое ребро - направление пересылки пакетов от одного блока к другому:

$$
\begin{aligned}
&G := (V, E) \\
&\forall v \in V, v - \text{блок обработки пакетов} \\
&\forall e \in E, e = \{v1, v2\} - \text{порядок выполнения } v2 \text{ после } v1
\end{aligned} \tag{1}
$$

Для появления пакетов в графе используются блоки входа $v_{in} : \nexists e, e = \{v, v_{in}\}$, получающие пакеты из любых источников: сетевой карты, pcap файлов, ядра ОС или генерирующие их самостоятельно. Аналогично блоки выхода пакетов из графа $v_{out} : \nexists e, e = \{v_{out}, v\}$ могут посылать их по сети, записывать в файл или удалять.

Рассматриваются системы, совмещающие высоко оптимизированные методы из параграфа 1.2 и модель графа (1). Впервые понятие графа встречается в системе Click. Его расширяет фреймворк VPP, основной идеей которого является одновременная обработка пакетов, объединённых в вектор. Click и VPP в первую очередь задумывались для создания маршрутизаторов, поэтому они больше предназначены для конфигурирования и переиспользования существующих блоков, чем для написания новых. С другой стороны, они предлагают динамическую конфигурацию – управление функциями во время выполнения.

Парадигму статичной конфигурации, задаваемой в исходной коде, используют системы Snabb и NetBricks, вводящие понятие предопределённых, но конфигурируемых функций, которые одновременно являются оптимизированными и быстро создаваемыми.

В результате обзора делается вывод о том, что в VPP и Click для создания новой функции требуется преодолевать высокий порог вхождения, а Snabb и NetBricks не предоставляют масштабирования, а также блоков, находящихся в режиме постоянного опрашивания.

В параграфе 1.4 рассказывается о существующих концепциях развёртывания сетевых функций. Рассматриваются плюсы и минусы развёртывания на обычном сервере - "bare metall", виртуализации сетевых функций NFV и контейнеризации. Упоминаются SR-IOV - технология создания на сетевой карте нескольких логических портов, PCI passthrough - технология, позволяющая виртуальной машине общаться напрямую с сетевой картой. Отдельно рассматривается технология Программно Конфигурируемой Сети (Software Defined Network - SDN).

В параграфе 1.5 приводятся исследования в области алгоритмов масштабирования и планирования выполнения сетевых функций. Большое количество работ пытаются решить проблему динамического масштабирования. Отметим, что ни одна из перечисленных выше промышленных систем не предполагает динамического масштабирования. Некоторые исследования говорят об оптимальном планировании выполнения в условиях ограниченных ресурсов. В других статьях рассказывается об оптимальном

размещении функций по исполнителям, принимая во внимания пропускную способность связей между ними. Часть работ задаётся вопросом минимизации миграций обработки пакетов с одного вычислительного ресурса на другой, т.к. в этом случае велика вероятность потери части выполненной работы и неэффективная перепосылка пакетов. Некоторые работы касаются проблемы разделения состояния системы между различными исполнителями и эффективного определения разных потоков для распределения нагрузки.

Однако большинство работ только вскользь касаются первичного вопроса о моменте времени для принятия решения о масштабировании функции вверх и вниз.

В параграфе 1.6 даётся постановка задачи. Множество всех пакетов, принятых функцией f ко времени t будем обозначим $IN(t)$. Множество всех пакетов, обработанных функцией f ко времени t обозначим $OUT(t)$. Сетевая функция f должна удовлетворять требованию отсутствия потерь пакетов (2).

$$\forall t, \exists t_1 : IN(t) \subset OUT(t_1) \tag{2}$$

Значение $\frac{|IN(t)|}{t}$, при котором сетевая функция удовлетворяет (2) будем называть пропускной способностью сетевой функции (throughput)

После прихода пакета в сетевую функцию он может или обрабатываться, или ожидать своей очереди. Введём функции $intime(p)$, $processtime(p)$, $waittime(p)$, $resources(p)$ возвращающие время прихода пакета, время обработки пакета, время ожидания пакетом своей очереди и количество ресурсов, использованных при обработке пакета соответственно. Тогда $outtime(p) = intime(p) + processtime(p) + waittime(p)$ - время выхода пакета из функции.

Значение $\frac{\sum_{p \in IN}(outtime(p) - intime(p))}{p}$ будем называть временем выполнения функции f для пакета p. Для пользователей сети эта величина будет определять задержку отклика сети (latency).

Каждый сетевой пакет имеет отправителя $from(p)$ и адресата $to(p)$. Будем называть сетевым соединением S множество пакетов, имеющих общих отправителя и адресата.

Множества $IN(t)$ и $OUT(t)$ состоят из пакетов различных сетевых соединений. Пакеты каждого соединения S упорядочены друг относительно друга по времени входа и выхода из функции. Таким образом, на множествах $IN(t)$ и $OUT(t)$ можно определить отношения частичного порядка.

Тогда эффективное выполнение сетевой функции можно сформулировать в виде задачи многокритериальной оптимизации:

$$\begin{aligned}
&\forall t : \frac{|IN(t)|}{t} \to max \\
&\forall p \in IN : (processtime(p) + waittime(p)) \to min \\
&\sum_{p \in IN} resources(p) \to min \\
&\text{При} \\
&\forall t, \exists t_1 : IN(t) \subset OUT(t_1) \\
&\forall p_1, p_2 \in IN : p_1 \prec_{in} p_2 \Rightarrow p_1 \prec_{out} p_2
\end{aligned} \tag{3}$$

Искомое решение задачи системы программирования сетевых функций должно:

- обеспечивать созданным функциям решение задачи (3)
- обладать высокоуровневыми примитивами и низким порогом входа
- осуществлять автоматическое масштабирование сетевой функции в рамках многоядерной системы
- иметь возможность исполнения в облачной инфраструктуре
- поддерживать возможность автоматического масштабирование в многомашинном окружении

Во второй главе диссертационной работы представлено решение задачи системы программирования сетевых функций.

В параграфе 2.1 предлагается высокопроизводительная реализация модели (1), учитывающая возможность масштабирования - рисунок 1. Вводятся существующие положения:

- Граф статичен, задаётся в исходном коде до начала обработки
- Рёбра $e = \{v1, v2\}$ являются кольцевыми буферами
- Буферы работают по принципу "первый зашёл - первый вышел"
- Блок F ожидает пакеты во входном буфере B_{in}, забирает их, преобразовывает и отдаёт в один из выходных буферов B_{out}

Вводятся предлагаемые положения, повышающие производительность:

- Блоки графа выполняются параллельно без синхронизации
- Каждый блок выполняется одним потоком на отдельном ядре
- Каждый блок работает в режиме постоянного опрашивания
- Буферы позволяют одновременные чтение и запись без задержек
- Блоки могут выполняться в скалярном или векторном режиме
- Буферы позволяют чтение, запись вектора пакетов единовременно
- Управляющая система централизована и находится вне графа
- Некоторые блоки могут быть сгруппированы в один блок - сегмент
- Статичными является не конкретные блоки графа, а его топология

В параграфе 2.2 даётся базовая методология масштабирования. Если пакеты будут приходить во входной буфер блока слишком часто, блок

R – входная функция (приём пакетов), F – Блоки обработки пакетов, S – выходная функция (посылка пакетов), круги – промежуточные буферы, прямоугольники – ядра процессора

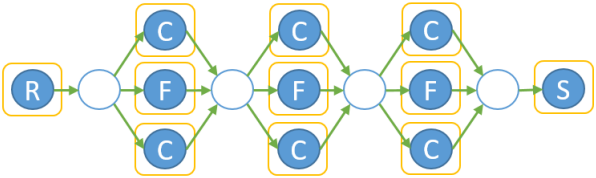
Рис. 1 — Простейший граф

перестанет успевать их обрабатывать и его входной буфер, служащий выходным для предыдущего блока, переполнится. Три сценария поведения предшествующего блока при переполнении его выходного буфера:

- замещать пакеты в выходном буфере последующими
- сбрасывать пакеты
- приостанавливать исполнение (в случае получения пакетов из сетевой карты остановка приведёт к сбросу пакетов сетевой картой)

Основной задачей становится как быстрое освобождение промежуточного буфера для приёма новых пакетов. Предлагаемая базовая методология масштабирования в рамках одной системы использует механизм клонирования вычислительно сложных блоков на свободные ядра в системе (рисунок 2).

Создание клона означает создание идентичной копии блока с теми же входными и выходными буферами, выделение свободного ядра для его выполнения, старт копии (далее – клон) на выделенном ядре. Это возможно, так как промежуточные буферы допускают одновременные чтение и запись из разных потоков. Удаление клона означает остановку выполнения копии блока, добавление ядра в список свободных для переиспользования в сетевом или другом приложении на данном оборудовании.



C – клон блока F соответственно

Рис. 2 — Базовая методология масштабирования

Предлагаемая методология приводит к перемешиванию пакетов, так как любой клон может получить любой пакет из буфера. Это противоречит пункту 4 задачи (3). Данное противоречие будет разрешено далее, в параграфе 2.4.

Рассмотрим простейший сценарий - единовременное статичное выделение максимально возможного количества клонов каждому блоку. Во-первых, это противоречит пункту 3 задачи (3). Во-вторых невозможно решить задачу эффективного статичного предоставления каждому блоку

максимума клонов из ограниченного набора ядер в условиях неизвестной и изменяющейся во времени сложности выполнения каждого блока. Это означает, что выделение клонов должно быть динамическим. Будем рассматривать централизованный планировщик, располагающийся на отдельном ядре и динамически принимающий решения о выделении клонов.

Отметим, что нехватка ядер в одной системе для создания клонов, означает невозможность поддержания в этой системе заданной пропускной способности. В этом случае необходимо решить задачу масштабирования в многомашинном окружении, рассмотренную в параграфе 2.6.

В параграфе 2.3 строится эвристический алгоритм масштабирования. Задача масштабирования сетевой функции в рамках одной системы формулируется следующим образом: *Определение моментов времени для добавления и удаления клонов каждого блока, в условиях неизвестной зависимости входного потока пакетов от времени и зависимости сложности выполнения каждого блока от времени и выполнении условий задачи (3).*

Даётся замечание о неприменимости классических методов, например, "Управления с прогнозирующими моделями" (Model Predictive Controller - MPC) из-за невозможности аналитически предсказать достаточное и, при этом, минимальное количество ресурсов, необходимое для обработки текущего входного потока пакетов. Предлагаемый алгоритм должен иметь итерационную структуру.

Для решения задачи масштабирования планировщик должен периодически опрашивать систему. Период наблюдения планировщика за системой назовём тактом планировщика. После каждого опрашивания системы планировщик должен принимать решение о создании или удалении клона.

Первым рассматривается пороговый алгоритм создания клонов. Пусть блок F имеет j клонов, а во входном буфере b_{in} находятся $len(b_{in})$ пакетов. Тогда в избежание переполнения буфера необходимо добавить клон C_{j+1} блока F, если: $len(b_{in}) > MAX$. Для минимизации количества используемых ядер клон необходимо остановить, если с обработкой входящего потока может справится меньшее количество ядер процессора. Это означает остановку клона C_{j+1}, если: $len(b_{in}) < MIN$. Значения MIN и MAX в общем случае задаются пользователем.

Добавим реализацию обратного давления - "backpressure propagation" - клон не должен быть добавлен, если все выходные буферы b_{out} блока F заполнены - это приведёт лишь к переиспользованию ядер.

$$\begin{cases} len(b_{in}) > MAX \\ \exists i : len(b_{out_i}) < MAX \end{cases} \Rightarrow \text{ добавить } C_{j+1}$$
$$len(B_{in}) < MIN \Rightarrow \text{ остановить } C_j \qquad (4)$$

Лемма 1. *Пусть клоны последовательных блоков A и B имеют константные пропускные способности a и b соответственно. Тогда, если $a > b$, $a \neq nb$, $n \in \mathbb{N}$, и для масштабирования применяется алгоритм пороговых значений (4) то $\nexists$ такого количества клонов J блока B, которое бы сохранялось сколь угодно долго.*

Следствием леммы является невозможность приведённого алгоритма выйти на стабильное количество клонов, бимодальность последнего клона блока из-за недостаточности условия удаления клонов.

Кроме того, если система не справляется с входным потоком пакетов, буфер всегда будет перегружен, и клоны будут постоянно добавляться, что в конечном итоге приведёт к замедлению работы системы из-за расходов на синхронизацию.

Назовём скоростью блока с J клонами за любой такт планировщика i количество пакетов $V_{J,i}$, обработанных всеми клонами блока за этот такт. Назовём сложностью блока с J клонами за любой такт планировщика i время $t_{J,i}$, затраченное в среднем на обработку одного пакета за этот такт.

Лемма 2. *Пусть для блока с J клонами последний такт до добавления J клона - z. Скорость блока в последний такт до добавления J клона - $V_{J-1,z}$. Тогда, если сложность блока не меняется со временем, то для любого последующего такта a ($a > z$): $V_{J,a} < V_{J-1,z}$, является необходимым и достаточным условием для уменьшения количества клонов.*

Для использования леммы 2 введём вектор скоростей V, где элемент $V[j]$ - скорость блока F при наличии j клонов. Вектор формируется при добавлении клонов - перед добавлением $j+1$ клона в $V[j]$ сохраняется текущая скорость - cur. Тогда (4) можно переписать в виде:

$$\begin{cases} len(b_{in}) > MAX \\ \exists i : len(b_{out_i}) < MAX \quad \Rightarrow \text{ добавить } C_{j+1} \\ V[j+1] > cur \end{cases} \qquad (5)$$
$$cur < V[j-1] \Rightarrow \text{ остановить } C_j$$

Во-первых, такой подход решает проблему бимодальности - клон не будет остановлен, пока не упадёт скорость. Во-вторых, клоны не будут создаваться, если известно, что скорость с большим количеством клонов заведомо меньше текущей, что обеспечит стабильную работу системы, если она перегружена.

Однако в общем случае (наиболее распространённом) не выполняется главное условие леммы 2 - сложность блока зависит от длины и других параметров входящих пакетов, а эти параметры обычно различны. В этом случае, необходимо разбить время на интервалы, в течении которых изменением сложности можно пренебречь, и ввести механизм периодических

проверок соседних значений количества клонов для стабилизации вектора скоростей:

$$\begin{array}{l}\text{Периодически полагать } V[j+1]=0 \Rightarrow \text{добавить } C_{j+1}\\ \text{Если } cur < V[j-1] - \text{клон будет удалён, обновив } V[j+1]\\ \text{Периодически полагать } V[j-1]=0 \Rightarrow \text{остановить } C_j\\ \text{Если } len(B_{in}) > MAX - \text{клон будет восстановлен, обновив } V[j-1]\end{array} \quad (6)$$

Предложенная схема стабилизации (6) требует постоянных проверок. Это приводит к искусственной нестабильности количества ядер. Кроме того, итерационная природа алгоритма приводит к большому времени отклика на изменения поведения при большом количестве клонов (обновлении всего вектора).

Таким образом, необходима эвристика, инвариантная относительно изменения сложности пользовательской функции. Алгоритм, основанный на запоминании прошедших сценариев не может быть инвариантным, поэтому было предложено отказаться от вектора скоростей в пользу времени выполнения.

Нулевой попыткой будем называть такое обращение к буферу, которое вернуло 0 пакетов. Временем простоя клона за такт планировщика будем называть время, затраченное на все нулевые попытки клона, произошедшие за этот такт.

Лемма 3. *Пусть период наблюдения планировщика за системой - T секунд, сумма времён простоя J клонов за такт планировщика - P секунд. Если $P > T$, то с таким же объёмом работы справятся $J-1$ клонов.*

Для стабилизации количества клонов в случае, если система перегружена, необходимо ввести скорость с большим и меньшим количеством клонов - inc и dec соответственно.

Пусть на одну нулевую попытку тратится W секунд (W можно вычислить до начала работы системы), рассматриваемый блок имеет M_j нулевых попыток за такт. Таким образом, итоговый алгоритм, осуществляющий стабильное клонирование выглядит так:

$$\begin{array}{l}\begin{cases} len(b_{in}) > MAX \\ \exists i : len(b_{out_i}) < MAX \\ \left[\begin{array}{c} inc > cur \\ inc \text{ неизвестна} \end{array}\right. \end{cases} \Rightarrow \text{ добавить } C_{j+1} \\ \left[\begin{array}{c} W * \sum(M_j) > T \\ dec > cur \end{array}\right. \Rightarrow \text{ остановить } C_j \end{array} \quad (7)$$

Полученный алгоритм через несколько итераций выходит на стабильное количество клонов.

В параграфе 2.4 рассматривается пункт 4 задачи (3) - сохранение порядка приходящих в функцию пакетов одного сетевого соединения (далее - соединения). Так как промежуточные буферы графа выполняются по принципу "первый зашёл - первый вышел", то задачу можно переформулировать как сохранение порядка пакетов каждым блоком.

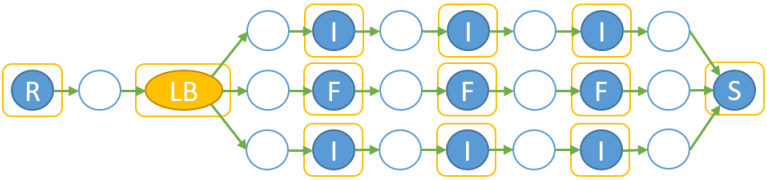
Дополним задачу утверждением: все пакеты одного соединения должны обрабатываться на одном ядре. Это важно, так как блок может осуществлять операции сборки данных более высокого уровня из отдельных пакетов более низкого уровня. Обработка части пакетов другим клоном блока будет отрицательно влиять на стабильность и производительность.

Предполагая, что пользоательские блоки работают корректно, перемешивание пакетов может произойти только при выдаче пакетов из буфера. Необходимые условия для удовлетворения сохранения порядка и выполнения на одном ядре:

$$\begin{cases} \text{У каждого буфера существует только один обработчик} \\ \text{Все пакеты одного соединения оказываются в одном буфере} \end{cases} \tag{8}$$

Назначим каждой группе соединений отдельную цепочку промежуточных буферов. Каждая пара входного и выходного буфера будет обслуживаться одним клоном. Для первичного распределения соединений по группам - цепочкам будем использовать балансировщик нагрузки, определяющий, к какому соединению относится пакет.

Клоны используют разные входные и выходные буферы. Кроме этого, разным соединениям требуются разные строки в таблицах маршрутизации. Клоны становятся независимыми. Будем называть независимые клоны блока – экземплярами блока. Граф принимает вид, показанный на рисунке 3.



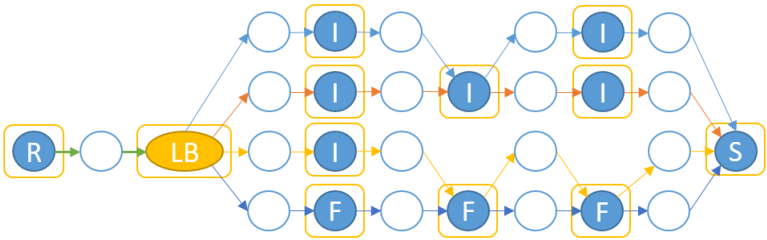
LB – балансировщик соединений, I – экземпляр блока F

Рис. 3 — Методология масштабирования, сохраняющая порядок пакетов

Условия (8) устанавливают, что у каждого буфера должен быть только один обработчик. При этом у одного обработчика может быть несколько входных буферов. В этом случае он должен выдавать пакеты в соответствующие выходные буферы, в избежании смешивания групп соединений. Таким образом, предлагается новая методология масштабирования (рисунок 3):

- Каждый блок F имеет множество входных буферов B_{in}, $|B_{in}| = E$ и множества выходных буферов B_{out}.
- Каждый экземпляр I_l блока F обрабатывает множество входных буферов $B^l_{in} \subset B_{in}$
- В избежании смешения групп соединений $\forall i \in [1 : E]$, если пакет был взят из входного буфера $b^i_{in} \in B_{in}$ он должен быть отправлен в один из выходных буферов $b^i_{out} \in B_{out}$
- Создание нового экземпляра I_k является разделением существующего экземпляра I_l. Разделение означает создание копии блока F, выделение свободного ядра для её выполнения и передача ей части входных буферов $B_k \subset B_l$, $B_l = B_l \setminus B_k$
- Удаление экземпляра I_k является объединением двух существующих экземпляров I_k и I_l. Объединение означает остановку выполнения экземпляра I_k, освобождение соответствующего ядра и объединение множеств буферов экземпляров $B_l = B_l \cup B_k$

Каждый блок запускается в единственном экземпляре, обрабатывающем все группы соединений. Со временем экземпляры добавляются и удаляются (множества обрабатываемых групп соединений делятся и объединяются). Разным блокам требуется разное количество экземпляров для удовлетворения пропускной способности. Пример графа после масштабирования показан на рисунке 4.



Различными цветами стрелок показаны пути в графе разных групп соединений

Рис. 4 — Пример графа в результате масштабирования

В параграфе 2.5 проводится адаптация алгоритма, полученного в параграфе 2.3 к новой методологии масштабирования.

Некоторые обращения экземпляра I_l к некоторым его входным буферам B^l_{in} окажутся нулевыми попытками. Обозначим количество нулевых попыток при обращении к буферу $b^i_{in} \in B^l_{in}$ как M^i_l. Тогда количеством нулевых попыток экземпляра I_l будем называть $M_l = \min_i(M^i_l)$. Экземпляр перебирает группы соединений последовательно, поэтому время простоя экземпляра является минимальным временем простоя из всех его групп соединений.

Рассмотрим ситуацию, когда I_l нельзя разделить на I_l и I_k. Это возможно, если $|B^l_{in}| = 1$. Это означает, что группа соединений требует большей пропускной способности, чем может обеспечить один экземпляр. При этом группу нельзя разделить (по крайней мере с помощью использующегося балансировщика). Такая задача может возникнуть, например, при обработке тоннельного VPN соединения. В этом случае, по желанию пользователя, предлагается вернуться к модели с клонами, которая распределит пакеты на разные ядра, но позволит поддержать пропускную способность.

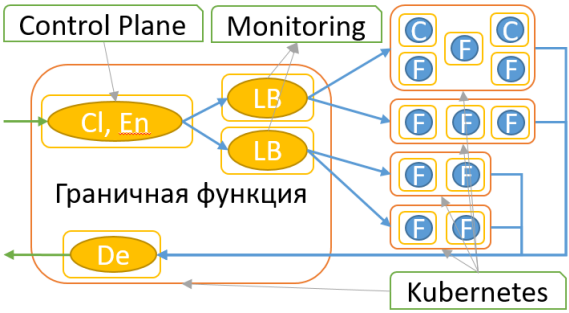
Пусть I - множество экземпляров блока F. B^l_{in} – множество входных буферов экземпляра I_l, C^l – множество клонов экземпляра I_l, M_l - время простоя экземпляра I_l. Итоговый алгоритм масштабирования, показан на листинге 1

Алгоритм 1 Алгоритм масштабирования

Через равные промежутки T, для каждого блока F
for $I_l \in I$ **do**
  if $|C^l| == 1$ **then** // Дополнительные клоны отсутствуют
    if $len(b^i_{in}) > MAX$ **then** // Один из вход. буферов переполнился
      if $|B^l_{in}| > 1$ **then** // Входной поток можно разделить
        Создать новый экземпляр I_k
        $I \leftarrow I \cup I_k$, $B^k_{in} \subset B^l_{in}$, $B^l_{in} \leftarrow B^l_{in} \setminus B^k_{in}$
      else if клоны разрешены **then**
        if $len(b^i_{out}) < MAX$ and $(inc > cur$ or $inc == 0)$ **then**
          Создать новый клон C_{new}, $C^l \leftarrow C^l \cup C_{new}$, $dec \leftarrow cur$
        end if
      else
        // Пропускная способность на одной машине невозможна
      end if
    end if
    $M \leftarrow M \cup M_l$
  else// Дополнительные клоны уже есть
    if $len(b^i_{in}) > MAX \& len(b^i_{out}) < MAX \& (inc > cur | inc == 0)$ **then**
      Создать новый клон C_{new}, $C^l \leftarrow C^l \cup C_{new}$, $dec \leftarrow cur$
    else if $W * \sum(M_{i_j}) > T$ or $dec > cur$ **then**
      Удалить клон, $C^l \leftarrow C^l \setminus C_{last}$, $inc \leftarrow cur$
    end if
  end if
end for
$A_0 \leftarrow \max(M), A_1 \leftarrow \max(M \setminus (A_0))$
if $W * (A_0 + A_1) > T$ **then** // Два экземпляра можно объединить
  Удалить экземпляр B, $B^{A_0}_{in} \leftarrow B^{A_0}_{in} \cup B^{A_1}_{in}$, $I \leftarrow I \setminus B$
end if

В параграфе 2.6 рассматривается задача развёртывания сетевых функций в облачной инфраструктуре. Предположим, что в результате масштабирования все ресурсы одной машины оказались заняты. Для дальнейшего масштабирования необходим перенос задачи в многомашинное окружение.

Вход и выход из созданной функциональности предлагается реализовывать с помощью граничной функции, распределяющей пакеты по обрабатывающей аппаратуре, на которой исполнятся цепочки необходимых микросервисов под управлением, например, системы управления контейнерами - Kubernetes - рисунок 5.



Cl – классификация, En – инкапсуляция, De – декапсуляция.
Рис. 5 — Предлагаемое размещение сетевых функций

Каждая цепочка исполняется до конца на своей машине ("Run to completion"), что означает однократную коммутацию пакета. Граничная функция программно конфигурируема извне в соответствии с принципами программно-определяемой сети.

Задача граничной функции – распределить сетевые соединения на обрабатывающую аппаратуру, учитывая обратную связь для балансировки нагрузки. Кроме того, граничная функция берёт на себя задачи по конфигурированию необходимой обработки. Это означает, что каждое сетевое соединение будет послано на машину, на которой уже сконфигурирован необходимый конкретно этому соединению граф микросервисов.

В третьей главе диссертационной работы реализуется программная система, использующая предложенные модели и алгоритмы.

В параграфе 3.1 даётся общее описание созданной системы NFF-GO, обосновывается выбор языка программирования Go и низкоуровневого фреймворка DPDK.

В параграфе 3.2 рассказывается об использовании модели графа (1). NFF-GO предоставляет набор блоков входа пакетов v_{in}: “Receive”, “Read”, “Generate“; выхода v_{out}: “Send”, “Stop”, “Write”; блоков построения графа: “Partition”, “Copy”, “Merge”, “Split”, “Separate” и блока обработки пакетов “Handle” (рисунок 6). Каждый блок допускает пользовательскую

настройку, а блоки “Handle”, “Split”, “Separate”, “Generate” включают в себя вызов созданных пользователем алгоритмов, которым передаётся пакет (или вектор пакетов) в виде высокоуровневого представления с возможностью дальнейшего разбора.

Рис. 6 — Пример графа обработки пакетов, построенного во NFF-GO

В параграфе 3.3 рассматриваются вопросы связи DPDK фреймворка, использующего язык C с остальной частью программного решения, использующей язык Go. Пакеты хранятся в DPDK контролируемой C памяти, над ними выполняются низкоуровневые DPDK функции receive, send, stop. Данные функции вызываются из управляющего Go кода при инициализации и далее используются по мере получения пакетов. Высокоуровневая Go часть пользуется указателями на пакеты без непосредственного вызова C функций и копирования пакетов - рисунок 7.

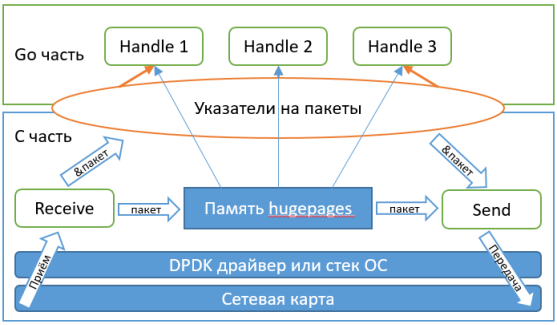


Рис. 7 — Схема взаимодействия C и GO

В параграфе 3.4 рассматривается высокоуровневое представление пакета, находящееся перед DPDK структурой пакета, хранящее указатели на заголовки поддерживаемых протоколов. Пользователь инициализирует указатели по мере необходимости и разбора пакета.

В параграфе 3.5 приводится исходный код примера, созданного на NFF-GO. Пример принимает пакеты и распределяет их по выходным портам в зависимости от их протоколов. Исходный код примера занимает 18 значащих строк кода.

В параграфе 3.6 рассматривается практическая реализация алгоритмов масштабирования. Было решено отказаться от отдельного балансировщика, который был заменён на возможности сетевой карты по масштабированию на стороне приёма (RSS) - рисунок 8.

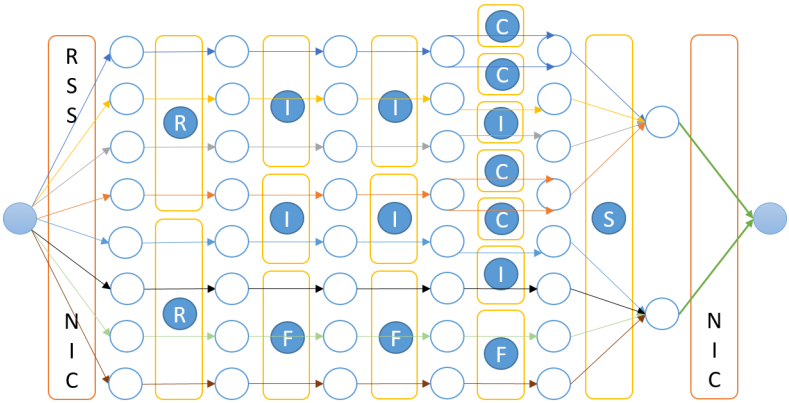


Рис. 8 — Схема масштабирования фреймворка NFF-GO

В параграфе 3.7 обсуждаются локальные оптимизации предоставляемых блоков. Способы масштабирования Generate, реализация Merge и Stop, эффективное использование Send.

В параграфе 3.8 обсуждаются нерешённые задачи, такие как учёт NUMA архитектуры, обработка потока байт вместо пакетов, обеспечение отказоустойчивости.

В четвёртой главе диссертационной работы приводятся экспериментальные результаты, полученные с использованием созданной системы NFF-GO. Рассматриваются следующие примеры:

В качестве вычислительно сложного примера, демонстрирующего малые накладные расходы, приводится реализации тоннельного режима семейства протоколов IP безопасности - IPSec.

- Каждый байт пакета изменяется во время шифрования, что является нагрузкой на кэш
- Длина пакета изменяется во время инкапсуляции и декапсуляции пакета
- Операции шифрования и аутентификации являются сложными, что требует масштабирования соответствующих частей функции
- Garbage Collector будет загружен, так как операция шифрования является операцией с расходованием памяти, которую необходимо освобождать

Как видно из рисунка 9, созданная реализация не уступает низкоуровневой реализации DPDK.

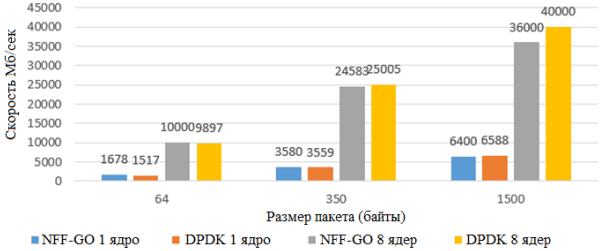


Рис. 9 — Сравнение NFF-GO и DPDK на примере IPSec

В качестве примера более легковесной функциональности (анализируются и изменяются только заголовки пакетов) используется трансляция сетевых адресов - NAT. Функциональность NAT поддерживается ОС Linux и не реализована как пример DPDK, поэтому NFF-GO сравнивается с реализацией Linux - рисунок 10.

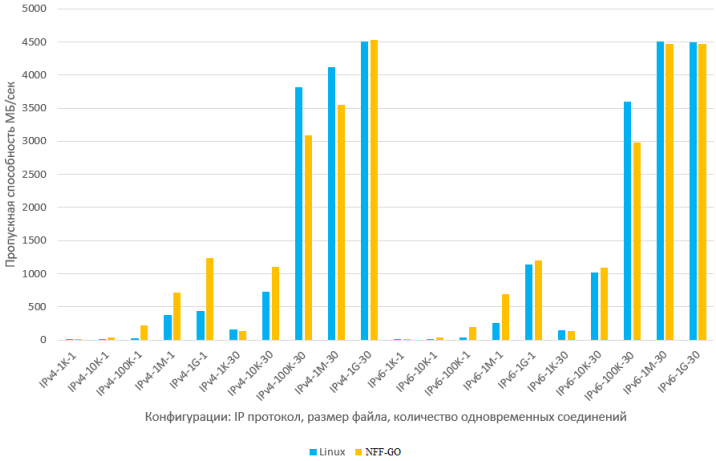


Рис. 10 — Сравнение NFF-GO и Linux на примере NAT

NAT представляет собой полностью готовый пример, учитывающий одновременные соединения, ARP и ICMP запросы, что показывает применимость NFF-GO для решения индустриально значимых задач.

Другим индустриальным примером является реализация пакетно-сервисного шлюза в архитектуре "Эволюция системной архитектуры" - SAE для стандарта передачи данных 5G. Данный пример занимает в десять раз меньшее число строк кода, по сравнению с DPDK версией, и не уступает в производительности.

Возможности масштабирования анализируются на модельных примерах. В первом примере представлен граф, состоящий из одной функции,

которая может динамически менять свою сложность (например, в отдельной потоке - go routine - читается файл с задачами, который изменяется извне приложения или, другой пример, в зависимости от времени в функцию приходят пакеты разной длины, на которые тратится разное количество времени). Кроме сложности функции будем динамически изменять интенсивность входящего потока. У наблюдаемой функции есть две характеристики - интенсивность исходящего потока и количество клонов (ядер CPU), которое тратится для того, чтобы достичь подобной интенсивности - рисунок 11.

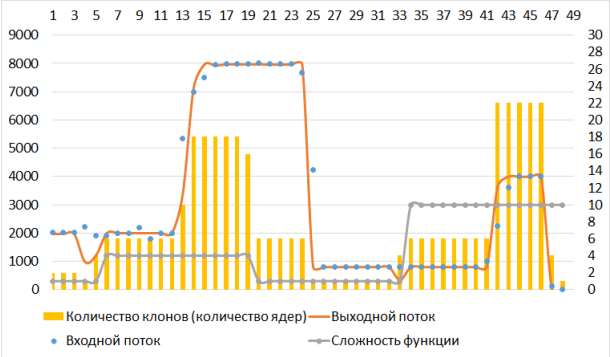


Рис. 11 — Масштабирование NFF-GO

Во втором примере рассматривается поведение экземпляров функции, обрабатывающей восемь сетевых соединений, их разбиения и соединения по экземплярам в зависимости от интенсивности каждого соединения - рисунок 12.

В нижней части графика приведена интенсивность каждого сетевого потока (по вертикали количество Мбит/сек), которые складываются в общую интенсивность входного потока. В верхней части графика приведены номера ядер (по вертикали), которые обрабатывают каждый из сетевых потоков. По горизонтали для обоих графиков отображены моменты времени.

В заключении сформулированы основные результаты работы:

1. Предложена высокопроизводительная реализация модели представления сетевой функции на основе графа предопределённых блоков. Промежуточные буферы являются кольцевыми буферами. Буферы предоставляют доступ в векторном режиме и одновременные операции чтения и записи без задержек. Блоки выполняются на отдельных ядрах в режиме постоянного опрашивания.
2. Разработана модель блочного масштабирования сетевой функции для удовлетворения требования пропускной способности с помощью клонирования на свободные ядра процессора. Исследуется и

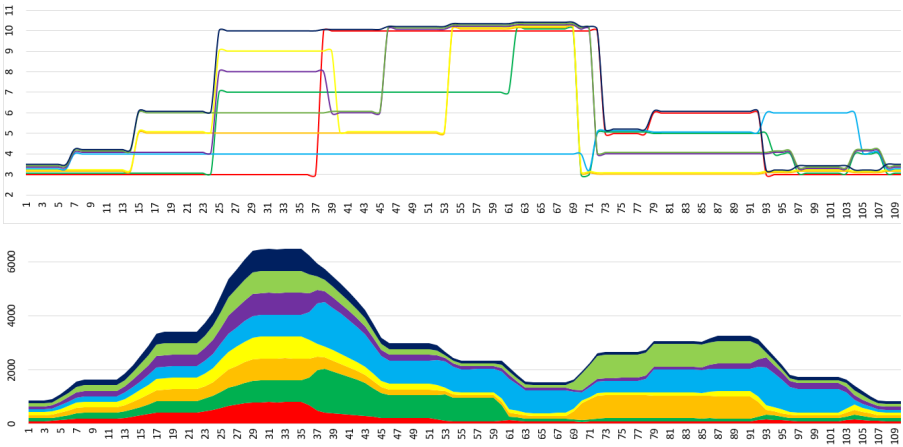


Рис. 12 — Масштабирование NFF-GO (экземпляры)

сравниваются два варианта модели. Первый вариант допускает обработку пакетов одного сетевого соединения на разных ядрах и изменение порядка пакетов. Второй вариант предварительно использует балансировщик нагрузки для распределения пакетов по сетевым соединениям. Рассмотрены эвристики для автоматического определения моментов времени для динамического увеличения или уменьшения количества клонов в условиях недостаточных знаний о поведении функций и потоков пакетов. Показано, что эвристики о количестве пакетов во входном буфере недостаточно, а эвристика о скорости обработки пакетов неустойчива относительно изменения производительности функции под влиянием внутренних или внешних причини. Предложена достаточная и стабильная эвристика, основанная на времени простоя функции, связанного с отсутствием пакетов для обработки.

3. Разработана схема развёртывания сетевой функции в условиях облачной инфраструктуры. Связь группы машин с внешней сетью осуществляет граничная функция, она же распределяет пакеты по требуемым графам обработки, находящимся на других машинах, и многомашинное масштабирование. Все задачи для одного пакета выполняются последовательно на одной машине в виде графа, что означает однократную коммутацию и отсутствие накладных пересылок пакетов. В результате, в совокупности с алгоритмом масштабирования по ядрам процессора, предлагается многоуровневое расщепление любого входного потока на неделимо мелкие части, что является шагом к обработке сетевых потоков неограниченной интенсивности.

4. Проведено сравнение методов создания и развёртывания сетевых функций. Представлены методы построения сетевых функций в закрытом коде, с использованием ядра операционной системы, с использованием специализированных средств программно-аппаратного ускорения, с использованием высокоуровневых фреймворков. Выделены этапы представления сетевых функций как специализированное устройство, стандартный сервер, виртуальная машина и контейнер.
5. На основе проведённого исследования реализована программная система NFF-GO, написанная на языках C и GO с использованием библиотеки DPDK. С использованием системы созданы сетевые функции: IP security, трансляция сетевых адресов, пакетный шлюз и другие. Система внедрена в корпорации производителе электроники Intel и некоторых корпорациях – интернет провайдерах.

Публикации автора по теме диссертации

1. *Philippov, I.* Novel Approach to Network Function Development / I. Philippov, A. Melik-Adamyan // Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia. — St. Petersburg, Russia : ACM, 2017. — 17:1—17:6. — (CEE-SECR '17). — URL: http://doi.acm.org/10.1145/3166094.3166111.
2. *Philippov, I.* Building Heuristic Scheduler for One-Machine Network Function Scaling / I. Philippov // Proceedings Fifth International Conference on Engineering and Telecommunication - EnT-MIPT 2018. — Moscow, Russia : IEEE, Принята к публикации 2019. — С. 49—54.
3. *Филиппов, И. В.* Размещение сетевых функций в облачных инфраструктурах / И. В. Филиппов, А. Ф. Мелик-Адамян, В. А. Сухомлинов // Информационные технологии. — 2019. — Апр. — № 4. — С. 223—228.
4. *Филиппов, И. В.* Новый подход к созданию сетевых функций / И. В. Филиппов, А. Ф. Мелик-Адамян // Труды 60-й Всероссийской научной конференции МФТИ. — Долгопрудный, 2017. — С. 44—45.
5. *Филиппов, И. В.* Использование, возможности и задачи технологии SDI / И. В. Филиппов, А. Ф. Мелик-Адамян // Тезисы 58-й научной конференции МФТИ. — Долгопрудный, 2015. — URL: http://conf58.mipt.ru/static/reports_pdf/402.pdf.

Список литературы

1. *ETSI.* "Network Function Virtualization" white paper / ETSI. — 2012. — URL: https://portal.etsi.org/nfv/nfv_white_paper.pdf.