TÉCNICO LISBOA

Parallel NF Synthesis

Francisco Manuel Chamiça Pereira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís David Figueiredo Mascarenhas Moreira Pedrosa

Examination Committee

Chairperson: Prof. Francisco António Chaves Saraiva de Melo
Supervisor: Prof. Luís David Figueiredo Mascarenhas Moreira Pedrosa
Member of the Committee: Prof. Luís Jorge Brás Monteiro Guerra e Silva

January 2021

Acknowledgments

First and foremost, I would like to thank my supervisor, Prof. Luís Pedrosa, for its outstanding supervision.

I would also like to thank Inês Lopes, my partner in life, for continuously providing me a stable source of happiness.

I would also like to thank Prof. Fernando Ramos, Prof. Justine Sherry, and Dr. Rúben Martins for their feedback. I consider myself very lucky to have had the opportunity of receiving it.

Last, but not least, I would like to thank my family for keeping me motivated and continuously providing me the means to achieve higher education, and my friends for all their support. Specially Shams Valibhai, for our long, fun, and interesting conversations.

Thank you.

Abstract

Dedicated network appliances running network functions (*e.g.* routers and firewalls) are associated with high infrastructure and management costs. In the beginning of this decade, network function virtualization (NFV) was proposed as a solution. It promised lower costs, easier management and increased development and deployment flexibility by decoupling the hardware from the network function itself. However, the performance of these software network functions needed to be competitive with the aforementioned dedicated appliances. New techniques of packet acquisition and processing were studied in order to achieve higher bandwidth and lower latency. High performance packet forwarding frameworks started becoming a reality. Moreover, parallelization started to be considered in the scope of network functions as a technique to improve performance and take advantage of the now common multi-core CPU architectures.

Alongside with Receive Side Scaling (RSS), a technique that allows packets to be forwarded to a specific CPU core depending on their fields, parallelism proved to be effective by increasing throughput. However, data synchronization, commonly associated with parallelism, takes its toll on performance. It is, thus, important to avoid data sharing between instances running on different CPU cores. Such sharing can be minimized or even eliminated by understanding how the network function maintains per-flow state and configuring Receive Side Scaling accordingly, which will dictate which packets should be processed on which CPU cores. Until now, this was a manual process, but we propose a framework that analyzes a network function's sequential implementation and provides a configuration for efficient parallel implementations that minimize data sharing.

Keywords

Network Functions; Receive Side Scaling; Parallelization; Software Synthesis.

Resumo

Dispositivos de rede dedicados são associados a elevados custos de infraestrutura e gestão. No início desta década, foi proposta como solução a virtualização de funções de rede (NFV) que, através do desacoplamento entre o *hardware* e a função de rede, prometia custos mais baixos, maior facilidade de gestão e maior flexibilidade. Porém, o seu desempenho deve ser competitivo relativamente ao dos aparelhos dedicados, o que levou ao estudo de novas técnicas de aquisição e processamento de pacotes, e à consideração de paralelização no âmbito de funções de rede.

O *Receive Side Scaling* (RSS) é uma técnica que distribui pacotes, de acordo com os seus campos, por núcleos do CPU. A paralelização de funções de rede, juntamente com o RSS, provou-se eficaz, reflectindo-se no aumento do desempenho. Contudo, a sincronização de dados, frequentemente associada com o paralelismo, pode acarretar um custo de desempenho. É, portanto, importante evitar a partilha de dados entre instâncias paralelas. Esta partilha consegue ser minimizada (ou até eliminada) se percebermos como é que uma função de rede mantém o seu estado. Esta compreensão leva-nos a configurar o RSS adequadamente, que irá ditar que pacotes serão processados em que núcleos. Até agora, este processo era inteiramente manual. Nós propomos uma ferramenta que analisa a implementação sequencial da função de rede e faculta uma configuração de RSS que leva a implementações paralelas eficientes.

Palavras Chave

Funções de Rede; *Receive Side Scaling*; Paralelização; Síntese de software.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
DLNA	Digital Living Network Alliance
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
GPU	Graphics Processing Unit
IDS	Intrusion Detection System
LAN	Local Area Network
LVA	LibVig Access
NAT	Network Address Translator
NF	Network Function
NFV	Network Function Virtualization
NIC	Network Interface Card
NIDS	Network Intrusion Detection System
NUMA	Non-Uniform Memory Access
PCIe	Peripheral Component Interconnect express
RSS	Receive Side Scaling
RX	Receive (typically refering to a queue)
SDN	Software Defined Networks

STA Statistical Traffic Analyzer

TCP Transmission Control Protocol

TLS Transport Layer Security

TX Transmit (typically refering to a queue)

VM Virtual Machine

VPN Virtual Private Network

WAN Wide Area Network

1

Introduction

Contents

Since the seed of software network functions (NFs) was planted, research has been done to uncover a collection of techniques that enable processing packets entirely in software, allowing commodity servers to replace inflexible network equipment.

Although processing packets at line-rate has been a challenge, these techniques have enabled packet processing at throughputs of 10s of Gbps and latencies of 100s of ns. Developing NFs with this kind of performance can be quite challenging though, as every nanosecond counts at this scale.

One technique to improve throughput is Receive Side Scaling (RSS), which spreads packet processing across multiple CPU cores. The NIC is configured to hash a given set of fields of the packet and use the hash value to select the CPU core that should process the packet. Getting this right in practice is a challenge however. Once cores contend for access to shared state, the overhead of ensuring consistent access to memory, both at the application level (locks or lock-free data structures) and the CPU cache (cache coherence), often outweighs the gains of parallelism. A naive parallel implementation can end up being slower than its sequential counterpart.

Recent work in NF verification, as discussed in the next chapter, explored the use of symbolic analysis to build a sound and complete representation of the entire behavior of an NF. In that case, this representation was used to check the behavior against a specification for correctness but, in this thesis, we use it to synthesize a parallel version of it using RSS. This raises three basic challenges: how to preserve sequential semantics across packets, how to generate code that runs efficiently and actually benefits from the parallelism, and how to balance the gains of assigning more or fewer cores to the NF.

1.1 Main contribution

With this goal in mind, we created Maestro, a static analysis tool that automatically parallelizes an NF following a shared-nothing architecture. However, that might not always be possible. Some NFs preserve state in such a way that makes them impossible to parallelize without the use of locking mechanisms. As such, Maestro is also able to synthesize locks that allow the NF's semantics to be preserved, if it is deemed necessary.

Our contribution to the NFV state of the art is, therefore, a tool that allows for automatic generation of parallel implementations of NFs. Although Maestro was built as a push-to-parallelize tool in the Vigor framework, we provide all the knowledge required to port its functionality to other frameworks. Vigor was the chosen playground, as it already gives the complete representation of the sequential NF for free as a byproduct of its verification process. Because the process of maintaining semantics from the sequential to the parallel implementations is associated with choosing the right RSS configuration, we developed a library independent of Maestro that finds the RSS configurations that allow this to be done.

1.2 Organization

The remainder of this dissertation is divided into 4 chapters.

The following chapter presents an overview of the current state of the art of the Network Function Virtualization field. It begins with an introduction on how and why this field was born. Then, it shows the major advances in packet I/O frameworks that allowed for fast packet processing in userspace, followed by an exposition of the parallelization challenges and solutions in this field, and some examples of parallel NFs. Finally, we present the work in NF verification that allowed us to build Maestro.

Chapter 3 explains how Maestro automatically generates parallel implementations of NFs. It begins by exposing how one can achieve a shared-nothing parallel implementation of an NF via partitioning its state and carefully orchestrating packets to the available cores (hence the name Maestro). Subsequently, an overview of Maestro's architecture is shown, to which follows one section per module in that architecture explaining the details associated with it.

Chapter 4 contains the evaluation of our proposed tool. It encompasses measures of time taken to generate parallel implementations, example-driven explanations of such generations, and performance evaluations of the newly created parallel NFs. In this chapter we can also find performance variation results regarding configurations on NUMA architectures and different traffic distributions.

Finally, chapter 5 shares the conclusions taken from this work.

2

Related work

Contents

In this chapter, we present an overview over the state of the art of the field of Network Function Virtualization—where this thesis primarily resides. After an introduction of how this field came to life follows the main challenges associated with it. Then, we explore some of the techniques discovered to counter those challenges, and follow with examples of real world applications. We close this chapter by exploring an example of how software verification was adopted into the field of Network Function Virtualization, and how we use that example in this thesis.

2.1 Network function virtualization

Network services are commonly associated with physical, proprietary, and closed-source middleboxes. These middleboxes implement a wide range of functionality, and are responsible for more than only forwarding packets. They can address security (*e.g.* firewalls and Intrusion Detection Systems), performance (*e.g.* WAN optimizers), and support for new applications and protocols (*e.g.* TLS proxies) [4].

Telecommunication companies provide these services, fulfilling requirements for high-quality, stability, and strict protocol adherence with specialized hardware. This requires dense deployments of network equipment. Sherry *et al.* [5] found, upon studying 57 enterprise networks, that the typical number of middleboxes in an enterprise is comparable to its L2/L3 infrastructure—*e.g.* switches and routers. However, these middleboxes are associated with high infrastructure and management costs, which result from their complex and specialized processing, variations in management tools across devices and vendors, and the need to consider policy interactions between these appliances and other network infrastructure. Additionally, as users continue to increase requirements for newer and more diverse services with higher data rates, the telecommunication service providers must continually purchase, store and operate new physical equipment. Ultimately, this leads to an increase of capital and operating expenses.

Sherry *et al.* proposed offloading packets to what they called *middlebox service providers* running in the cloud as a way to counteract the expenses and inertia associated with these middleboxes. This lead to the proposal of network function virtualization (NFV), in 2012, as a solution to the aforementioned problems [6]. By the end of that year, the formation of the Industry Specification Group for Network Functions Virtualization (ISG NFV) [7]—under the auspices of the European Telecommunications Standards Institute (ETSI)—was announced. Its formation joined multiple telecommunication service providers and IT vendors in a collaboration setup to tackle the challenges of network function virtualization. Although it is not a standards development organization, it seeks to define the requirements that network operators may adopt and tailor for their commercial deployment.

NFV promised to build more dynamic and service-aware networks, reduce product cycles, operating and capital expenses, and improve service agility [8]. This could be achieved by decoupling the physical network equipment from the functions that run in them. The idea was to build software instances of network functions and run them on a virtualized environment using commodity hardware instead of the

typical network device.

Although the concepts of Software Defined Networks (SDN) and NFV are related, it is important to mark the distinction between them. In an SDN, according to the Open Networking Foundation, *the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications* [9]. In the same way that NFV aims at running network functions on commodity hardware, the SDN control plane can be implemented entirely in software an run on top of the same hardware (but still using dedicated hardware for the data plane). In fact, they can be used in conjunction. However, contrary to NFV, SDN focus on separating the handling of packets and connections from overall network control. On the other hand, NFV aims to separate network functions from dedicated hardware [8], going even further than SDNs and incorporating both the data and the control planes entirely on software. That is, SDN implements the control plane in software and leaves the data plane in hardware, and NFV implements both planes in software.

Decoupling network functions from their dedicated hardware promised significant advantages [10]:

1. Independent development timelines for software and hardware.
2. Different network functions could share resources on the same hardware, and the same hardware could be used for different network functions at various times.
3. Improved flexibility in assigning NFs to hardware, improving scalability in a more dynamic way and with finer granularity.
4. Accelerated service innovation.

Given the flexibility associated with designing a network function entirely in software, a large number of traditional functions where considered to be virtualizable (*e.g.* DHCP servers, NATs, routers, firewalls, IDSs, VPN servers, and DLNA servers) [11].

The NFV proposal relied on virtualization—virtual machines (VMs) and containers—to ensure isolation between network functions. This provided safety—NFs from different vendors could safely and independently run on the same hardware, as they would access isolated memory—and allowed for some degree of performance isolation, essential for serving multiple customers. However, as noticed by Panda *et al.* [4], this virtualization was found to incur substantial performance overheads for simple NFs. In their study, they showed that it was possible to achieve memory and packet isolation without the performance penalty associated with virtualization. Consequently, although the term *network function virtualization* is still in use today, it is not necessarily associated with virtualization. Furthermore, in order to better make the distinction, the term *software network functions* has become more commonplace.

Sherry *et al.* [5], upon their study on the costs associated with middlebox deployments and the potential benefits of outsourcing them to a cloud-based infrastructure, examined multiple options for architecting cloud-based middlebox services. According to the authors, cloud-based NFV solutions would not only come with the already mentioned benefits, but would also make this service available to users

who could not otherwise afford the associated costs and complexity of an in-house IT infrastructure—*e.g.* small businesses, and home and mobile users. However, such an architecture would have to meet three challenges in order to be considered viable:

1. Provide functionality and semantics equivalent to that of an on-site middlebox.
2. Minimize the complexity of enterprise-side functionality.
3. Minimize the performance penalty associated with detouring packets to the cloud.

A few years later, on their NFV state of the art study, Mijumbi *et al.* [8] extended the number of challenges associated with making it an acceptable solution for telecommunication service providers:

1. NFV architectures should be able to achieve performance similar to that obtained from functions running on dedicated hardware.
2. Functions or services from different subscribers should be protected/isolated from each other.
3. The NFV infrastructure (physical and virtual resources) should be protected from the delivered subscriber services.
4. Multiple availability classes may be defined which should be supported by an NFV framework.
5. Any acceptable NFV platform must be an open, shared environment capable of running applications from different vendors.
6. Support for both physical and virtual NFs is important for operators making the transition to NFV as they may need to manage legacy physical assets alongside virtualized functions for some time.
7. NFV needs to be acceptably scalable to be able to support millions of subscribers.

In this thesis, we focus on the first challenge—performance improvements of network functions, such as latency and throughput. For many years, running network functions on dedicated middleboxes brought significant performance advantages when compared to using commodity hardware. Low-level packet processing using traditional tools like raw sockets were not providing the performance required when using links of Gbps [12].

2.2 Packet I/O frameworks

In order to address the mentioned performance challenges, new packet I/O frameworks were proposed. These high-performance userspace frameworks, typically responsible for communicating with the network interface card (NIC), acquiring packets and delivering them to the application, can implement a series of features that enable them to achieve high packet delivery/processing rates [13]:

Zero-copy Typically, in order to receive and send packets to and from a network interface card (NIC), an application makes read and write system calls in userspace buffers, which copy the content of kernel-space buffers configured with direct memory access (DMA). DMA is also later used by the NIC to read the packets from this space and place them in its transmit queues. However, memory

allocation for packets is a significant overhead in the Linux kernel [1]. To mitigate this, a technique called zero-copy is used, wherein the NIC directly reads and writes to buffers visible to both the application and the NIC, and copies between kernel and userspace are avoided. In conjunction with a careful API design to avoid copying within the application, the CPU intervention in the process of reading and writing packets to and from a NIC is minimized.

Kernel bypass Modern operating systems feature a complete network stack that is in charge of providing an interface for sending and receiving data and handling a wide variety of protocols and hardware. However, this interface does not perform optimally when trying to capture traffic at high speed (the Linux network stack starts dropping packets when receiving at a rate of Gbps [14]). One of the reasons behind this is that each system call entails a context switch, from user-level to kernel-level and vice versa. Such system calls and context switches may consume thousands of CPU cycles per-packet, which can be avoided by bypassing the kernel altogether [15]. Frameworks such as DPDK [16] implement their own networking subsystems, entirely separate from the kernel and go as far as implementing userspace NIC drivers. This allows userspace programs to interface directly with the NIC, without any need for kernel intervention.

I/O batching Instead of continuously polling the NIC for every single new incoming packet, it has become common for network drivers to poll groups of incoming packets. This process is called *batching*, and reduces the number of PCIe register access and status writeback from NICs [2]. It also improves the temporal locality of memory access, reducing the number of cache misses [17].

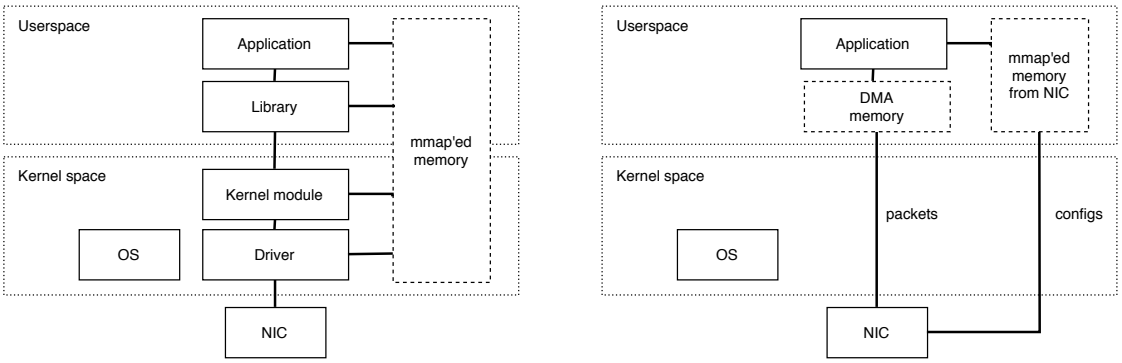
Hardware multi-queue support Most modern NICs are equipped with multiple receive and transmit queues. This feature can be used not only for virtualization purposes, but also to distribute packets from the receive queues to different CPU cores. One can associate each CPU core with one of the available multiple queues, and carefully distribute the incoming traffic across multiple cores to perform parallel processing of packets. Moreover, as will be later addresses in more detail, this packet orchestration can be used to develop parallel NFs whose instances have no need to communicate with each other or be synchronized.

Emmerich *et al.* [1] categorized these frameworks into 2 groups: those relying on a driver running in the kernel (fig. 2.1(a)), and others that completely bypass the kernel and the entire network stack, re-implemented drivers running in userspace (fig. 2.1(b)).

2.2.1 Frameworks relying on a kernel driver

The packet I/O frameworks relying on a driver running in the kernel use both the default driver and an additional kernel component that provides a fast interface for the userspace application. We provide here a list of examples of such frameworks.

In 1993, McCanne and Jacobson presented the Berkeley Packet Filter (BPF) [18], a kernel architec-



(a) Architecture using in-kernel driver

(b) Architecture using full userspace driver

Figure 2.1: The 2 architectures of userspace packet processing frameworks mentioned by Emmerich *et al.* [1].

ture for packet capturing and filtering. It used a register-based filter running user defined code in a virtual machine in kernel-space. It proved to be general and extensible—as the filter machine was not protocol dependent—and portable, running on most BSD (and derivatives) systems. Later, an extended version of BPF was proposed (eBPF) [19], improving its performance when running on modern hardware. eBPF was incorporated in the Linux kernel, and later exposed to userspace. By this point, eBPF became unbounded to not only filtering purposes, as users could use it to freely develop packet processing NFs.

In 2004, Deri presented PF_RING [20], a device polling packet capture kernel module that performs close to Gbps line-rate. PF_RING allocates ring buffers in kernel-space, where the packets are inserted. Later, these packets are accessed by the userspace application through a shared memory region.

Netmap [12], published in 2012, is a standard component in FreeBSD and also available on Linux that provides fast packet access to userspace applications. It achieves high performance by implementing zero-copy, batched I/O, and multi-queue support.

Finally, in 2016, PFQ (Packet Family Queue) [17] was presented as another high-performance packet I/O framework for fast packet capturing, processing and distribution. It is an open-source Linux kernel module and is able to process packets at line-rate on 10Gbps links.

2.2.2 Userspace-driver frameworks

As mentioned, all these frameworks relied on a component running on the kernel. On the other hand, Intel developed, in 2010, the Intel Data Plane Development Kit (DPDK) [16], a framework that not only completely bypassed the kernel, but provided multi-core and NUMA aware functionalities. DPDK is a framework that provides a set of software libraries and drivers for fast packet processing. These libraries offer many functionality, including handling memory allocation for network packets, buffering

packet descriptors for ring-like structures, and passing packets between the NIC and the application. It offers two models: pipeline, where one core is taking packets from the NIC, passing to another core for processing, passing it finally to yet another for packet transmission handling; and run-to-completion, where the packets are distributed among the cores, and each packet is processed entirely on the one core. It is in wide use and many applications already use these libraries to achieve high performance [21–24].

Some authors go even further and develop a driver made solely for the purpose of demystifying userspace drivers. Emmerich *et al.* noticed that the majority of developers building high-speed network applications still treat userspace drivers as blackboxes, when in reality frameworks like DPDK and netmap made them more accessible [1]. In order to mitigate this issue, as knowing the details of these drivers can improve the application's performance, the authors built *ixy*, a userspace network driver designed for simplicity and educational purposes.

While undergoing an explanation of how various network packet processing frameworks work and how they built *ixy*, the authors mention the importance of batching for increasing the performance. Receiving and transmitting packets requires information regarding queue index registers (RX and TX queue, respectively). This information must be accessed on the NIC, requiring a trip on the PCIe bus, a costly operation. Batching mitigates this issue at the cost of trashing the L1 cache. The batch size must be, therefore, carefully chosen.

Queue sizes are also important, as the NIC must be able to receive and send bursts of packets without overflowing its rings (leading to packet losses). However, having too large buffers can be detrimental, as they lead to an increase in latency.

The authors also tested multiple NUMA configurations. Under a NUMA architecture, a PCIe device (*e.g.* the NIC) can only be attached to a single NUMA node. Moreover, the thread processing the packets can only run on a single NUMA node, and the developer is free to choose the node which will contain the memory used for storing packet buffers by this thread. The authors studied the performance variation under the eight possible scenarios that arise from the combinations of these configuration choices. They arrived at the conclusion that pinning the memory, but not the process itself, to the wrong NUMA node (*i.e.*, the one not connected to the NIC) does not take a toll on performance. Forwarding between 2 different nodes is fastest when the memory is pinned to the egress nodes and the CPU to ingress nodes, and slowest when both are pinned to ingress nodes. The authors come to the conclusion that the best choice is to pin the thread to the ingress NIC and distribute packets buffers across the nodes.

Many of the lessons learned in each of these approaches have informed design choices in this thesis. Several of the performance trade-offs described here are also further explored in the evaluation chapter.

2.3 Taking advantage of multi-core CPU architectures

Until the beginning of this millennium, CPU designers achieved performance gains on processors by increasing clock speeds, optimizing execution flows and increasing cache sizes [25]. This led most applications to enjoy regular performance gains without even releasing new versions. However, because of technological issues like power consumption, heat dissipation and current leakage, this steady and regular CPU performance growth came to a halt. Because of this, the major processor manufacturers (*e.g.* Intel and AMD) turned to simultaneous multithreading—SMT, allowing multiple threads to run in parallel in the same core—and chip multi processors—CMP, running threads in different CPU cores in parallel.

Since the transition of network functions from dedicated appliances to commodity hardware, increasing a network function's performance was made possible not only by tweaking its software implementation, but also by upgrading the hardware. Taking advantage of the multi-core CPU architectures, capable of executing multiple threads at the same time, could bring significant improvements to an NF's performance.

Much like the use of multiple threads to serve concurrent client requests made to a server, the performance increase of parallelizing a network function is translated to more packets being processed in the same amount of time. This can be viewed as data parallelism, *i.e.*, parallel processing different data elements (packets). The key idea is to spawn parallel instances of a network function and distribute packets between them, this way allowing for parallel (and as independently as possible) processing of packets. The *independent* processing of packets on different cores (later referenced as shared-nothing model) is the key concept here, and will have critical importance to this thesis.

Parallelization aims to reduce the time taken to solve a computational problem by taking full advantage of multi-core and/or multi-processor architectures. However, if not executed correctly, a parallel implementation can decrease performance when compared to its corresponding sequential implementation. As such, parallelizing implies careful considerations on how one converts the sequential implementation into multiple parallel tasks without compromising the original program's semantics whilst maximizing the performance. Studying data dependencies within the code as well as managing concurrent accesses to shared memory are also now a new concern.

In order to aid the developer on this task, tools such as OpenMP [26], a set of compiler directives and callable runtime library routines created to allow shared-memory parallelism were created. OpenMP, specifically, allowed for facilitated loop-level and coarse-grained parallelism. However, when acquiring and processing tens of millions of packets per second, data synchronization becomes a problem. Barriers and locking mechanisms are associated with high performance tolls, and can lead to increased packet processing time on an NF's execution, which itself can lead to packet drops. Moreover, not only is loop-level parallelism not relevant in the context of NFs, but for taking advantage of OpenMP, batching

large amounts of packets for later processing would be required. This is impractical, as performance increase of network functions is commonly associated with trimming tens of nanoseconds of packet processing time, and therefore the packets should be dealt with as fast as possible

Vesović *et al.* [27] created a parallel IP lookup implementation in userspace with OpenMP. IP lookup tables are read far more often than they are updated, and therefore changes to this shared state on different instances of the NF are infrequent. The authors claim that using OpenMP does not bring higher performance tolls when compared to a parallel implementation using pthreads. This NF, however, represents an optimal case for using OpenMP, and these results may not generalize for other scenarios.

2.4 Parallelization challenges and proposed solutions

2.4.1 Parallel NF's pitfalls

Fusco and Luca [28] studied common pitfalls of network monitoring applications in the context of multi-core systems. Not only that, they also designed a multi-core aware packet capture kernel module, later used to build a network monitoring system capable of processing more than 4 Gbps per physical core. They gave 3 reasons for packet capture performance to drop when running on multiple threads or when using multi-queue adapters:

1. Resource competition of threads on the NICs' RX queues.
2. Unnecessary packet copies.
3. Improper scheduling and interrupt balancing.

The authors tackled these challenges by using virtual capture devices, multi-threaded polling, and zero-copy mechanisms. Implemented in PF_RING, each virtual capture device is associated with one RX queue, and RSS is used to distribute packets across the queues. They combined the PF_RING zero-copy ring buffers for each queue with a driver—TNAPI— that fetches packets using multi-threaded packet polling. Moreover, it does not deliver packets to the network stack, therefore avoiding the kernel overhead.

2.4.2 Shared-nothing model

Papadogiannaki *et al.* [2] designed a scheduler tailored for network packet processing applications that takes into consideration the heterogeneity of the commodity hardware (CPU and GPU), the applications, and the incoming traffic. The scheduler, given a specific network packet processing related application, utilizes the most efficient device or group of devices.

The authors considered two different architecture models for capturing network traffic and distributing it to different devices for processing. On one hand, a master-worker model (fig. 2.2(a)), where the master

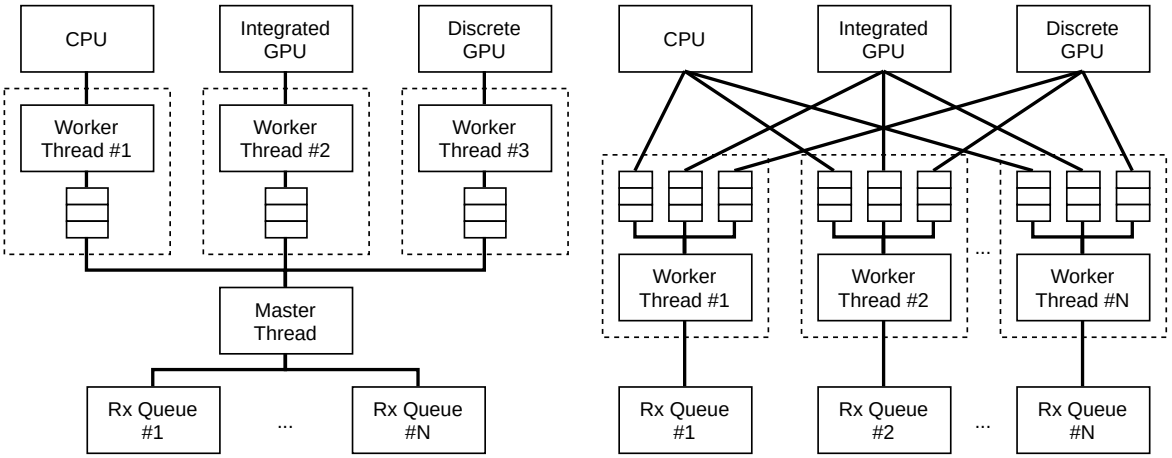
thread is responsible for capturing packets from the NIC, distribute them to the corresponding devices and keep statistics and utilization of each device, which allows it to better distribute the incoming traffic based on the utilization and processing capabilities of each device. The major disadvantage of the master-worker architecture is its excessive use of locks between the different threads. Not only that, but adding more NICs to the system requires spawning more master threads, increasing the number of locks. To overcome this, a shared-nothing model was proposed, which did not require synchronization between threads, pairing each available network interface with a unique thread (fig. 2.2(b)).

Despite the fact that, according to the authors, these models sustain similar performance for traffic with packets of 1500 bytes, the difference becomes more clear as the packet size decreases. For 60 bytes sized packets, the shared-nothing model achieves $3\times$ greater throughput. This difference in performance when varying the packet size is to be expected. The vast majority of NFs use only the packet headers, which makes their throughput under large packets I/O bound. On the other hand, when processing small packets, the throughput becomes CPU bound. That is why, typically, throughput is represented in packets per second units, instead of bits per second when analyzing the performance of an NF. Scenarios that are I/O bound are not representative of an NF's performance, and as such, minimal sized packets are used for performance measures.

In order to minimize power consumption and latency or maximize throughput, the scheduler performs an initial analysis of the power consumption, latency and performance of each available device. It then decides the best combination of available devices that meet the desired target policy, keeping track of incoming traffic in order to adapt the batching and batch processing.

The authors implemented four network packet processing applications (what we call software network functions)—IPv4 Packet Forwarding, Deep Packet Inspection, Packet Hashing, and Encryption. They evaluated the throughput, latency and power consumption achieved under an energy-critical policy (*i.e.* handle all input traffic at the maximum energy efficiency). The scheduler switches to the configuration that keeps the selected hardware component under the computational load which is required to process the incoming traffic (including not using it if is not deemed necessary). However, latency can vary under different configurations. The scheduler achieves energy savings ranging from $0.3\times$ on heavy workloads and $3.5\times$ on lighter loads. On a final remark, the IPv4 Forwarding application reaches almost 30Gbps under this policy.

Although their work focused more on efficient utilization of computational capacity of available resources and, with that, energy savings, their shared-nothing model will be used in this thesis as guidance for automated generation of parallel network functions.



(a) Master-worker architecture.

(b) Shared-nothing architecture.

Figure 2.2: The 2 architectures used by Papadogiannaki *et al.* [2].

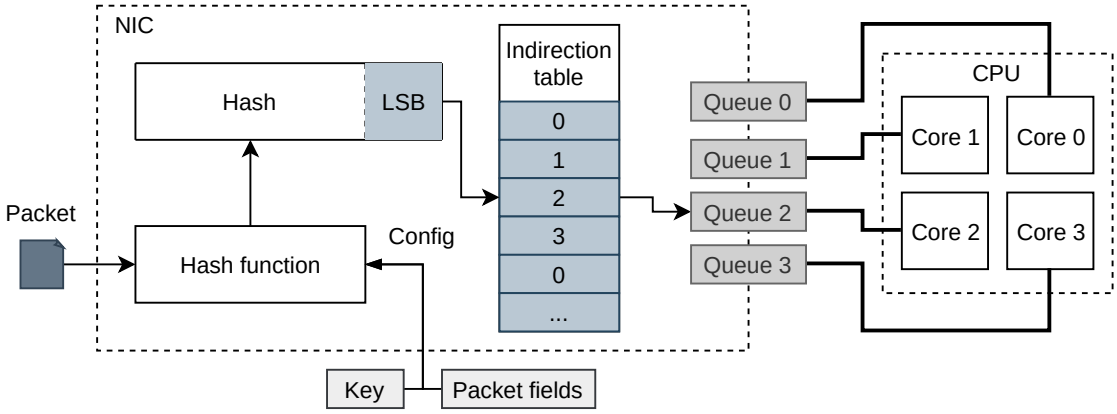


Figure 2.3: RSS mechanism.

2.4.3 Packet distribution

When parallelizing an NF, a question immediately arises—how can one distribute incoming packets among the NF's parallel instances? This distribution needs to happen at line-rate speeds in order to not bottleneck the connection between the NIC and the NF. Moreover, if we want the NF to follow the shared-nothing model and avoid the performance toll associated with synchronization, there might be a need to redirect specific traffic to specific cores. An NF that, for example, keeps track of how many packets arrived from the same IP will need to receive packets coming from the same IP always on the same core. If packets were randomly distributed among the cores, this NF would need to synchronize its instances, avoiding concurrent writes to the same data. As such, the packet distribution method needs also to provide some way to control which packets are sent to which cores.

Receive Side Scaling (RSS) [29] is a packet distribution technology implemented on the NIC that runs

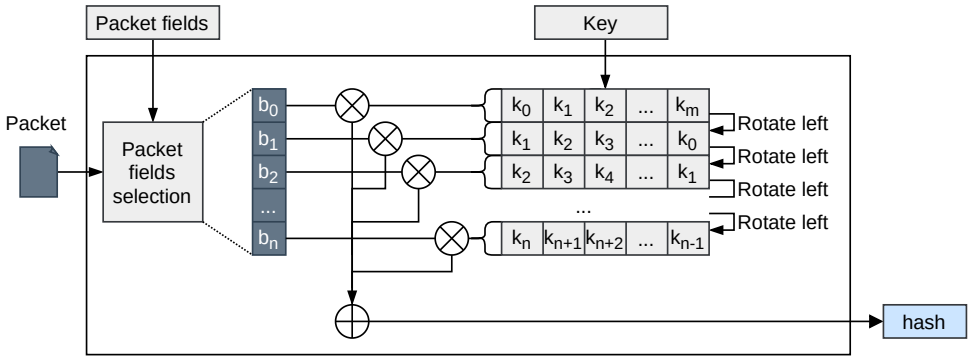


Figure 2.4: Toeplitz-based hash function.

at line-rate and provides (some) control of assignment of incoming packets to specific cores. Figure 2.3 depicts how the RSS mechanism works. It can be configured with both a key and a list of packet fields. The packet fields are used as input to an RSS toeplitz-based hash function [30], and the key can be interpreted as the hash function itself. This hash function, shown in fig. 2.4, iterates through all the input bits, rotating its key in every iteration. When the current input bit is one, the key is xor'ed against the hash output, which corresponds to a 32-bit value that initially starts at zero. The output of the hash function, *i.e.* the hash, is then used to index an indirection table. This indirection table stores packet queue identifiers which, when associated with only one core, can be used to steer the packet to that core.

Symmetric RSS. Using the RSS default configuration, packets associated with the same TCP connection will be processed on different CPU cores—a change on the tuple {*source IP, destination IP, source port, destination port, protocol*} of a packet leads to a different hash value, and therefore is processed on a different core. This means that NFs that maintain state according to TCP connections would have to synchronize data between the different instances, using locking mechanisms to access it, ultimately degrading performance.

To avoid this, Woo and Park presented Symmetric RSS [31], an RSS configuration that allows for packets associated with the same TCP connection to be sent to the same CPU core. They achieved this by tweaking the hash function's key so that the resulting independent hashes of two packets with source and destination IPs swapped (same goes for source and destination ports) are equal, and therefore these packets are sent to the same CPU core. This is made possible by a specific property of the RSS hash function—each packet field given as input is always associated with a particular set of bits of the key. Therefore, a symmetry between the fields can be represented as a set of equalities between groups of bits in the key. In the end, the key will have bits bounded to the same value, and bits unbounded and free to take any value.

This small but very important contribution is a solution for a particular case of symmetry between packet fields. However, sending packets associated with the same TCP connection to the same core only

allows the parallelization without the use of locking mechanisms of NFs that maintain state according to the TCP connections. Other NFs might require packets with symmetries between other packet fields to be sent to the same core. There can also be a need for symmetries between packets arriving from different NICs, or even some other complicated non-symmetric constraint between packets. In this thesis, we generalize and automate the process of acquiring RSS configurations that send packets following user given constraints to the same core.

Balancing the indirection table. The indirection table of RSS, containing queue indexes, is used to associated RSS hashes to cores. The least significant bits of the hash are used to index this table, and the retrieved value, in conjunction with the fact that cores are associated with queues, is used to redirect the packet to a specific core.

Typically, the indirection table is uniformly filled with queue indexes. This presents no issue if an NF is subjected to uniform traffic, uniformly distributing packets through cores. However, the Internet traffic typically does not follow a uniform distribution (it can under attacks, which is why traffic following a uniform distribution can be used by IDSs to detect these attacks). The expression ”mice and elephants” is typically associated with the Internet's traffic distribution [32–34], *i.e.*, a small percentage of flows represent a large percentage of the traffic. This traffic distribution can be modeled not by a uniform distribution, by a Zipfian one [35]. Under this scenario, the indirection table is not uniformly accessed, and a few cores will tend to receive the majority of the traffic.

With this in mind, Barbette *et al.* [36] developed RSS++. They showed that dynamically modifying the indirection table according to the load of each core can improve the NF's performance. By having each thread record the time spent processing the packet, a master thread could use that data to balance the indirection table if needed. In this thesis, we incorporate this technique to achieve better scalability values by decreasing traffic skew.

2.4.4 RSS on DPDK

Wu *et al.* [23] evaluated different design options of sketch-based network measurements using the DPDK framework on general purpose multi-core machines with 100Gbps network links. DPDK includes a set of poll mode drivers—PMDs, APIs to configure the devices and their respective queues—for a set of NICs. It enables the NF to quickly receive, process and deliver packets by accessing the RX and TX descriptors directly without any interrupts [37]. The authors consider two types of configurations, one using a single RX queue—they call this NONE mode—and another using multiple RX queues—RSS mode.

Their study focused on three streaming algorithms for network traffic analysis and monitoring with the goal of finding the flows that have the biggest number of packets within a certain amount of time—Count-Min sketch, Reversible sketch, and Simple Hash Table. Using these algorithms, they presented

two parallel designs for NONE mode—SYNC and ASYNC—and one for RSS mode. On the SYNC design, all cores work in a synchronous way, and as only one RX queue is available, one of the cores—the master—will receive the batch of packets and inform the other cores—the slaves—to start working on their corresponding packets. Each core informs the master when its task is completed, and the master only receives the next batch when every core completed their task. The ASYNC mode differs from the SYNC mode by allowing each slave to process its packets independently from the other slaves. This can be achieved by giving a ring buffer to each core, which will be filled by the master. On the other hand, RSS mode uses multiple RX queues, so there is no need for a master core to distribute the packets to the other cores.

Regarding the data structures used by the algorithms, there can be a local copy on each thread, or be shared among them. The former approach doesn't require concurrent updating mechanisms, contrary to the latter (and with this, the typical synchronization performance toll). However, the authors claim that using a shared copy can conserve memory space and take advantage of cache locality. For these reasons, the authors present methods for updating a counter, a set, and a critical section concurrently.

Although these safety mechanisms for concurrent updates to shared data can be useful assets when designing a parallel implementation of an NF, in this thesis we aim to generate parallel implementations that run as independently as possible. In fact, the authors conclude that using separate data structures on an RSS design provides lower packet drop rate and packet processing time when compared to a shared copy approach, confirming the need for avoiding shared data between NF instances on parallel implementations.

2.5 Parallel NFs

In the recent years, many solutions were proposed to tackle the challenge of network function parallelization. In order to show that building parallel network functions is not a trivial task, and instead requires careful considerations of packet distributions, data sharing, cache coherence, and manual configurations, we present a subset of implemented software network functions that take advantage of multi-core architectures.

2.5.1 Statistical Traffic Analyzer

Trevisan *et al.* [21] presented design choices to enable Statistical Traffic Analyzers (STAs) to collect statistics on a multi-10Gbps line-rate.

The challenge is not how to efficiently transfer packets from the NIC to the main memory anymore—they used the DPDK framework to handle that—but rather how to process packets in userspace as fast as possible by taking advantage of multi-threading technologies. Per-flow load balancing, required by

STAs, can be handled by the NIC using Symmetric RSS. In order to deal with unexpected large packet processing delays, packet acquisition libraries implement circular buffers, preventing packet drops. But because these circular buffers tend to have a range of 1MB, they can only spend 1ms absorbing at a 10Gbps throughput scenario.

The authors proposed a decoupling of each analysis module using a large buffer, an acquisition thread that extracts packets from the RX queue and enqueues them to the large buffer, and a processing thread that dequeues packets from the large buffer and processes them. This can be achieved by using lock-free shared buffer data using zero-copy acquisition. These two threads, one responsible for packet acquisition and the other for analysis functions, run on the same CPU core. As such, if the packet acquisition thread retrieves packets by polling, it leaves no room for another thread to run on the same core. As such, the authors resort a scheduler strategy provided by the Linux kernel—*SCHED_DEADLINE*. This strategy schedules threads within a previously configured deadline, allowing for both threads to execute at the same time in an almost periodical fashion. This strategy allows for DPDKStat, the STA developed by the authors, to achieve 21Gbps of throughput without losses using 4 physical cores.

In a NUMA environment, DPDKStat achieves 40Gbps with no packet losses with 16 physical cores available, and although the authors were unable to test this by the 16 RSS queues limitation of the NICs, they claim that the system could potentially handle even more traffic.

2.5.2 Intrusion Detection System

In 2012, Jamshed *et al.* developed Kargus [38], a highly scalable software based Intrusion Detection System (IDS) that exploits the full potential of commodity computing hardware. Using a 12 core machine with 2 GPUs, it achieved 33Gbps of normal traffic, and 9 to 10Gbps when all packets contained attack signatures. This was made possible by batch processing incoming packets and parallel execution with a load balancing algorithm that dynamically adjusted resource consumption according to the input workload.

Kargus adopts a single-process multi-thread architecture, with one thread running on each CPU core. Each Kargus thread can either be an IDS engine or a GPU dispatcher. The IDS threads read the incoming packets from multiple NIC RX queues and perform the IDS-related operations. They also detect if their own CPU core is overloaded, and offload their workload to a GPU-dispatcher thread.

The authors adopted this architecture over the multi-process model to allow efficient sharing of attack signatures information among threads—avoiding having a copy of the ruleset, most of the times GBs of data, on each process—and to avoid reserving a portion of GPU memory to employ GPUs on multiple processes. They also decided to not go with the pipeline model, as it does not exploit multiple RX queues, suffers from cache bouncing, and would lead to inefficient use of CPU cores, as the I/O threads

would underutilize CPU cores comparing to the analyzer threads.

Regarding packet acquisition, Kargus allocates a large contiguous kernel buffer (*e.g.* 8MB) per RX queue to allow packet acquisition batching, and modifies the RSS seed (also mentioned as the RSS hash function key) to allow for packets corresponding to the same TCP connection to be steered into the same RX queue (again, using Symmetric RSS). Kargus is NUMA aware, and in order to avoid cross-domain memory access, each IDS engine thread handles only the packets coming from the NIC attached to its NUMA node, and do not share global variables.

Nam *et al.* [39] extended Kargus and were able to achieve 79Gbps of throughput when dealing with packet sizes of 1514 bytes. They adopted a shared-nothing architecture, where each thread is running a separate IDS engine and no data structure is shared with other threads, eliminating the need for inter-core communication and synchronization mechanisms between them. In order to reduce the overhead of per-packet operations—typically where the bottleneck can be found—the authors offloaded some operations from the threads to the NICs, taking advantage of programmable NICs (*e.g.* decoding and flow hash computation). Furthermore, they reduced the number of cache misses by removing the rarely used fields of the packet metadata structures, forcing them to be dynamically allocated on demand, and this way shrinking the size of the packet metadata. Finally, in case the many-core processor faces a high workload, the subsequent flows are dynamically offloaded to a host-side CPU, preventing packets from being dropped.

2.5.3 Router

RouteBricks [40] is an architecture designed for router functionality parallelization across servers and cores within servers using commodity hardware and without centralized components. Each server can be connected to external nodes, and the traffic is, using a load balancer, distributed among the servers, which will perform packet processing for one up to a few router ports.

The hardware used included a server prototype using Intel Nehalem architecture (released in 2008), employing a NUMA architecture using two CPUs and two NICs holding two 10Gbps ports each. The Nehalem architecture provided memory access parallelism, as it used multiple memory controllers, each integrated within a socket. As shown at the time [41], this provided a solution for the bottleneck reached for using a shared-bus with a single external memory controller whilst undergoing heavy loaded memory and I/O operations.

RouteBricks relies on multi-queue NICs to avoid network queue access synchronization between each core. Therefore, each network queue is accessed by a single core. It also takes advantage of the NIC's multi-queue to address the problem of not having as many cores available as the number of NIC ports, and to split the packet stream of one port to multiple cores in case a single core is not enough to handle the traffic. A single server running RouteBricks achieved an IP routing performance

Listing 2.1: Example code used for symbolic execution.

```
1  int f(int input) {
2      int a = input;
3      int b = a + 1;
4
5      if (b >= 5) {
6          return b * 5;
7      }
8
9      return a;
10 }
```

rate of 24.6Gbps, being bottlenecked by the CPU. Using four servers on cluster mode, it achieved a performance rate of 35Gbps.

2.6 Network function verification

The benefits of transitioning network functions from dedicated appliances to commodity hardware were mentioned in the beginning of this chapter. However, as this is associated with software-based implementations of network functions, the lack of reliability of software becomes a new problem, introducing bugs, unexpected behavior, and security vulnerabilities. As such, software verification has been applied to NFs as a means to counter these reliability issues. In this section, we explore how we can use the byproducts of the verification process to build a sound and complete representation of the NF's functionality that we can later use to automatically generate a parallel implementation of it.

2.6.1 Symbolic Execution

Symbolic execution is a verification technique that can be used for—but not exclusively—automatic test input generation, live code path extraction, and verifying if a program violates certain properties (*e.g.* division by zero and NULL pointer dereference).

This technique analyses the program, following its simulated execution assuming symbolic values for inputs instead of real values. Faced with an operation on one of these values, the symbolic value is manipulated in order to reflect this operation. When the execution branches based on a symbolic value, both branches are followed, and this simulation continues on both paths independently. When a path terminates or hits a bug, the symbolic execution engine can use a solver to translate symbolic values into a concrete inputs that follow the given path. KLEE [42] is an example of a powerful symbolic execution tool designed for robust, deep checking of a broad range of applications.

Figure 2.5 represents the result of a symbolic execution of the function shown in listing 2.1. The argument of the function (`input`) is interpreted as a symbol (α), and might take any integer value. The

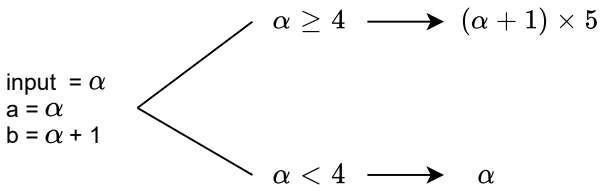


Figure 2.5: Result of symbolic execution of listing 2.1.

second line of the function stores the argument in the variable `a`. As the argument can take any integer value, this freedom is propagated to `a`, taking the value of symbol α. The third line stores an arithmetic operation of this symbol into a second local variable—`b`. Again, as we allow α to take any possible integer value, we are unable to pinpoint the resulting value of this arithmetic expression. Therefore, `b` assumes the expression $\alpha + 1$. By the fifth line, the program bifurcates its behavior depending on the current value stored in `b`. As `b` is assigned to an expression containing α, both branches are valid code paths by this point. Therefore, the symbolic execution bifurcates, with one branch assuming the branch condition was met, and the other assuming it was not. Finally, each branch returns from the function with expressions depending on the symbol α, as both represent in the code arithmetic expressions with a variable depending on α. By the end, we can safely say that the function will return $(\alpha+1)\times 5$ iff $\alpha \geq 4$, and α iff $\alpha < 4$, without actually executing the function.

2.6.2 Vigor

Zaostrovnykh *et al.* developed Vigor [3], a software stack and toolchain for building and running NFs that are guaranteed to be correct while preserving competitive performance and developer productivity, targeting NFs implemented on top of standard network I/O frameworks—*e.g.* DPDK. Vigor takes as input an NF implementation and a specification that the implementation must satisfy, and it automatically produces either a proof that the implementation satisfies the given specification, or a counter example. It also verifies low-level properties like memory safety, crash freedom, hang freedom, and absence of undefined behavior.

Using Vigor, NF developers can write NF code on top DPDK and use data structures from Vigor's library (libVig) to implement stateful parts of the NF—*i.e.*, the state that persists across packets. The API for accessing libVig's data structures that ensures persistence of state across packets (libVig's functions) is verified against a beforehand written contract which states that given some pre-conditions, an invocation to the function leads to the post-conditions. The Vigor verification process also includes verification of the entire NF stack, including the NF's core implementation, the DPDK's utilities used by the NF, the NIC driver, and the underlying operating system. In order to perform this verification, the authors developed a lean operating system (NFOS) that can be symbolically executed in an exhaustive

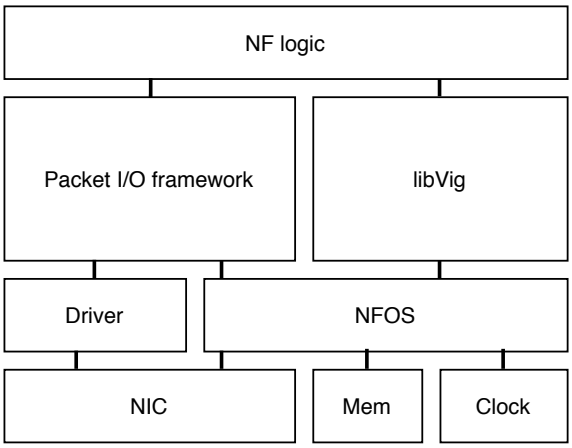


Figure 2.6: The Vigor stack [3].

manner using KLEE, and modified both DPDK (in order to make it verifiable) and KLEE (to speedup the process, by avoiding enumerating unnecessary paths). The Vigor stack is depicted in fig. 2.6.

The same authors, in conjunction with others and prior to presenting Vigor, presented VigNAT [24], a Network Address Translator (NAT) built on top of DPDK that is guaranteed to be semantically correct, crash-free, and memory-safe. Vigor uses VigNAT's split NF architecture in order to provide verification, doing it in four steps:

1. Use of exhaustive symbolic execution of the stateless NF code to obtain all live paths—also called *call paths*—through the code.
2. Convert the resulting symbolic execution traces to programs, each one representing one path through the stateless code.
3. Annotate each trace with lemmas corresponding to the NF specification.
4. Validate each annotated trace, using the pre- and post-conditions from the libVig API contracts.

Although writing NFs with Vigor does not require verification expertise, it imposes some constraints:

1. Vigor assumes that the stateless part of the NF has one top-level infinite loop—the event loop—that receives/sends packets, and all other loops are bounded.
2. Any state that persists across packet arrivals must reside in libVig data structures.
3. NF developers must respect the memory ownership model of libVig data structures—*e.g.*, after inserting a value into a map, the caller is not allowed to change that value by reference; instead, it must remove the entry from the map, modify it, then re-insert it.
4. It does not handle symbolic-pointer arithmetic, and does not support all of the C standard library (because of inherited constraints imposed by KLEE and VeriFast).

The authors presented five NFs verified with Vigor: a NAT, a Maglev load balancer, a MAC-learning bridge, a traffic policer, and a stateful firewall, and show that each performs on par with standard,

functionally equivalent, non-verified NFs that use the same high-performance packet I/O framework. These examples can be useful for testing the automated tool for generating parallel NFs proposed on this thesis.

The efficient mechanism for verification provided by Vigor also provides the tools needed for analyzing the NF's code and retrieving information about data dependencies between processing of different packets. As has already been mentioned, eliminating data sharing between threads running on different cores is of major importance. Knowing how a certain NF's implementation maintains state between the processing of different packets gives enough information to, if possible, obtain an RSS configuration specific to this implementation that eliminates data sharing between the threads. We address this in more detail in the next chapter.

2.6.3 Call paths

Because the call paths generated by Vigor contain a complete representation of an NFs behavior, they allow for automated and generalized methods for NF modifications. In this thesis, specifically, they are used to automatically generate parallel implementations. They show all the possible code paths that can happen, triggered by a packet's arrival, and are generated via symbolic execution of the NF.

The call paths are composed of both constraints and calls. The constraints uniquely identify the call path, indicating which conditions must be true—or false—if a packet were to trigger this specific code path. If a packet arrives and all those specific call path's conditions are met, then what follows is the execution of every call—with their corresponding arguments—specified in the call path.

Considering the NF example in listing 2.2, its corresponding call paths can be seen in listing 2.3. When symbolically executing the function `process_packet`, bifurcations on the execution occur in lines 4, 9, and 13. When the condition in line 4 holds true, only a single code path can be taken (the packet is dropped). On the other hand, the branches in lines 9 and 13 can happen independently, and therefore their possible combinations give out 4 possible code paths. In summary, there are 5 possible code paths in this function, and all are enumerated in listing 2.3 in lines 1, 5, 12, 18, and 26 with their corresponding symbolic constraints. Inside each enumerated code path is a list of function calls triggered by that code path.

The call paths represent, therefore, a complete representation of the NF's behavior, and allow for automatic manipulation of the NF. We propose, in this thesis, the use of these call paths to automatically parallelize NFs in a push-to-parallelize fashion.

Listing 2.2: Pseudo C code of an NF.

```
struct Map* tb;

void process_packet(int device, pkt_t p) {
    if (!is_tcp(p)) {
        drop(p);
    }

    else {
        if (!map_contains(tb, p.src_port)) {
            map_put(tb, p.src_port, device);
        }

        if (map_contains(tb, p.dst_port)) {
            int device_dst = map_get(tb, p.dst_port);
            forward(p, device_dst);
        } else {
            broadcast(p);
        }
    }
}
```

Listing 2.3: Call paths generated from the symbolic execution of listing 2.2.

```
1: !is_tcp_result
    => is_tcp(tb, p.src_port) -> is_tcp_result
    => drop(p)

2: is_tcp_result && !map_contains_0 && !map_contains_1
    => is_tcp(p) -> is_tcp_result
    => map_contains(tb, p.src_port) -> map_contains_0
    => map_put(tb, p.src_port, device)
    => map_contains(tb, p.dst_port) -> map_contains_1
    => broadcast(p)

3: is_tcp_result && map_contains_0 && !map_contains_1
    => is_tcp(p) -> is_tcp_result
    => map_contains(tb, p.src_port) -> map_contains_0
    => map_contains(tb, p.dst_port) -> map_contains_1
    => broadcast(p)

4: is_tcp_result && !map_contains_0 && map_contains_1
    => is_tcp(p) -> is_tcp_result
    => map_contains(tb, p.src_port) -> map_contains_0
    => map_put(tb, p.src_port, device)
    => map_contains(tb, p.dst_port) -> map_contains_1
    => map_get(tb, p.dst_port) -> map_out_0
    => forward(p, map_out_0)

5: is_tcp_result && map_contains_0 && map_contains_1
    => is_tcp(p) -> is_tcp_result
    => map_contains(tb, p.src_port) -> map_contains_0
    => map_contains(tb, p.dst_port) -> map_contains_1
    => map_get(tb, p.dst_port) -> map_out_0
    => forward(p, map_out_0)
```

3

Architecture

Contents

This thesis expands the NFV state of the art by presenting a tool designed to generate parallel implementations of NFs that are semantically equivalent to their sequential counterparts. Maestro (the mentioned tool) allows developers to effortlessly scale their NFs—written in the Vigor framework—with multiple cores. This chapter, divided into five sections, aims to describe how Maestro was built to provide a solution for automatic parallelization. In the first section, we explain how one can achieve a parallel shared-nothing solution architecture of an NF by sharding packets by their fields. The following section exposes an architectural overview of our solution. The remaining sections are a deep dive into each component of Maestro's architecture.

Alongside the exposition of our proposed tool, we provide a running example of an NF subjected to the Maestro's pipeline. The use of a practical example, introduced in the following section, aims to provide a better understanding of how each of the Maestro's components work in conjunction towards the goal of parallelizing NFs.

3.1 Achieving shared-nothing modeled parallel NFs by sharding

Parallelization of applications came to life alongside the birth of parallel execution units, or CPU cores. As illustrated in the previous chapter, it comes associated with multiple challenges. The burden of parallelizing NFs has traditionally fallen onto the developer, who becomes responsible for taking all these challenges into consideration when crafting a second, now multi-core, implementation of their application. This second implementation must ideally scale with the number of cores, while also maintaining the semantics of the original sequential implementation.

However, one can use properties of NFs, and specifically Vigor's NFs, to not only alleviate the difficult process of parallelization, but automate it altogether. We will now explore how both data parallelism and sharding can be used to parallelize NFs, and how we can use Vigor to automate it.

3.1.1 Data parallelism

Parallelization of an NF can be approached as a data parallelism problem. When parallelized, the same NF implementation runs in multiple cores, each receiving a portion of the traffic. This alleviates the difficulty associated with parallelization, as it allows for this process to be done with minimal modifications to the sequential code. The problem is shifted from heavily modifying the sequential implementation to controlling and distributing the input given to the program—which, when dealing with NFs, is the traffic.

However, data parallelism alone is insufficient. Maintaining state across the processing of packets is, most of the times, vital for achieving their purpose. The parallel instances can either share this state, or each maintain a core-specific copy. In the former approach, synchronization mechanisms are necessary to ensure that each instance does not write concurrently to the shared memory, which would lead

to erroneous and unpredictable behavior. As discussed in the previous chapter, synchronization mechanisms are associated with a performance toll, and therefore we will mainly devote our attention to the parallel shared-nothing model. In this model, each core has its own state copy, and can read and write to it without needing to communicate with the other cores. However, when using this model, mindless steering packets to each core does not guarantee semantic equivalence between the sequential and parallel implementations.

To understand this, consider an NF whose purpose is to forward the first 100 packets that it receives, dropping all that follow. In a sequential implementation, if the NF receives 101 packets, only the first 100 are forwarded. However, when running two shared-nothing parallel instances of this NF and randomly steering the incoming packets to either the first or the second core, the semantics may differ. Both implementations can be semantically equivalent only if, in the parallel scenario, one of the cores receives no packets. Otherwise, each core receives a fraction of the 101 packets, and no packet is dropped, as the 100 packet limit is not met in either core.

The semantics would be kept if the parallel instances were not running independently, sharing state-related data structures between them. This would translate to, in the given example, using a packet counter shared by both cores. However, as was exposed in the previous chapter, locking mechanisms take a big toll in performance. Avoiding their use is vital to achieve good performance values.

3.1.2 Minimizing locking mechanisms through sharding

As described in the previous chapter, Vigor extracts information from the NF, generating its corresponding call paths. These call paths represent all the paths in the NF's code that can possibly be taken upon a packet arrival. With these cal paths, we can retrieve information on how a packet was modified, if it was dropped or forwarded, and to which device it was forwarded. Moreover, we can also track all the changes made to state-related data structures (formally verified structures living in the libVig library).

Maestro takes advantage of the Vigor framework by using its generated call paths to extract information regarding possible dependencies between parallel instances. The extraction of these dependencies is crucial to achieve a parallel implementation that minimizes the number of locking mechanism. Maestro tries to replace these locking mechanisms, intended to prevent erroneous behaviors upon concurrent accesses to memory, with an RSS-based mechanism that redirects traffic matching specific patterns to specific cores—*i.e.*, sharding. How this careful redirection of packets can be used to maintain an NF's semantics is better exposed firstly by the means of an example.

Let us envision an NF whose sole purpose is to obtain information on the distribution of the TCP/UDP source ports used by its arriving packets. Its pseudo-code is illustrated in listing 3.1. All the NFs' state—in this case, a table storing a counter for each source port—is stored in a libVig vector.

The need to prevent simultaneous writes and read-writes to the same memory addresses, at first

Listing 3.1: Example of an NF responsible for obtaining information on the distribution of TCP/UDP source ports.

```
1 struct Vector* ports_table;
2
3 void process_packet(int device, pkt_t p) {
4     int counter = vector_get(ports_table, p.src_port);
5     vector_update(ports_table, p.src_port, counter + 1);
6
7     // forwarding packet
8     fwd(p);
9 }
```

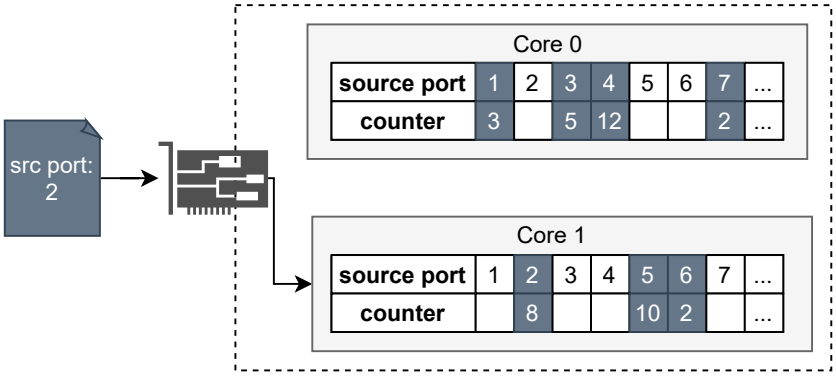


Figure 3.1: Example of packet-related state partitioning. The vector is illustrated as a table associating source ports with counters. The darker colored cells are the active entries in that vector instance.

glance, implies the requirement of locking mechanisms. This is only true if multiple parallel instances concurrently write (or read-write) to the same memory.

Maestro takes advantage of the fact that the global state of this NF is accessed using a specific packet field—the source port—and partitions the vector into multiple sections, each used only by a single parallel instance. In this case, it configures sharding by the source port by configuring RSS to steer packets with the same source port to the same core. This guarantees that information associated with a specific source port can always be found on the same core. This partition can be seen in fig. 3.1, which illustrates the steering of a packet to its specific designated core, according to its source port value.

It is important to note that this partition is *virtual*, not necessarily contiguous, and only achieved by the steering of packets. Each parallel instance allocates its unique and independent data structures, and each NF only writes to a specific subset of its data structure's memory. Therefore, each parallel instance, whenever possible, behaves as it would in a sequential scenario, neither communicating nor sharing data with the other instances, and dealing with only a subset of all the possible traffic. This is called the shared-nothing model, introduced in the previous chapter, and it is the parallelization solution that Maestro employs by default in its automatic generation of parallel implementations.

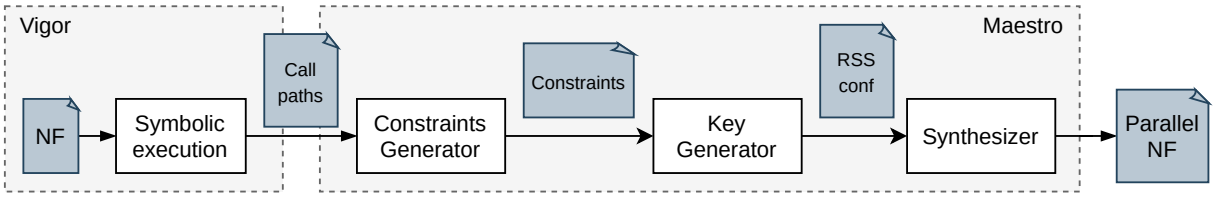


Figure 3.2: Maestro's architecture.

3.2 Overview

Maestro's architecture is a pipeline composed of three modules: the Constraints Generator, the Key Generator, and the Synthesizer. Each module tackles a specific challenge in the problem of automatic parallelization of NFs. The pipeline is comprehensively explained in this chapter, and the whole architecture can be visualized in fig. 3.2.

The Constraints Generator uses the call paths generated by Vigor to extract information on how the NF accesses and modifies its state, generating a list of libVig accesses (LVAs). It then uses this list of accesses to find all the constraints that packets processed in the same core need to satisfy for semantic equivalence to be preserved between the sequential and parallel implementations. Running this module against the previous example (listing 3.1) would result in a constraint dictating that packets with the same source port should be processed in the same core.

The module that follows, the Key Generator, takes the list of constraints and finds a set of RSS keys that steer packets to the same core if they satisfy the given constraints. This is done by translating the problem into a Satisfiability Module Theory formula, which is then handed to the Z3 solver [43] to find RSS keys that satisfy the formula.

Finally, the Synthesizer, using this newly found RSS keys and the NF's call paths, generates a parallel implementation of the original NF.

3.3 Constraints Generator

The main goal of this module is to find the constraints that packets processed in the same core must satisfy if the parallel implementation of the NF is to be semantically equivalent do the sequential implementation. To achieve this, it needs a complete representation of the NF in question. This requirement is met using the NF's call paths generated by Vigor, which are parsed into a list of calls containing symbolic expressions. Vigor already provides a tool for parsing its call paths, and so this module only needs to use this tool to get a data structure containing all the information associated with the NF. From this point on, we use the term "call paths" to represent both the files generated by Vigor and the data structure containing this parsed information into symbolic expressions, as they are interchangeable.

The call paths are then submitted to a pipeline of three stages:

1. LVA generation

   The call paths are used to generate a list of libVig accesses (LVAs). Each LVA contains information about that specific access, *e.g.* the memory address of the object being manipulated, which expression was used to access its memory, and which packet fields are contained in this expression.

2. LVA analysis

   This stage filters the given LVAs, removing both duplicates and those associated with an object which was only read and never modified. Then, given the filtered LVA list, it discovers how many ports are being dealt with and chooses the optimal RSS packet fields configuration. By this point, Maestro tries to find some property of the NF that prevents it from generating a parallel shared-nothing implementation. For example, using again the example in listing 3.1, if it kept a per-packet counter instead of a per-source-port counter, the NF would need to share that state with all its instances. There would be no sharding that allowed for a semantically equivalent shared-nothing parallel implementation. Maestro captures these scenarios in this stage, and returns an error describing why it was unable to generate the parallel shared-nothing implementation.

3. Constraint generation

   The final stage of this pipeline uses the modified list of LVAs to generate the constraints that packets arriving at the same core must satisfy. These constraints are later given to the Key Generator.

3.3.1 LVA generation

An LVA is a structure extracted from the call paths containing details associated with a call to the libVig API. Each libVig call encountered in the call paths can be associated with a specific access to the NF's state, and indicates not only the function that was called, but also its arguments and, more importantly, their packet dependencies. Table 3.1 contains a description of all the information contained in an LVA.

An example of an LVA is illustrated in listing 3.2. It was generated by the presence of a call to a libVig API function in the call paths of the NF under analysis. Although we do not show the NF's source code that triggered this LVA's creation, the LVA contains all the information we need to understand the context of this state call. It corresponds to a `map_get` call. The packet that triggered this function call was received from device 0, and later forwarded to device 1. We also know that the expression passed as a key to the libVig `map` contained packet-related data. Specifically, this data is associated with the the bytes 16, 17, 18, and 19 from the packet's layer 3 header, which follows the protocol 2048. As these bytes correspond to the IPv4 destination address bytes, we know that this field was used as a key to index the `map`.

LVA field	Description
file	Call path file containing the libVig call associated with this LVA.
src device	Device that received the packet.
dst device	Device to which the packet was given to be forwarded.
interface	The name of the called libVig function.
operation	Type of operation associated with this LVA (typically READ or WRITE).
object	Address in memory of the libVig data structure being manipulated.
read	This field is associated with the read argument, *i.e.* the argument associated with the memory that will be read. For example, the libVig map receives a key as an argument, and accesses the stored data using that key. In this example, this field would contain the expression used as the key. This field is relevant in the process of finding the appropriate packet sharding that will lead to generating a shared-nothing parallel implementation. As such, a specific set of fields can follow this field, indicating information regarding packet-related data in this field's expression: • layer: the OSI module network layer associated with packet data in this expression. • protocol: the number identifying the protocol associated with the packet data—*e.g.* layer 3 and protocol 2048 (in decimal) corresponds to IPv4. • fragments: the packet data sub-expressions that can be found on the original expression, containing information regarding the byte offset and the packet data expression itself. • dependencies: the bytes from the packet that are being used in the expression.
write	Similar to the "read" field, but associated with a write argument. In the example used above, its expression would correspond to the data being written into the map.
result	The expression representing the data retrieved from the data structure upon a read operation.
success	A boolean value that indicates if the function was successful, or failed (it can fail if, for example, one tries to read from a map with a key that was never used to store data).

Table 3.1: LVA fields' descriptions.

In order to obtain this information, Maestro loops over the call paths given by Vigor, grabbing both the constraints and the calls associated with the call path. These constraints contain information about the conditions that must be met if a packet is to trigger that specific code path—*e.g.*, each packet layers' protocol, the device (port) it came from, and the value that must be stored in a specific key in a map. Having both the constraints and the calls, Maestro then loops over the calls, where it can encounter two possible types of calls:

Packet access calls. In order to handle variable length headers, the Vigor API allows NFs to access the packets in chunks, as opposed to a single buffer containing the whole packet. In this stage, these packet chunks requests are traced and used to associate a specific packet layer to an expression. In the example shown in listing 3.2, we can see the call to a libVig function (`map_get`) that uses an offset of the symbol `packet_chunks` as its argument. This symbol contains information regarding the packet, and its association with a specific packet field can be made by checking how many packet chunks requests were made before this call.

Listing 3.2: LVA example.

```
1  ========================================
2  Access 162
3    file          test000017.call_path
4    src device    0
5    dst device    1
6    interface     map_get
7    operation     READ
8    object        1073781208
9    read          (ReadLSB w32 (w32 57) packet_chunks)
10
11   layer         3
12   protocol      2048
13   fragments     offset 41 expression (ReadLSB w160 (w32 41) packet_chunks)
14   dependencies 19
15                 18
16                 17
17                 16
18
19   result        (ReadLSB w32 (w32 0) allocated_index)
20   success       1
21 ========================================
```

The constraints in a call path also contain information regarding each packet chunk, attributing a value (or multiple values) to specific packet fields. This happens if the NF has a behavior that varies depending on packet data, which is very common. An SMT solver is used to query these call path constraints and retrieve information regarding the possible values of the packet fields in the current layer—*e.g.*, the protocol of the current layer.

LibVig calls. Each libVig call has a specific signature. In order to extract information on how the NF's state is being accessed, Maestro needs to know all the libVig calls' signatures. That is, if more data structures are added to libVig, then Maestro needs to be updated, as the new functions' signatures must be taken into consideration.

Additionally, each data structure in libVig needs to be studied in the context of parallelization. Take the `map_get` signature as an example:

```
int map_get(struct Map *map, void *key, int *value_out)
```

This function is used to access the data associated with the `key` stored in `map`. The retrieved data in written to `value_out`.

The question then becomes—*which situations cause an access to the same memory when this function is called concurrently?* In this case, using the same key would trigger an access to the same memory, and therefore the second argument is used to create a new LVA. In the example (listing 3.2), this corresponds to the `read` field in the LVA.

The expression stored in this field is then traversed in order to find possible dependencies. These can be associated with the packet itself, or some other symbol associated with some other data structure. In the given example, the expression contains the symbol `packet_chunks`, so we know it uses packet information. This information is retrieved by comparing it to previously made packet management calls,

and Maestro finds that it uses packet data associated with the layer 3, protocol 2048 (`0x0800`), and byte offsets 19, 18, 17 and 16 of that layer's header. This data corresponds to the complete destination IPv4 address of the packet. This packet dependency data is stored in the LVA, and will be later used to build the constraints that bind each packet processed in the same core.

When an NF's behavior is not altered according to the packet's protocol, the information retrieved from the solver will be ambiguous. When queried against multiple possible protocol values, it will return that they all might be true. For example, if an NF does not discriminate against TCP or UDP, the solver, when questioned about the layer 4's protocol in the packet, responds saying it could be either TCP or UDP. In that case, the LVA is filled with the fields associated with the specific byte offsets from all the possible protocols that can be associated with any packet that triggers that call path. This can be seen in a more complex LVA example illustrated in listing A.1. The expression in the `read` field in the example uses both information stored in the layer 3 and layer 4 of the packets. However, the layer 4 can either be TCP or UDP in that call path, and so they are both stored.

As concurrent accesses to different objects do not cause erroneous behavior in a parallel scenario, there is also a need to associate the LVA with a specific object. In this case, the object identification is stored in the first argument, `map`, containing its memory address. This information is also stored in the LVA in the field `object`.

3.3.2 LVA analysis

As referenced in the beginning of this section, this step of the Constraints Generator's pipeline firstly deletes duplicated LVAs. It then proceeds to delete all LVAs associated with objects which were only read from and never written to.

The function calls associated with read operations to these objects can still contain packet dependencies, but these are now irrelevant. In a parallel scenario, when data is never modified, the read accesses can happen concurrently without the concern for erroneous behavior. As these read expressions can be packet dependent, and therefore the LVA being packet dependent, these LVAs need to be deleted. Not deleting them would result in these LVAs being taken into consideration when finding the proper RSS configuration.

By this point, if all the LVAs were deleted, Maestro terminates its the pipeline, and returns to the user an RSS configuration with a random key and all the RSS compatible packet fields. This can only be done because having no LVAs left is interpreted as the inexistence of constraints that packets arriving to the same core must satisfy.

The next step is to iterate over all the LVAs in order to extract packet dependencies. These will be used in the Key Generator to feed the RSS mechanism with the proper fields. If an LVA depends on a specific packet field which is not compatible with RSS—*e.g.* MAC addresses—then the pipeline is

terminated, and Maestro warns the user that the NF contains non-RSS compatible dependencies. This would happen if, for example, an NF used a MAC address as a key in a map.

The final step of this stage uses the remaining list of LVAs to determine if independence between parallel instances is achievable. To do this, each specific data structure must be carefully studied, as their use can potentially harm the process of parallelization.

We proceed to explain how the use of libVig verified data structures can influence the process of parallelization, focusing on three of the more frequently used (given the set of NF examples provided by Vigor).

`map`. The `map` is used to store integers associated with a key of arbitrary type. The key is passed through a hash function, and the hash is used to address the stored data.

For two keys to access the same memory in the map they need to be equal. That is, in a parallel scenario, redirecting packets that cause a read (or write) operation with the same key in the same map to the same core is sufficient to maintain semantic equivalence.

`vector`. The `vector` is used to store arbitrary type data associated with an integer index.

Maintaining semantic equivalence when using a `vector` is very similar to the situation where we use a `map`. The accesses to memory in a pair of libVig `vector` calls are only equal if the two indexes are equal. Therefore, the captured `read` expression in a `vector`'s LVA is enough to form constraints linking the packets processed on the same core that lead to semantic equivalence when using this data structure.

`dchain`. The `dchain` (double-chain) is a double linked-list of integers. The values of the stored integers are not chosen by the user but by the `dchain` itself, which is configured to store a limited total number of integers. When using this data structure, one can use the libVig API to allocate (store) a new index, verify if a specific index is already allocated, or rejuvenate a specific previously allocated index. When prompted to allocate a new index, the `dchain` marks one of its integers as reserved and returns it to the user. Each index is associated with a timestamp and is timed-out if it passes a certain fixed amount of time without being asked to rejuvenate by the user. When an index is timed-out, it becomes unallocated.

In the NF examples provided by Vigor, this data structure is used primarily for two purposes:

1. Having the concept of time baked in, it is used to cleanse the NF of flows that are probably no longer being used.
2. Allocate ports associated with flow translations in a NAT.

Using the `dchain` for the first purpose typically involves storing the allocated indexes in other data structure (*e.g.* a `map` or a `vector`). When using the data stored in these other data structures to check if a specific index was previously allocated in the `dchain`, this presents no threat to parallelization. If the packets that always access memory in those data structures are sent to the same core, then we can be

Listing 3.3: Modified version of the NF example in listing 3.1, now using a dchain to incorporate timeouts.

```
1  struct Vector* ports_table;
2  struct Vector* dchain_indexes;
3  struct Dchain *dchain;
4
5  void process_packet(int device, pkt_t p) {
6      int index = vector_get(dchain_indexes, p.src_port);
7
8      if (!dchain_is_index_allocated(dchain, index)) {
9          int new_index = dchain_allocate_new_index(dchain);
10         vector_update(dchain_indexes, p.src_port, new_index);
11     } else {
12         dchain_rejuvenate_index(dchain, index);
13     }
14
15     int counter = vector_get(ports_table, p.src_port);
16     vector_update(ports_table, p.src_port, counter + 1);
17
18     // forwarding packet
19     fwd(p);
20 }
```

sure that, by association, the `dchain` residing in that same core allocated these indexes.

To better understand this, let us consider the example in listing 3.3, which represents a modified version of the example in listing 3.1. The NF continues to serve the same purpose as before—count the number of packets that share the same source port. However, these modifications now allow for the incorporation of the notion of time into the NF. Whenever a timeout occurs in the `dchain`, the counters are restarted.

Under this example, if we configure the RSS to send packets with the same source port to the same core, we can be sure that the packets that trigger an allocation of a new index in the `dchain` are sent to the same core as the ones that trigger a call to `dchain_is_index_allocated`. These indexes are, in this NF, stored in state indexed by the source port. Therefore, the packets that allocate indexes and store them in the `vector` will be sent to the same core as the ones that retrieve the indexes from the `vector`, using the same key, to check if the index was allocated.

However, the user has no control over the value of allocated indexes. Distinct uses of this data structure can both lead to access to the same memory and to different memory. This uncertainty hinders the process of parallelization. While in the `map` case we could assert that using the same key would trigger an access to the same memory, the same can not be said regarding the `dchain`. When allocating a new index, we are unable to know beforehand which memory will be addressed.

Therefore, the second purpose of this data structure brings issues to parallelization. An example of the use of a `dchain` for this purpose breaking semantics in a parallel scenario is illustrated in fig. 3.3. In this example, we use a NAT running on two cores, each containing a `dchain` responsible for allocating ports used in the process of translation. The RSS in both NICs (bottom and top) is configured such that every packet that comes from the bottom NIC is sent to the same core as another packet that comes

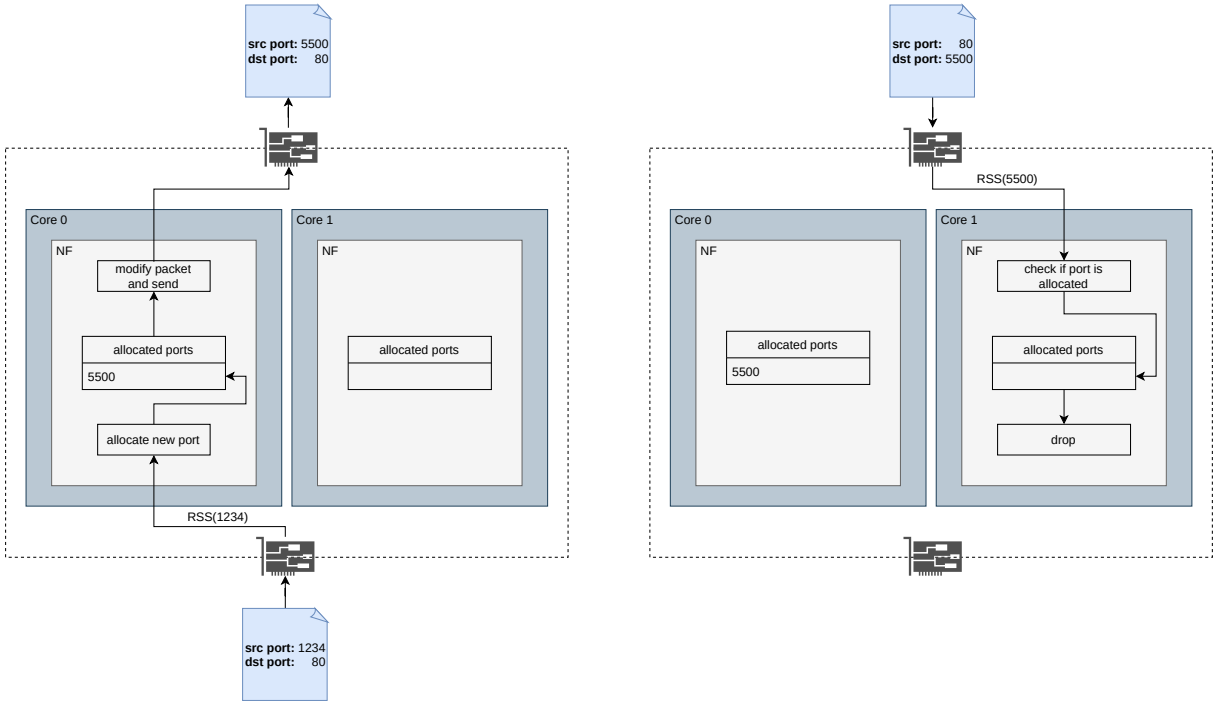


Figure 3.3: Example of `dchain` breaking parallel semantic equivalence.

from the top NIC if the source port of the former equals the destination port of the latter.

Upon the arrival of the first packet, in the left side, the NIC runs RSS against its source port, and redirects it to core 0. There, a new port is allocated, a common step in translation. This new port will be used by the returning response packet to address the private device hiding in the LAN (on the bottom). The `dchain` allocates the index 5500, which is then used as the new source port of the packet. It is then forwarded to the WAN.

Later, the response packet arrives, containing the previously allocated index as its destination port. The packet is handed to the NIC, which runs RSS using its destination port. Because its destination port differs from the source port of the previous packet (1234, and now 5500), it is possible that the resulting hashes will differ, redirecting the packets to different cores. That is exactly what happens, as the response packet is sent to core 1. In there, it checks if 5500 was already allocated in its `dchain`, which was not, resulting in the drop of the packet. However, we can see that 5500 was indeed allocated, but on the other core. On a sequential implementation, the response packet would not be dropped, as it would find that the index was allocated. Hence, semantic equivalence was broken.

When a packet field that contains data allocated in the `dchain` is used to configure RSS, it can break semantic equivalence. This is due to the fact that the core in which the index was allocated is not necessarily the same as the one in which it is checked the existence of that allocated index. For them to be equal, the packet that triggers the allocation would have to be steered to the same core that later

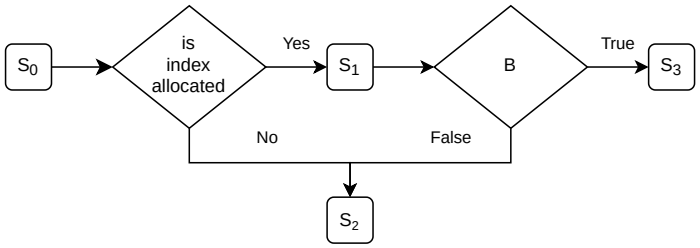


Figure 3.4: Flow chart of an NF that uses the constraint B to ensure parallel semantic equivalence when interpreting packet fields as `dchain` allocated indexes.

checks that same allocation. Because the user has no control over the generated values of the `dchain`, and the `dchain` implementation has no notion of the RSS mechanism, we say that the interpretation of `dchain` indexes as packet fields breaks parallel semantic equivalence.

In summary, *because we do not control the values allocated by the `dchain`, if the packet field that is passed as an argument to the function `dchain_is_index_allocated` is used to configure RSS, we can not guarantee that the packet that checks if the index was allocated is going to be sent to the same core as the packet that previously allocated the index*.

One solution for this problem is to reimplement the `dchain`, making it allocate indexes that, when given to the current RSS configuration, make the response packet be steered to the same core where this index was previously allocated. This would correspond to, using the example in fig. 3.3, allocating an index that would later, interpreted as a destination port field in the response packet, be given to RSS and make the packet be steered to core 0.

Another solution, and implemented by Maestro, is to try to find a constraint that allows bypassing this issue altogether. What is breaking semantic equivalence is the fact that, when a packet field is interpreted as a `dchain` index, it is possible to find it not allocated in a parallel implementation (because we are checking in the wrong core) when it otherwise would be found allocated in the sequential implementation. If it so happens that even if the NF fails to encounter a previously allocated index in a parallel scenario, both the sequential and parallel implementation are semantically equivalent, then this would stop being an issue.

Consider an NF whose flow chart is illustrated in fig. 3.4. It begins by running the code block denoted S_0. It then proceeds to check if the value in a specific packet field was previously allocated in the `dchain`. If the checking fails, it runs S_2, otherwise runs S_1. If, after running S_1, the condition B holds true, then it runs S_3. Otherwise, it runs S_2.

If we can find a condition B that is false every time the parallel implementation, contrary to its sequential counterpart, fails to find the allocated index in the `dchain`, then we could use this constraint B to redirect packets to the same core. We call this condition B as it allows us to bypass the problem that comes with interpreting packet fields as allocated indexes. Using B to build a constraint that packets in

the same core must satisfy assures us that, regardless of failing to encounter the index in the parallel scenario, the sequential and the parallel implementations are semantically equivalent (so long as S_1 does not alter NF state, modify, drop or forward the packet).

And so, when Maestro, in the LVA analysis step, encounters an LVA associated with a `dchain`, checks if its allocated indexes are used to fill packet data, or if packet data is checked to exist as allocated indexes. Based on this information, it takes two possible actions:

1. If the indexes are never used to fill packet data and never used to check their existence as allocated indexes, the LVAs associated with this `dchain` are ignored, as they never introduce constraints to the parallelization challenge.
2. If the indexes are interpreted as packet fields, then it tries to find a constraint in the list provided by the call paths that serves the same purpose of the mentioned constraint B. When this constraint is found, all the LVAs associated with this `dchain` are deleted, as they are no longer taken into consideration, and B is added later to the list of generated constraints (section 3.3.3). Failing to find this extra constraint results in a failure, and Maestro warns the user that it is unable to parallelize the NF without resorting to locking mechanisms.

3.3.3 Constraint generation

The next step in the Constraints Generator's pipeline is to, finally, generate the constraints. These constraints, representing the conditions that packets arriving to the same core must satisfy, will be then handed to the Key Generator.

To generate these constraints, Maestro loops over the filtered LVAs, creating a new expression for each pair of LVAs associated with operations to the same object—*i.e.*, LVAs that share an equal `object` field. This new expression is an equality between the two `read` expressions stored in the LVAs. If these two expressions are equal, then the packet must be processed in the same core.

Additionally, using the list of packet dependencies present in each LVA, it translates the `packet_chunks` symbols in the `read` expressions to bit-extract operations of a symbol that the Key Generator uses as the RSS hash input. As will be explained in the next section, the constraints given to the Key Generator must be between RSS hash inputs, and not between Vigor packet chunks. Moreover, considering that some LVAs may contain multiple possible values for specific packet fields, the constraints are generated as combinations of all the possible values of packet fields.

3.4 Key Generator

The previous module in the Maestro pipeline is responsible for deriving the specific constraints that all the packets arriving to the same core need to fulfill for the parallel execution to share the same semantics

```
0x6d  0x5a  0x56  0xda  0x25  0x5b  0x0e  0xc2
0x41  0x67  0x25  0x3d  0x43  0xa3  0x8f  0xb0
0xd0  0xca  0x2b  0xcb  0xae  0x7b  0x30  0xb4
0x77  0xcb  0x2d  0xa3  0x80  0x30  0xf2  0x0c
0x6a  0x42  0xb7  0x3b  0xbe  0xac  0x01  0xfa
0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x00  0x00  0x00  0x00
```

Figure 3.5: Example of an RSS key.

as the sequential one. This module tackles the next challenge—*how can one configure RSS to always send packets that satisfy these constraints to the same core*.

The indirection table of the RSS mechanism provides a direct map between the hash value and the cores. Typically, an indirection table has more entries than the number of available cores in a machine, and so, multiple entries of the table point to the same core. Mapping between these two concepts generally involves taking some number of least significant bits of the hash to index the table.

However, the challenge can be simplified. Instead of searching for an RSS configuration that steers packets that satisfy some given constraints to the same core, Maestro instead searches for a configuration that always outputs the same hash when these constraints are satisfied. The different is subtle, but simplifies the problem greatly at the cost of possibly shrinking the space of possible solutions.

For any pair of packets that generate different RSS hashes, it is not guaranteed that they will both be steered to the same core. It is possible, as both hashes, when used to index the indirection table, can arrive at entries storing the same value. But it is not guaranteed to happen. By requiring the hashes to be equal, it is guaranteed that both packets will arrive at the same core, as the same value will be used to index the indirection table, therefore arriving to the same value.

In order to understand how Maestro configures RSS to achieve equal hashes on specific packets, we must first understand in detail how the hash function works.

3.4.1 RSS hash function

The RSS toeplitz-based hash function, as explained in the previous chapter, can be configured—before the NF begins to process packets—with both a key and set of packet fields. It receives as input a bit-vector of appended packet fields. The output, *i.e.* the hash, is always 32 bits long. During the algorithm, the bit-vector created by appending the packet fields is iterated over bit by bit. When the traversed bit is 1, the hash is xor'ed against the least significant 32 bits of the key. The key is left-rotated by one bit in every iteration. This leads to a length constraint in the key, as it must have as many bits as the whole hash input + 31 bits.

As a practical example, let k assume the value represented in fig. 3.5. Additionally, consider a scenario where a specific packet arrives at the NIC with the properties shown in table 3.2.

Packet field	Value	Value in hexadecimal
IPv4 source address	66.9.149.187	0x420995bb
IPv4 destination address	161.142.100.80	0xa18e6450
TCP source port	2794	0x0aea
TCP destination port	1766	0x06e6

Table 3.2: Values of the packet fields used to configure RSS in a packet example.

The RSS mechanism running on the NIC, given that it was previously configured to extract both IPv4 addresses and both TCP ports, will generate the following input (d) and output ($h(k, d)$):

$$d = \underbrace{\texttt{0x420995bb}}_{\text{IPv4 src}}\overbrace{\texttt{a18e6450}}^{\text{IPv4 dst}}\underbrace{\texttt{0aea}}_{\text{TCP src}}\overbrace{\texttt{06e6}}^{\text{TCP dst}} \qquad h(k, d) = \texttt{0x51ccc178} \tag{3.1}$$

The value of the hash will then be used to index the indirection table, and the retrieved value will be the queue index associated with a specific core. The NIC will then insert this packet into that queue.

For simple constraints between packets, it is possible to arrive at a satisfying RSS configuration solely by correctly choosing the packet fields set. Again, this is better grasped via an example.

Let us imagine an NF that, in order to maintain semantics, required packets with equal IPv4 source and destination addresses to be sent to the same core. By configuring the RSS mechanism to use both of these fields, one can be certain that arriving packets, if they satisfy this constraint, will generate the same hash, and therefore be sent to the same core. The RSS input, generated by appending both these fields, will be exactly the same for both packets if they share the same values. Because of the non-stochastic nature of RSS, equal inputs generate equal hashes.

In the mentioned example, the correct configuration is chosen by correctly selecting the packet fields that ought to be used by the RSS mechanism. The key can be randomly generated, and only effects packet core distribution in this scenario (this is analyzed in detail later in this document).

Although this simple solution is useful to many NFs, there are, however, some scenarios that need addressing:

1. What if the NF requires a subset of packet fields that can only be used as a group in the RSS mechanism? One can not, for example, use only IPv4 source addresses, as RSS must be configured with both the source and destination addresses.
2. What if the NF requires complex constraints between packets? An example of a more complex constraint would be *if a packet's TCP source port equals another packet's TCP destination port, they need to generate the same hash*.
3. What if there are constraints between packets arriving in different ports? It is common for NICs to have multiple ports, and these ports' RSS can be individually configured. Moreover, the constraints can go even further and relate packets between different NICs.

To address all these scenarios in a generalized manner, we built a C library—libR3S [44]—capable of taking constraints as inputs and outputting RSS configurations that satisfy the given constraints. Maestro uses this library to generate RSS configurations that satisfy the constraints given by its Constraints Generator module. The development of a library independent of Maestro allows developers to find suitable RSS configurations without having to use Maestro and all its associated frameworks (Vigor and DPDK).

We find these suitable configurations by resorting to the Z3 solver. In order to find keys that satisfy the necessary conditions, libR3S encodes the problem in a logical format—the statement—and gives it to the solver, who is then responsible for finding a solution for the problem (not finding a solution is also valuable information).

The next section will describe the process of formulating the problem in a logical statement to be read by the solver.

3.4.2 Building the statement

Z3, an SMT (Satisfiability Modulo Theories) solver, is responsible for finding solutions to logical formulas. These logical formulas can include booleans, bit-vectors, and arithmetic and logical operations. In order to use the solver's capabilities to find RSS configurations that satisfy given constraints, one must first build the statement to be given to Z3.

By the end, we want the query given to Z3 to encode the following problem—*for a given set of constraints between packets, find a set of RSS keys that always send packets that satisfy these constraints to the same core*. In order to build this statement, we need to encode both the RSS hash function and the given packet constraints into an SMT format. As such, and to facilitate notation, we first introduce a few relevant concepts. Let:

L	be the hash function's key length in bits	(3.2)
k	be the hash function's key, a bit-vector with L bits	(3.3)
d	be the bit-vector of packet fields given to the hash function as input	(3.4)
$h(k, d)$	be the toeplitz-based hash function in fig. 2.4	(3.5)
o	be the 32-bit hash function's output	(3.6)
$H(k, k', d, d')$	be a statement that is true iff $h(k, d) = h(k', d')$	(3.7)
$C(d, d')$	be the constraints between the RSS hash inputs d and d'	(3.8)

We here represent $C(d, d')$ as the function that gives the needed constraints between packets. It

stands to reason that it should receive both d and d' as arguments, as both are generated from packets. As such, it is not possible to formulate a constraint that uses packet fields not captured by RSS. This does not represent a problem, as the battle was already lost upon the failure of capturing the fields by the RSS. If one can not capture the required fields, then it is impossible to use RSS to orchestrate packets to the correct cores.

Let us reuse the same example presented in section 3.4.1, now to build the constraint C. In the end, we want to encode the following problem into a query for the solver—*every pair of packets with the same IPv4 source and destination addresses should generate the same RSS hash*. This means that C should represent an equality of the source and destination addresses between a pair of packets. Upon configuring RSS to capture both IPv4 addresses, we find that the first 32 bits of the RSS input will correspond to the IPv4 source address, and the second 32 bits to the destination address. Representing an equality between the source addresses of a pair of packets corresponds to an equality of the first 32 bits of both the RSS inputs generated from those packets. The same can be said regarding the destination address, only now using the second 32 bits of the inputs. Therefore, the constraint *every pair of packets with the same IPv4 source and destination addresses should generate the same RSS hash* is represented as:

$$C(d, d') := (d[0:31] = d'[0:31]) \wedge (d[32:63] = d'[32:63]) \tag{3.9}$$

The constraint represented in eq. (3.9) is only truthful if the first 32 bits of d equal the first 32 bits of d', and the second 32 bits of d also equal the second 32 bits of d'. Because RSS was configured, in the mentioned example, with the fields $\{$IPv4 source address, IPv4 destination address$\}$, the RSS input will correspond to a 64 bit value, whose first 32 bits correspond to the first field, and the second 32 bits to the second.

This specific translation from a prose-constructed constraint to a logical SMT-compatible constraint exemplifies how these translations can be made. If a constraint encapsulates equalities of packet fields between a pair of packets (as it did in the given example), it can be represented as equalities between bits of the hash inputs d and d' (as in eq. (3.9)).

We now have the necessary information for understanding what a constraint $C(d, d')$ is. However, there is also a need to translate the RSS hash into a set of logical formulas that can be interpreted by the solver, which involves unrolling the loop in the hash algorithm.

As can be extracted from toeplitz-based hash algorithm, each iteration of the hash can involve a *xor* operation against the hash output and the left-most 32 bits of the key. Additionally, the RSS key is left-rotated by 1 bit in each iteration. If we were to ignore the branching in the algorithm and consider that the *xor* happens at all times, then the first bit of the output would correspond to the *xor* between all

the first 32 bits of the key. Similarly, the second bit of the output would correspond to the *xor* between all the left-most 32 bits of the key starting by the second bit. This pattern repeats until the output has 32 bits.

The only part missing from the output is the branch condition. The *xor* only happens in case the input bit that is currently being accessed has the value of 1. This condition can be taken into consideration by taking advantage of the fact that the value 0 is neutral in the *xor* operation, and therefore not tempering the result. As such, each bit of the output should not simply be the *xor* between bits from the key, but between logical *and*s that involve the bit of the key and the bit of the input. If the input is 1, then the *xor* takes effect, and it does not otherwise.

As such, the relation between the toeplitz-based hash function and its output, *i.e.* $h(k, d) = o$, can be represented as the following system of equations:

$$\begin{bmatrix} o[0] \\ o[1] \\ \vdots \\ o[31] \end{bmatrix} = \begin{bmatrix} (d[0] \wedge k[0]) \oplus (d[1] \wedge k[1]) \oplus ... \oplus (d[|d|-1] \wedge k[|d|-1]) \\ (d[0] \wedge k[1]) \oplus (d[1] \wedge k[2]) \oplus ... \oplus (d[|d|-1] \wedge k[|d|]) \\ \vdots \\ (d[0] \wedge k[31]) \oplus (d[1] \wedge k[32]) \oplus ... \oplus (d[|d|-1] \wedge k[|d|+30]) \end{bmatrix} \tag{3.10}$$

Therefore, as H represents the equality between a pair of hash outputs, it can be defined by the following expression:

$$H(k, k', d, d') := \bigwedge_{b=0}^{31} \left[\bigoplus_{x=0}^{|d|} (d[x] \wedge k[x+b]) = \bigoplus_{y=0}^{|d'|} (d'[y] \wedge k'[y+b]) \right] \tag{3.11}$$

We now have all the necessary pieces of build the needed SMT-solver-compatible statement. We will build the statement by iteratively increasing its complexity.

Base statement. Initially, the statement should represent the following query—*find a single key* k *such that, given any two hash inputs* d *and* d' *that obey the constraints* C*, the corresponding hashes will always be equal*. That is:

$$\forall_{d,d'} \, . \, (C(d, d') \wedge d \neq d') \rightarrow H(k, k, d, d') \tag{3.12}$$

As represented in eq. (3.12), we ignore hash inputs that are equal. This is an optimization, as giving the same input to the hash function returns the same hash value. It is guaranteed that a pair of equal packets generate the same hash, and therefore are sent to the same core. As such, we do not need the solver to consider equal hash inputs.

If a solver, when given this statement and a specific set of constraints $C(d, d')$, returns a value for the key, then we are certain that this returned key will be such that, when a pair of packets that satisfy the given constraints arrive at the NIC configured with this key (and the proper packet fields), both packets

will be sent to the same core.

Compatibility with multiple keys. Currently, the statement is only taking into consideration a single key. That is, the hash is calculated against the pair of hash inputs using the same key. However, an NF can have different behaviors depending on the specific NIC's port that received the packet (the same logic can be applied for ports on different NICs). As such, there may exist a necessity for constraints between different ports in order to maintain semantics on the parallel execution. Each port can have its RSS mechanism individually configured, and therefore the statement must take into consideration multiple keys—one for each port in each NIC.

In order to take multiple keys into consideration, the constraints must be specific not only to each pair of packets arriving from the same port, but also to every pair of packets arriving from different ports. The following pair of constraints represent an example of this multi-key awareness requirement—*for every pair of packets that arrive at port 1, if their IPv4 addresses match then their RSS hashes must also match; additionally, for every pair of packets between the ports 1 and 2, if their TCP/UDP ports match then their RSS hashes must also match*.

Let $C_{i,j}$ be the constraint between a pair of packets coming from ports i and j, which were previously configured with the RSS keys k_i and k_j, respectively. In this example, k_1 corresponds to the key used to configure RSS in the port 1, and k_2 used to configure RSS in port 2. $C_{1,1}$ would, therefore, correspond to the constraint between a pair of packets arriving at port 1, and would encode *equal IPv4 addresses*. Moreover, the second constraint in the multi-key example, as it refers to a relation between a packet that arrives at the port 1 and another that arrives at port 2, would correspond to a constraint $C_{1,2}$. $C_{1,2}$ would, therefore, encode *equal TCP/UDP ports*.

This example only uses the constraints $C_{1,1}$ and $C_{1,2}$. It imposes no constraint between pairs of packets both arriving from port 2 ($C_{2,2}$). Although it is not necessary here, we can still enumerate all the combinations of ports to which we can associate a given constraint. Equation (3.14) contains all the possible constraints that can be defined when using only 2 ports.

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} \tag{3.13}$$

Note that defining the constraint between packets arriving from ports 1 and 2 is equal to defining the constraint between packets arriving from ports 2 and 1, as the order is irrelevant. As such, $C_{1,2} = C_{2,1}$, and there is no need to define all the constraints on this matrix. It is enough to consider, for example, all the constraints $\{C_{i,j} : i \leq j\}$. Lets us define F as the constraint that is always false. Using F as a constraint allows the solver to ignore that specific constraint between ports, as it only needs to send packets that follow the constraints to the same core. If F is, by definition, always false, then no pair of packets will satisfy it. Therefore, eq. (3.14) becomes:

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ F & C_{2,2} \end{bmatrix} \tag{3.14}$$

Although the given example only uses 2 ports, we need to generalize for any n ports. As such, let us denote $K = \{k_1, k_2, \ldots, k_n\}$ as the set of all RSS keys that configure each port. The constraints matrix containing all constraints between packets arriving from all the combinations of ports is represented in eq. (3.15).

$$\begin{bmatrix} C_{1,1} & C_{1,2} & C_{1,3} & \ldots & C_{1,n} \\ F & C_{2,2} & C_{2,3} & \ldots & C_{2,n} \\ F & F & C_{3,3} & \ldots & C_{3,n} \\ \vdots & \vdots & \vdots & & \vdots \\ F & F & F & \ldots & C_{n,n} \end{bmatrix} \tag{3.15}$$

The solver, when given the statement in eq. (3.12), finds a single key that makes all the packets that follow its given constraint generate the same hash output. In order to make this statement multi-key aware, every constraint in the constraint matrix in eq. (3.15) must be taken into equal consideration. This can be done by chaining these constraints together with logical *and*s. This allows the solver to manage each key combination problem as a specific statement that must be truthful, and also take into consideration the dependencies between the keys. As such, the statement then becomes:

$$\forall_{d,d'} \cdot \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} [(C_{i,j}(d, d') \wedge d \neq d') \rightarrow H(k_i, k_j, d, d')] \tag{3.16}$$

Compatibility between different sets of chosen RSS packet fields. As it was demonstrated, there are possible NFs whose semantic equivalence in a parallel execution is bound to constraints between packets from ports configured with different RSS keys. The same can be said regarding different set of chosen RSS packet fields. Again, to exemplify such a scenario, a hypothetical constraint can be used—*for every pair of packets that arrive at the ports 0 and 1, if the TCP ports of the packet arriving to port 0 match the UDP ports of the packet arriving to port 1 then their RSS hashes must also match*. Admittedly, this is an unusual and—expected to be—infrequent scenario. However, as it is possible, it must be taken into consideration. Because these constraints are automatically extracted from the NF's code, it bodes well for the Key Generator module to be capable of dealing with such unusual scenarios.

Each packet field, as was illustrated in eq. (3.1), corresponds to a set of bits in the RSS hash input. This mapping between the packet fields and the hash input is only done for the set of packets fields present in the RSS configuration. Let Γ_i be the set of all chosen RSS packet fields' options in port i. Additionally, let $\gamma \in \Gamma$ be a RSS packet fields' option. In order to better understand these new concepts, consider the following Γ:

$$\Gamma_1 = \{\{\text{IPv4 src}, \text{IPv4 dst}\}, \{\text{IPv4 src}, \text{IPv4 dst}, \text{TCP src}, \text{TCP dst}\}\} \tag{3.17}$$

In this case, Γ_1 corresponds to the set of all sets of chosen packet fields (the RSS packet fields' options) in port 1. When a packet arrives at the NIC, the RSS mechanism tries to match the packet to one of these available options:

1. If the packet uses IPv4 with fragmentation, then the first option is matched, and the IPv4 source and destination fields are used to produce the hash input.
2. If the packet uses TCP/IPv4, then the second option is matched, and the fields IPv4 source and destination and TCP source and destination are used to produce the hash input.
3. Otherwise, the packet does not undergo the RSS algorithm. Typically is sent to either a default core, or a randomly picked one.

Each different matched option in each port will generate a specific hash input, varying not only its value but also its length (some options use more fields than others). In the statement given to the solver, the notion of all the possible pairs of hash inputs is represented by the use of the universal quantifier. One is lead to believe that RSS configurations using different Γs require, at first glance, populating this quantifier with more symbols—one for each hash input associated with a specific RSS option (γ)—. This, however, highly increases the complexity of the query.

The solution to this problem can be found by understanding that considering all the possible values that a symbol can acquire is equivalent to considering all possible values that the symbol's partitions can acquire. In that regard, d and d' become bit-vectors of the maximum hash input length between all the possible hash inputs in the given Γs. In eq. (3.17), that would correspond to a hash input of 96 bits, as it is the biggest that can be created with Γ_1. For each γ in each of the given Γs, we extract as many least significant bits from d as one needs to form a γ specific hash input (and analogously for d').

This extraction is represented by the δ notation. Considering that d has as many bits as the maximum hash input that can be generated for the given Γs, then δ^d_γ would correspond to the extraction of the necessary number of bits from d for the generation of the hash input associated with γ. Using again eq. (3.17) as an example:

$$\Gamma_1 = \{\gamma_0, \gamma_1\} \quad \gamma_0 = \{\text{IPv4 src}, \text{IPv4 dst}\} \quad \gamma_1 = \{\text{IPv4 src}, \text{IPv4 dst}, \text{TCP src}, \text{TCP dst}\} \tag{3.18}$$

$\delta^d_{\gamma_0}$ represents, in this example, an extraction of the first 64 bits of d (32 bits for each IPv4 address), and $\delta^d_{\gamma_1}$ an extraction of the first 96 bits of d (32 bits for each IPv4 address and 16 bits for each TCP port).

```
0x80  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x00  0x00  0x00  0x00
```

Figure 3.6: RSS key that leads to bad core distribution.

The statement now needs to consider constraints between packets arriving at different ports using different keys and matching different RSS packet field options. And so, using the presented notation, eq. (3.16) becomes:

$$\forall_{d,d'} \cdot \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \bigwedge_{\gamma \in \Gamma_i} \bigwedge_{\gamma' \in \Gamma_j} \left[(C_{i,j}(\delta_{\gamma}^{d}, \delta_{\gamma'}^{d'}) \wedge \delta_{\gamma}^{d} \neq \delta_{\gamma'}^{d'}) \rightarrow H(k_i, k_j, \delta_{\gamma}^{d}, \delta_{\gamma'}^{d'}) \right] \tag{3.19}$$

Discarding trivial solutions. One solution that is guaranteed to always satisfy eq. (3.19) corresponds to a set of all the keys with all the bits with the value 0. These keys, under the RSS algorithm, will always produce a hash output of, also, 0. This, however, is not useful in the scope of this project. The RSS is primarily used to orchestrate packets to the available cores. If the hash output is constant, then the value retrieved from the indirection table is always the same, and all the packets will be redirected to the same core. And so, to discard these trivial solutions, we add constraints to the possible values of all the keys, dictating that they must be non-zero:

$$\forall_{d,d'} \cdot \bigwedge_{i=1}^{n} k_i \neq 0 \wedge \left[\bigwedge_{j=1}^{n} \bigwedge_{\gamma \in \Gamma_i} \bigwedge_{\gamma' \in \Gamma_j} \left[(C_{i,j}(\delta_{\gamma}^{d}, \delta_{\gamma'}^{d'}) \wedge \delta_{\gamma}^{d} \neq \delta_{\gamma'}^{d'}) \rightarrow H(k_i, k_j, \delta_{\gamma}^{d}, \delta_{\gamma'}^{d'}) \right] \right] \tag{3.20}$$

3.4.3 Finding solutions

When the solver, upon given the statement eq. (3.20), returns a set of values for the keys, it must be true that they satisfy the given statement. The solver always returns the first set of satisfying solutions that it finds. That, however, may not be ideal.

Consider an NF configured with 1 port and $\Gamma_1 = \{\{\text{IPv4 src}, \text{IPv4 dst}\}\}$. When given the constraint *every pair of packets using fragmented IPv4 that arrive at port 1 and have the same IPv4 addresses must generate the same hash*, the solver finds the key represented in fig. 3.6.

Although the found key satisfies the given constraint—this happens to be one of the more simple scenarios mentioned in the beginning of this section where simply choosing the right packet fields is enough to achieve a satisfying key—it has only one bit with the value 1 (the very first one, as `0x80` = `0b10000000`). Because all the other bits are 0, the RSS hash output can only be either `0x00000000` or `0x80000000`, limiting the maximum number of cores that can receive packets to just two. Moreover,

the only bit of the hash input that influences this decision is the first one, typically associated with the network number of the IP and expected to vary much less than the least significant bits of the IP (the subnet bits).

The solution resides in asking the solver to give the value 1 to as many bits as possible in the keys, so long as they still satisfy the given statement. This constitutes a Partial MAXSAT problem [45]. We give the solver a statement that its corresponding solutions should always satisfy (eq. (3.20))—also known as hard constraints—and also a set of clauses that they should try to satisfy—soft constraints. These new clauses, the soft constraints, are composed of equalities between key bits and the value 1.

There is no absolute need for maximizing the number of satisfied soft constraints (*i.e.* the number of bits of the key with the value 1). Most of the times, a randomly selected set of bits with the value 1 is sufficient to avoid the corner case problems mentioned above. As such, Maestro uses a slightly modified version of the diagnosis based approached introduced by Fu and Malik [46]. It begins by seeding the key with random bits. Then, when (probably) the solver deems both the hard and soft constraints unsatisfiable as a whole, it generates an UNSAT core containing the set of clauses which constitute the root cause of this unsatisfiability. Maestro randomly discards a subset of these soft constraints, and asks the solver again to find a satisfying set of keys. The process terminates when either the soft constraints run out, or a set of satisfying keys are found.

Because of the stochastic nature of this method, libR3S provides the option to run multiple independent instances of the solver in parallel. Each independent run seeds the key with a (probably) different random set of bits, and possibly finds a different set of satisfying keys. The user can also provide a traffic sample to the library, which is used to test the keys found by the solver. The distribution test involves calculating the hash for every packet in the traffic sample, and subsequently the core to which the packets are sent. Using the number of packets sent to each core, it then calculates the corresponding standard deviation. If the standard deviation is lower than a user provided threshold, then the key passes the test. When all keys pass this test, all the processes are terminated, and the set of keys is retrieved to the user. Otherwise, the process is killed, and another one is spawned in its place.

Symmetric RSS. As exposed in the previous chapter, Woo and Park presented a formula to generate RSS keys that send packets from the same TCP connection do the same core. A pair of request/response packets that belong to the same TCP connections have their respective IP addresses reversed, as well as reversed TCP ports. This is analogous for a UDP connection.

Their formula consists of three conditions (note that we use the notation $k[A:B]$ to denote all the k bits between, and including, A and B):

1. $k[0:14] = k[16:30] = k[32:46] = k[48:62] = k[64:78] = k[80:94] = k[96:110] = k[112:126]$
2. $k[15] = k[47] = k[79] = k[95] = k[111]$
3. $k[31] = k[63]$

They also presented a key that satisfies these constraints:

```
0x6d  0x5a  0x6d  0x5b  0x6d  0x5a  0x6d  0x5b
0x6d  0x5a  0x6d  0x5a  0x6d  0x5a  0x6d  0x5a
0x6d  0x5a  0x6d  0x5a  0x6d  0x5a  0x6d  0x5a
0x6d  0x5a  0x6d  0x5a  0x6d  0x5a  0x6d  0x5a
0x6d  0x5a  0x6d  0x5a  0x6d  0x5a  0x6d  0x5a
```
(3.21)

When queried to find an RSS key that satisfies these constraints, libR3S found the key shown in eq. (3.22), which also satisfies Woo and Park's formula.

```
0x50  0x04  0x50  0x05  0x50  0x04  0x50  0x05
0x50  0x04  0x50  0x04  0x50  0x04  0x50  0x04
0x74  0x5b  0x9a  0xe5  0xc8  0x29  0x10  0xe6
0x55  0x32  0x87  0xda  0x6e  0xf0  0x6a  0xfb
0x18  0x99  0x64  0x5a  0xb7  0x68  0x32  0x34
0x47  0x2d  0x93  0xd7  0x37  0x25  0xaa  0xab
0x81  0x44  0x90  0x49
```
(3.22)

Note that eq. (3.21) and eq. (3.22) have different lengths. This is a NIC-dependent detail, as some NICs can deal with packet fields' options that generate bigger hash inputs.

The code used to generated this key is shown in listing A.2.

3.5 Synthesizer

After the Key Generator stage, the user has—hopefully, as it can be impossible for some NFs—access to an RSS configuration that allows parallelizing its NF without the use of locking mechanisms. The step that naturally follows is to configure both the NF to run on multiple cores, and the RSS mechanism with the discovered keys and packet fields.

The Synthesizer stage is not fundamentally necessary for acquiring a parallel implementation. However, it is convenient, as it provides the user a push-to-parallelize method. Moreover, it can be seen as a proof-of-concept that Vigor can be used to automatically generate modified versions of any NF written in its framework.

This stage receives as input the found RSS configuration and the call paths associated with the NF, and outputs a single C file corresponding to a parallel implementation of the NF.

The first step of this stage is to build a binary decision tree that represents the NF, whose nodes correspond to either calls to packet management functions, calls to libVig functions, or conditional statements. Every time a conditional node is added to the tree, it bifurcates into two subtrees, one corresponding to the flow of the program in which the condition is true, and the other to the scenario where the condition is false. Algorithm 3.1 demonstrates how this process is done.

To build the tree, Maestro iterates over the calls on all the call paths at the same rate. If all the

Algorithm 3.1: Binary decision tree generation algorithm

```
build_BDD(CP)
    i ← 0
    while CP > 0 do
        filtered ← has_at_least_n_calls(CP, i + 1)
        call ← filtered[0].calls[0]
        eq ← []
        diff ← []
        for cp ∈ CP do
            if cp.calls < i or cp.calls[i] ≠ call then
                diff ← diff + cp
            else
                eq ← eq + cp
        if diff = 0 then
            node ← node_from_call(call)
            BDD.add_node(node)
            CP ← filtered
            i ← i + 1
            continue
        c ← find_discriminating_constraint(eq, diff)
        condition ← node_from_constraint(c)
        then ← build_BDD(eq)
        else ← build_BDD(diff)
        BDD.add_if_then_else(condition, then, else)
        return BDD
    return BDD
```

call paths share the same call, then a new node corresponding to this call is added to the tree. If a call is different in some call paths, then Maestro tries to find the single constraint that differentiates the two groups of call paths. We are certain to find a single constraint differentiating the groups, as this difference between groups is caused by the existence of an `if` statement in the NF's code. After finding the discriminating constraint, Maestro makes two recursive calls to this tree building function, one building a subtree involving the call paths that satisfied the constraints, and the other building a subtree involving the call paths that did not satisfy the constraints. When both subtrees are finally generated, they are both appended as branches to the original tree. The algorithm terminates when there are no more calls in all the call paths.

If the call paths example from the previous chapter (listing 2.3) were to be given as input to the Synthesizer, the corresponding generated binary decision tree would be as illustrated in fig. A.2. When starting the algorithm, the Synthesizer would find a call to the function `is_tcp` in every call path. Therefore, it would add this call to the tree. Subsequently, it would find that the next call varies throughout the call paths—one call path calls the function `drop`, while others call `map_contains`. Knowing this, it would try to find the constraint that discriminates these two groups of call paths, and would find that the call path that drops the packets does not satisfy the constraint `is_tcp_result`, while the other call paths do.

It then would recursively call the build function two times, each following one of the branches containing the corresponding call paths.

Following the branch that does not satisfy the constraint, the Synthesizer sees that only a single function is called—`drop`. It adds this call to the current subtree, and terminates this branch of recursion. This subtree is captured by the original call to the build function, which adds it to the original tree (containing only a node representing the call to `is_tcp`).

The other recursive instance generates all nodes associated with calls that are equally shared among the call paths, only stopping when encountering the first change—some call paths call the function `map_contains`, while others call `map_put`. Again, it finds the discriminating constraint (`map_contains_0`), and recursively calls two times. This process continues until no more calls remain.

The next step is to traverse this tree and generate C code. This process of translation is very straight forward, as the initial code is already in C, and therefore there is no translation between different languages or platforms. The generated C code will, however, be different from the original implementation, as Maestro will allocate one instance of the libVig data structures in each core, configure RSS with the given key and packet fields, and use a modified version of Vigor that runs in every core.

3.5.1 Parallel implementation with locking mechanisms

Albeit not ideal, Maestro can be configured to generate parallel implementations that use locking mechanisms to ensure semantic equivalence between the sequential and the parallel implementations. This method is necessary for parallelization in case Maestro is unable to find a way to configure RSS to achieve a shared-nothing architecture.

In a lock-based parallel implementation, RSS is configured both with a random key, and with all the available RSS-compatible packet fields. In this case, RSS is not used to orchestrate packets to certain cores in order to maintain semantics, but as a load-balancing mechanism.

Contrary to the shared-nothing model, Maestro needs to make a distinction between read-packets and write-packets. Read-packets lead to read operations on the NF's state, and write-packets trigger writes to said state. Because there is no guarantee that packets that access the same state arrive always at the same core, we assume that both read- and write-packets that access the same state can arrive at the same time to different cores, therefore performing operations on the same state in parallel.

Parallel read-packets do not harm the NF's semantic equivalence, as they make no change to its state. However, write-packets need to be controlled in such a way that prevents them from both writing to the same state at the same time, and additionally writing to the same state that is being currently read. Moreover, individually protecting parallel writes to the same data structures is not enough. The state associated with an NF typically involves multiple data structures. Interrupting the process of a write-packet would lead to partial changes to its state, possibly leading to erroneous behavior, as only a

Algorithm 3.2: Lock API

```
    Input: Write permissions array W, number of pending writes p, write spinning lock s, thread ID id
allow_writes(W, p, s, id)
    W[id] ← ALLOW
block_writes(W, p, s, id)
    W[id] ← BLOCK
    if p = 0 then
        return
    W[id] ← ALLOW
    while p > 0 do
        busy_waiting()
    W[id] ← BLOCK
write_lock(W, p, s, id)
    atomic_increment(p)
    W[id] ← ALLOW
    success ← false
    while success = false do
        success ← true
        foreach tid ∈ Threads do
            if W[tid] = BLOCK then
                success ← false
                break
    lock(s)
write_unlock(W, p, s, id)
    unlock(s)
    atomic_decrement(p)
```

subset of the NF's state would reflect the arrival of that packet. There is, therefore, a need for processing write-packets sequentially, whilst preventing read-packets to be processed during this process.

Therefore, we need to generate lock-based implementations that distinguish between read-packets and write-packets, and only allow write-packets to be processed sequentially and not at the same time as read-packets. With this in mind, Maestro generates a modified version of the packet processing loop in Vigor. This modified loop is shown in algorithm 3.3, using a custom made locking API represented in algorithm 3.2.

In the modified packet processing loop, we first assume the packet to be a read-packet. As write-packets can not be processed at the same time as read-packets, writes are blocked using the function `block_writes`, and the packet is processed. However, if a write is attempted, the processing function signals the loop by writing to a global flag ($attempt$), indicating that a write attempt was made during a read state. In that case, the packet becomes a write-packet, atomically increments a global counter that stores the number of pending write-packets (p), and waits for all the reads to finish (`write_lock`). Before a read-packet is processed, it checks if the counter of pending writes is zero (in `block_writes`). If it is, then it proceeds to being processed, indicating it through a global flag (an entry in the write permissions array W). On the other hand, if there are pending writes, then it waits for the counter to be zero, allowing

Algorithm 3.3: Packet processing loop in a lock-based parallel implementation using the lock API in algorithm 3.2

Input: Packet pkt, global write attempt flag $attempt$, global read/write state flag $state$, write permissions array W, number of pending writes p, write spinning lock s, thread ID id

```
while true do
    attempt ← false
    state ← READ
    block_writes (W, p, s, id)
    process(pkt)
    if attempt = true then
        state ← WRITE
        write_lock (W, p, s, id)
        process(pkt)
        write_unlock (W, p, s, id)
    else
        allow_writes (W, p, s, id)
```

other packets to perform writes without starvation. When no reads are happening, the write packets try to acquire the write lock (s, in `write_lock`). After acquiring the lock, the packet is processed, and finally the lock is released (`write_unlock`). In case the packet did not attempt to make writes, then it was correctly considered as a read-packet in the beginning, and writes are allowed when its processing is finished (`allow_writes`).

If an NF is subjected to mainly read-packets, the performance toll is minimized, as read-only workloads do not need to acquire any locks, atomically write to any shared variable, or write to shared data. They only need to signal that a read is happening, but DPDK provides the tools to allocate memory that is cache-friendly. Each read-packet only writes to its designated cell in the array, and therefore the cache coherence overhead associated with traditional read/write locking mechanisms is avoided.

Moreover, we can expect no starvation, as all write-packets are originated from read-packets, and therefore it is not possible for read-packets to never be processed because write-packets are always appearing. However, write-packets are expected to be associated with a costly performance toll, which may be amortized for intensive read workloads.

4

Evaluation

Contents

In this chapter, we evaluate a set of parallel implementations generated by Maestro. We aim to provide the answer to five questions:

1. How much time does it take for Maestro to find a suitable RSS configuration that allows parallelization without the use of locking mechanisms?
2. How much time does it take for Maestro to generate a parallel implementation?
3. How much performance variation can we expect between different (but equally valid) RSS configurations, patterns of incoming traffic, and NUMA configurations?
4. How does the parallel implementation performance scale with the number of cores?
5. How much performance difference can be found between a parallel lock-free implementation and a lock-based parallel implementation?

In the first section of this chapter, we use NFs provided by Vigor to generate their corresponding parallel implementations, showing each constraint found by Maestro and the respective generated RSS configuration. This is used both as an example—as each of the provided NFs constitute a unique example in the process of parallelization—and as a mean to provide the answer to the first two questions mentioned above.

As questions 3 to 5 relate to performance, it is important to understand this concept in the context of NFs, and how is it measured. Performance in NFs is associated with two concepts: throughput—the number of packets processed per second—and latency, *i.e.*, the time a single packet takes to go to the NF and come back. Thus, the following section explains the methodology behind acquiring values for both throughput and latency throughout the evaluation.

Before benchmarking each of the parallel NFs found, we show how their throughput can be influenced by the chosen RSS keys and packet fields, packet skewing, and NUMA configurations.

Finally, and closing this chapter, we evaluate the performance of each parallel NF, comparing it to their respective sequential implementations, and answering the final two questions.

4.1 Microbenchmarks

At the time of this writing, Vigor provides six NFs—vignop, vigpol, vigfw, vigbridge, viglb, and vignat. In this section, we explain the utility behind each of these NFs, and show how Maestro can be used to parallelize them, when possible.

Although pseudo-code for each NF is provided, it is used as a mean to illustrate their utility. It is not complete, and does not incorporate all the features present in the NFs. Furthermore, state related data structures and calls to the libvig API are largely simplified. These simplifications, however, only omit details which do not influence the parallelization process. As such, the exposed generated RSS configurations are identical to the ones associated with the complete NFs.

Listing 4.1: Pseudo-code of vignop.

```
1 void process_packet(int device, pkt_t p) {
2     if (device == WAN) {
3         forward(p, LAN);
4     } else {
5         forward(p, WAN);
6     }
7 }
```

$$\begin{bmatrix} F & F \\ F & F \end{bmatrix}$$

Figure 4.1: Constraints matrix of vignop.

4.1.1 NOP

Vignop represents a no-operation (NOP) NF, *i.e.*, a stateless NF that simply forwards all the packets that arrive to one port to the other. It watches over two ports, LAN and WAN, and its pseudo-code is illustrated in listing 4.1. Maestro, through call path analysis, finds that this NF has no state. As such, there are no constraints between packets arriving at the same core. The vignop constraints matrix, generated by Maestro in 1 second, can be found in fig. 4.1, using the same notation as in eq. (3.15) (*i.e.*, a matrix with as many rows and columns as the number of available ports, and each with an entry corresponding to a constraint $C_{i,j}$ between the ports i and j).

As this NF maintains no state, RSS can be configured with all the packet fields options available. Maestro, using the found constraints matrix and all the available packet fields options, finds the RSS configuration represented in table 4.1. The generated RSS keys are completely random, as all the possible RSS keys lead to parallel implementations semantically equivalent to the sequential vignop. Maestro takes 4 seconds to synthesize a parallel implementation of vignop.

4.1.2 Policer

A policer aims to limit a specific user's download and/or upload rate according to a previously established contract. Vigpol, the Vigor's policer implementation configured with two ports (LAN and WAN), limits only the download rate, identifying a specific user's download by the IPv4 destination address of the packets coming from the WAN. Its pseudo-code can be found in listing 4.2.

When Maestro is run against this NF, it finds that all its state is accessed with the IPv4 destination address packet field (lines 13 and 14 in listing 4.2). This implies that packets with the same IPv4 destination address must be sent to the same core, as they access the same state. Therefore, Maestro generates the constraints illustrated in fig. 4.2.

As the policer's state is only accessed when a packet arrives from the WAN port, the only non-false constraint in the matrix is the one associated solely with the second port (WAN). Moreover, because the constraint uses the IPv4 destination address, the chosen RSS packet fields options must contain this

Port	Key	RSS packet fields options
LAN	0x6d 0x5a 0x56 0xda 0x25 0x5b 0x0e 0xc2 0x41 0x67 0x25 0x3d 0x43 0xa3 0x8f 0xb0 0xd0 0xca 0x2b 0xcb 0xae 0x7b 0x30 0xb4 0x77 0xcb 0x2d 0xa3 0x80 0x30 0xf2 0x0c 0x6a 0x42 0xb7 0x3b 0xbe 0xac 0x01 0xfa 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	All available
WAN	0x6d 0x5a 0x56 0xda 0x25 0x5b 0x0e 0xc2 0x41 0x67 0x25 0x3d 0x43 0xa3 0x8f 0xb0 0xd0 0xca 0x2b 0xcb 0xae 0x7b 0x30 0xb4 0x77 0xcb 0x2d 0xa3 0x80 0x30 0xf2 0x0c 0x6a 0x42 0xb7 0x3b 0xbe 0xac 0x01 0xfa 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	All available

Table 4.1: RSS configuration of vignop.

$$\begin{bmatrix} F & F \\ F & (d[\text{IPv4 dst}] = d'[\text{IPv4 dst}]) \end{bmatrix}$$

Figure 4.2: Constraints matrix of vigpol.

packet field. Although DPDK allows RSS configuration with an RSS packet field option containing only IPv4 addresses, this option is not compatible with our NIC (an Intel 10Gb X710). Therefore, the chosen packet field options not only contained the IPv4 addresses, but also the TCP/UDP ports. This changes the restriction in the keys found by Maestro, as they now have to not take the ports into consideration in order to send packet with the same IPv4 destination address to the same core.

The RSS keys and packet fields options associated with the parallel implementation of this NF were found by Maestro in 2 minutes and 31 seconds, and are shown in table 4.2. The first 127 bits of the RSS key associated with the LAN port are completely random, as there are no constraints associated with this port. Note that the remaining bytes of the LAN key are all 0, but this is irrelevant, as only the first 127 bits of the key are used in the hash.

Of the first 127 bits of the WAN key, only the 64^{th} has the value 1 . This is because the hash is capturing the TCP and UDP ports, which are associated with the second row of bytes in the key. Therefore, these bits must be 0, as TCP and UDP ports can not be taken into consideration when sending packets to the same core. The IPv4 source address has influence on the first 63 bits of the key, and therefore they also need to be 0. Essentially, because only one bit of the key has the value of 1 and is associated with the IPv4 destination address of the packet, the resulting RSS hash always turns out to be the value of the IPv4 destination address field of that packet.

Maestro takes around 3 minutes and 5 seconds to synthesize a parallel implementation of vigpol.

4.1.3 Bridge

A bridge associates MAC addresses with ports, and redirects packets to those ports accordingly. Vigbridge is Vigor's implementation of a bridge, and its pseudo-code is represented in listing 4.3.

Listing 4.2: Pseudo-code of vigpol.

```
struct Map* map;

void process_packet(int device, pkt_t p) {
    if (device == LAN) {
        // not policing outgoing packet
        forward(p, WAN);
        return;
    }

    struct RateChunk current = { size: p.size, time: now() };

    // incoming packet from WAN
    if (!map_contains(map, p.ipv4_dst)) {
        map_put(map, p.ipv4_dst, current);
        forward(p, LAN);
        return;
    }

    struct RateChunk past = map_get(map, p.ipv4_dst);
    int rate = get_rate(past, current);

    if (rate > MAX_RATE) {
        drop(p);
    } else {
        forward(p, LAN);
    }
}
```

$$\begin{bmatrix} F & F \\ F & F \end{bmatrix}$$

Figure 4.3: Constraints matrix of vigbridge.

The association between MAC addresses and ports is done dynamically by vigbridge, as it adds a new entry to the MAC table when a packet's MAC source address is not stored yet. Maestro, when run against this NF, realizes that in order to parallelize this NF it would need to configure RSS with MAC addresses. Unfortunately, RSS is not compatible with these fields. As such, Maestro returns to the user a message saying that it is unable to generate a parallel implementation because of the usage of RSS incompatible packet fields when accessing state.

However, we can use a slightly modified version of vigbridge that allows for RSS sharding. This modified version is represented in listing 4.4, and constitutes a static version of vigbridge. In this version, the MAC table is static and filled at configuration time. During the processing of packets, the NF never adds new entries, redirecting packets to their corresponding ports if a MAC is present on the static table, and flooding to all ports otherwise.

When run against this modified version, Maestro encounters a libvig data structure that is read but never written, and therefore all the possible RSS configurations will lead to semantic equivalence between the sequential and the parallel versions. It then deletes all the LVAs associated with the MAC table. Because the NF uses no other data structure to store state, there are no constraints bounding packets going to the same core, as represented in fig. 4.3.

Port	Key	RSS packet fields options
LAN	0x01 0x20 0x01 0x00 0x01 0x24 0x01 0x42 0x01 0x00 0x01 0x14 0x01 0x90 0x01 0x76 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	• TCP IPv4 (IPv4 src, IPv4 dst, TCP src, TCP dst) • UDP IPv4 (IPv4 src, IPv4 dst, UDP src, UDP dst)
WAN	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x36 0x41 0x95 0x96 0xf4 0x20 0x4f 0xdb 0x21 0xf3 0xfa 0x0d 0x0c 0x4e 0xa2 0xf6 0xfc 0x3f 0x53 0xc8 0x98 0x14 0xe0 0xdb 0xd1 0x41 0xbd 0x4f 0x78 0x2a 0x86 0xae 0x6b 0x1b 0x45 0x60	• TCP IPv4 (IPv4 src, IPv4 dst, TCP src, TCP dst) • UDP IPv4 (IPv4 src, IPv4 dst, UDP src, UDP dst)

Table 4.2: RSS configuration of vigpol.

Listing 4.3: Pseudo-code of vigbridge.

```
1  struct Map* map;
2
3  void process_packet(int device, pkt_t p) {
4      if (!map_contains(map, p.mac_src)) {
5          // update mac table
6          map_put(map, p.mac_src, device);
7      }
8
9      if (!map_contains(map, p.mac_dst)) {
10         flood(p);
11     } else {
12         int dst_device = map_get(map, p.mac_dst);
13         forward(p, dst_device);
14     }
15 }
```

The RSS configuration found for this static vigbridge is represented in table 4.3, which Maestro found in 2 seconds. Maestro takes around 10 seconds to synthesize a parallel implementation of vigbridge.

4.1.4 Firewall

Vigor's implementation of a firewall, vigfw, only allows packets from the WAN associated with flows started by packets that came from LAN. Its pseudo-code is illustrated in listing 4.5. Vigfw uses a map to store the flows, and accesses it with flow information, *i.e.*, IP addresses and TCP/UDP ports. However, packets that come from WAN trigger accesses to this map with inverted IP addresses and inverted TCP/UDP ports. This leads to the generation of the constraint matrix shown in fig. 4.4.

Note that the constraint bounding every pair of packets that come from LAN (and, analogously, from WAN) simply denotes that the hash inputs must be equal. This is because, when coming from the same port, they trigger exactly the same accesses to state. The constraint bounding any packet that comes from LAN with any other packets that comes from WAN state that their IP addresses and TCP/UDP ports should be inverted (we omit the UDP equality in the constraint, but it is analogous to the TCP one). With this information, Maestro is able to find the RSS configuration illustrated in table 4.4. This configuration

Listing 4.4: Pseudo-code of static vigbridge.

```
struct Map* map;

void process_packet(int device, pkt_t p) {
    if (!map_contains(map, p.mac_dst)) {
        flood(p);
    } else {
        int dst_device = map_get(map, p.mac_dst);
        forward(p, dst_device);
    }
}
```

Port	Key	RSS packet fields options
LAN	0xb1 0xa9 0x17 0x04 0x5d 0xd9 0x72 0xa7 0x97 0x30 0xdf 0x62 0x8f 0xd9 0xd3 0x83 0xc7 0xc3 0x33 0x0f 0xcf 0x5e 0xf5 0xb9 0x9e 0xb8 0x72 0x19 0x77 0x99 0x71 0x29 0x42 0x88 0x2d 0x9f 0x62 0x9f 0x47 0xf9 0xcf 0x26 0x5b 0x5e 0xff 0x2e 0xe1 0xc6 0xf1 0x14 0xd5 0xc0	• All available
WAN	0xb1 0xa9 0x17 0x04 0x5d 0xd9 0x72 0xa7 0x97 0x30 0xdf 0x62 0x8f 0xd9 0xd3 0x83 0xc7 0xc3 0x33 0x0f 0xcf 0x5e 0xf5 0xb9 0x9e 0xb8 0x72 0x19 0x77 0x99 0x71 0x29 0x42 0x88 0x2d 0x9f 0x62 0x9f 0x47 0xf9 0xcf 0x26 0x5b 0x5e 0xff 0x2e 0xe1 0xc6 0xf1 0x14 0xd5 0xc0	• All available

Table 4.3: RSS configuration of static vigbridge.

was found in approximately 50 minutes, which can be associated with the existence of symmetries in the constraints. Maestro takes around 1 minute and 40 seconds to synthesize a parallel implementation of vigfw.

4.1.5 Load Balancer

The main goal of a Load Balancer is to distribute traffic coming from the WAN to a series of identical servers on the LAN. The distribution algorithms used in load balancers typically aim to spread the computational load throughout the available servers as much as possible.

Viglb, the load balancer implementation in Vigor, registers new servers when it receives their packets coming from the LAN, and matches packets coming from the WAN with the servers it previously registered. Its pseudo-code can be seen in listing 4.6. For the sake of simplicity, we omit the algorithm responsible for assigning WAN packets with LAN servers, and represent it with the function `dchain_find_allocated`, which finds currently allocated indexes in the `dchain`.

When Maestro analyses the viglb's call paths, it finds a use of the function `dchain_find_allocated`. The use of this function prevents Maestro from finding a suitable RSS configuration that leads to a parallel implementation semantically equivalent with the sequential implementation without the use of locking mechanisms.

Listing 4.5: Pseudo-code of vigfw.

```
struct Map* map;

void process_packet(int device, pkt_t p) {
    if (device == WAN) {
        // inverse ports and IPs for the "reply" flow
        struct Flow flow = {
            src_port: p.tcpudp_dst,
            dst_port: p.tcpudp_src,
            src_ip:   p.ipv4_dst,
            dst_ip:   p.ipv4_src
        };

        if (!map_contains(map, flow)) {
            drop(p);
            return;
        }

        int dst_device = map_get(map, flow);
        forward(p, dst_device);
    } else {
        // store flow
        struct Flow flow = {
            src_port: p.tcpudp_src,
            dst_port: p.tcpudp_dst,
            src_ip:   p.ipv4_src,
            dst_ip:   p.ipv4_dst
        };

        map_put(map, flow, device);
        forward(p, WAN);
    }
}
```

$$\begin{bmatrix} (d = d') & \begin{pmatrix} d[\text{IPv4 src}] = d'[\text{IPv4 dst}] \ \wedge \\ d[\text{IPv4 dst}] = d'[\text{IPv4 src}] \ \wedge \\ d[\text{TCP src}] = d'[\text{TCP dst}] \ \wedge \\ d[\text{TCP dst}] = d'[\text{TCP src}] \end{pmatrix} \\ F & (d = d') \end{bmatrix}$$

Figure 4.4: Constraints matrix of vigfw.

New servers (or `backends` in the illustrated code) are registered in the viglb's state by the allocation of a new index in the `dchain`, used to store the servers' information. In a parallel implementation, this would happen in a single core, *i.e.*, servers would be registered in one core only. When a packet comes from the WAN wishing to arrive at any registered server (whichever one it is), it must arrive at a core that has at least one server registered. This is not guaranteed, as it would require a previous register of a server in that specific core. It can happen that a single server is registered, but no packets are forwarded to it in the parallel implementation. This behavior differs from the sequential implementation, where all the servers are registered in a single core.

The specific use of the function `dchain_find_allocated` prevents this automatic lock-free parallel implementation generation because it scans the `dchain` for allocated indexes, accessing multiple memory addresses. There is no RSS configuration that steers packets to the same core that guarantees

Port	Key	RSS packet fields options
LAN	0x20 0x00 0x24 0x22 0x24 0x22 0x20 0x00 0x20 0x00 0x24 0x22 0x20 0x00 0x24 0x23 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	• TCP IPv4 (IPv4 src, IPv4 dst, TCP src, TCP dst) • UDP IPv4 (IPv4 src, IPv4 dst, UDP src, UDP dst)
WAN	0x24 0x22 0x20 0x00 0x20 0x00 0x24 0x22 0x24 0x22 0x20 0x00 0x24 0x22 0x20 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	• TCP IPv4 (IPv4 src, IPv4 dst, TCP src, TCP dst) • UDP IPv4 (IPv4 src, IPv4 dst, UDP src, UDP dst)

Table 4.4: RSS configuration of vigfw.

Listing 4.6: Pseudo-code of viglb.

```
struct Map* backends;
struct Dchain* dchain;

void process_packet(int device, pkt_t p) {
    struct Backend backend;

    if (device == WAN) {
        // get backend
        int allocated_index = dchain_find_allocated(dchain);
        backend = map_get(backends, allocated_index);

        p.ipv4_dst = backend.ip;
        forward(p, backend.nic);
    } else {
        // add a new backend
        int index = dchain_allocate(dchain);
        backend = { ip: p.ipv4_src, nic: device };
        map_put(backends, index, backend);
        drop(p);
    }
}
```

that those packets will access the same state which was accessed for registering the servers. As such, Maestro, when run against viglb, preemptively terminates and returns a message to the user saying that the use of this function prevents the generation of parallel lock-free implementations.

4.1.6 NAT

A Network Address Translator (NAT) translates addresses between network addresses spaces. Their most common use is to allow communication between devices inside a private network with devices on the Internet whilst sharing only a single public IPv4 address. Vignat, a NAT implementation in Vigor, stores flows coming from the LAN and allocates a port that is used to index that flow. When a reply packet is sent to that port, vignat checks if the IP source address and TCP source port matches the IP destination address and TCP destination port of the stored flow, translating and redirecting it to the LAN if this check holds true. The pseudo-code is shown in listing 4.7.

Listing 4.7: Pseudo-code of vignat.

```
1  struct Map* map;
2  struct Dchain* dchain;
3
4  void process_packet(int device, pkt_t p) {
5      if (device == WAN) {
6          if (!dchain_is_index_allocated(dchain, p.tcpudp_dst)) {
7              drop(p);
8              return;
9          }
10
11         struct Flow internal_flow = map_get(map, p.tcpudp_dst);
12
13         if (internal_flow.dst_ip   != p.ipv4_src ||
14             internal_flow.dst_port != p.tcpudp_src) {
15             drop(p);
16             return;
17         }
18
19         // translate
20         p.ipv4_dst   = internal_flow.src_ip;
21         p.tcpudp_dst = internal_flow.src_port;
22
23         forward(p, internal_flow.internal_device);
24     } else {
25         // store flow
26         struct Flow flow = {
27             src_port:        p.tcpudp_src,
28             dst_port:        p.tcpudp_dst,
29             src_ip:          p.ipv4_src,
30             dst_ip:          p.ipv4_dst,
31             internal_device: device
32         };
33
34         // allocate a new port
35         int allocated_port = dchain_allocate(dchain);
36         map_put(map, allocated_port, flow);
37
38         // translate
39         p.ipv4_src   = EXTERNAL_IP;
40         p.tcpudp_src = allocated_port;
41
42         forward(p, WAN);
43     }
44 }
```

Note that this NF interprets the `dchain` allocated indexes as TCP ports, as can be seen in line 6, where the TCP port is checked if it is allocated as a `dchain` index.

As mentioned in section 3.3.2, this interpretation of `dchain` indexes prevents parallel semantic equivalence when configuring RSS with interpreted packet fields. Therefore, we can not apply the same methodology used in the previous NFs for building constraints, as it would use the `map` accesses to generate them. In this case, the constraints generated by using accesses to the `map` use interpreted packet fields, which deem them unusable for the purpose of sharding with those fields.

However, the call paths in this NF that trigger the call to `dchain_is_index_allocated` follow the pattern shown in fig. 3.4. Recall that S_0 corresponds to the code executed before the call to check if the index is allocated in the `dchain` is made. Therefore, in this case, S_0 is empty, as nothing happens in the WAN before the index is checked if it is allocated (in line 6). S_1 corresponds to the code executed before

$$\begin{bmatrix} F & \begin{pmatrix} d[\text{IPv4 dst}] = d'[\text{IPv4 src}] \ \wedge \\ d[\text{TCP dst}] = d'[\text{TCP src}] \end{pmatrix} \\ F & F \end{bmatrix}$$

Figure 4.5: Constraints matrix of vignat.

Port	Key	RSS packet fields options
LAN	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x2f 0xb9 0x81 0x7b 0xfc 0xb0 0x21 0x8a 0x12 0xb5 0x2f 0x75 0x5c 0xd3 0xc8 0x92 0xda 0x7f 0xbf 0x1a 0x63 0x69 0xd8 0x8d 0xa2 0x2c 0x47 0x57 0x18 0x13 0xc6 0x47 0xcd 0x47 0xc2 0xc9	• TCP IPv4 (IPv4 src, IPv4 dst, TCP src, TCP dst) • UDP IPv4 (IPv4 src, IPv4 dst, UDP src, UDP dst)
WAN	0x00 0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	• TCP IPv4 (IPv4 src, IPv4 dst, TCP src, TCP dst) • UDP IPv4 (IPv4 src, IPv4 dst, UDP src, UDP dst)

Table 4.5: RSS configuration of vignat.

a constraint B if the index is found allocated in the `dchain`. Considering B as the condition checked in lines 13 and 14, then S_1 contains the code in line 11. S_2 is the sink, *i.e.*, the code executed if the index was not found allocated in the `dchain`, or if B was not satisfied. Therefore, it contains lines 7-8 and 15-16. Finally S_3, the code executed if B is satisfied, contains lines 20-23.

The fact that the NF's behavior is equal both when the `dchain` indexes checking fails and the B condition fails allows us to use B to configure RSS and achieve a lock-free parallel implementation that is semantically equivalent to the sequential implementation.

Maestro, when analyzing vignat's call paths, realizes that the NF interprets TCP/UDP destination ports as `dchain` indexes, but also finds that it follows the flow shown in fig. 3.4. It then extracts the condition B, and uses it to build the constraints matrix shown in fig. 4.5. This constraint that bounds packets from the LAN and WAN sent to the same core allows for sharding according to the destination device residing in the WAN, *i.e.* shards on the external server's IP address and port. This means that all the flows starting in the LAN but associated with the same address and port in the WAN are sent to the same core, and all the flows coming from the same address and port in the WAN are also sent to that core.

These constraints are then handed to the solver, which finds the RSS configuration shown in table 4.5. This process takes about 50 minutes to complete, as the constraints contain symmetry.

The generated LAN key contains only a single bit with the value one—the 64^{th} bit. This key needs to take into consideration only the IPv4 destination address and TCP/UDP destination port. Therefore, to discard the IPv4 source address, it needs to set all the bits of the key that are xor'ed to it to zero—*i.e.*, the first 63 bits. If there was no need for symmetry between the LAN and WAN, the LAN could potentially

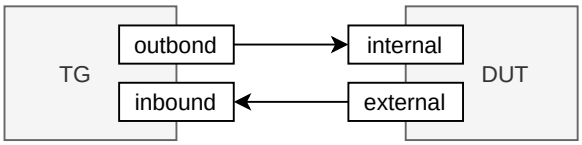


Figure 4.6: Testbed topology used to measure performance, using a traffic generator (TG) and a device under test running an NF (DUT).

have some bits with the value one in the last 32 bits of the total 127 RSS useful bits. However, the hashes generated by packets in the LAN need to equal those generated by the WAN, so long as the former's IPv4 destination address and destination port equals the latter's IPv4 source address and destination address.

Looking now at the WAN, its only bit with the value one is its 32^{nd} bit. There is no room for bits with the value one associated with the source port, as they would also be xor'ed to the IPv4 destination address (which we wish to ignore in the WAN). These constraint between both keys, *i.e.* the symmetry, leads to the existence of only a single key that satisfies the given constraints.

Maestro takes around 2 minutes and 10 seconds to synthesize a parallel implementation of vignat.

4.2 Benchmarking methodology

Testbed. The used testbed is represented in fig. 4.6, as per RFC 2544 [47] and following the same methodology used by Zaostrovnykh *et al.* in Vigor [3]. Both the machine used for traffic generation (TG) and the machine used for testing the NF (device under test, or DUT) are equipped with dual socket Intel Xeon Silver 4114 CPU @ 2.20GHz, 32 GB of DRAM, and Intel X710 10Gb DPDK-compatible NICs. Turbo Boost, Hyper-Threading, and power saving features were disabled. Each of the available NICs in the machines have two ports, connected as shown.

As mentioned in the beginning of this chapter, the two key concepts related to NF's performance are throughput and latency, and measuring them requires the use of different techniques. Both techniques use MoonGen [22] to generate and replay traffic, and measure packet loss and latency.

Throughput. In order to measure throughput, the TG is configured with both a traffic sample and a packet rate value, replaying at the given rate and sending it to its outbound port. Meanwhile, as the NF receives the incoming packets through its internal port and sends them to the external after being processed, the TG receives those later packets in its inbound port. It then compares the amount of packets sent with the ones received, calculating a loss value.

As traffic arrives faster than the NF can handle, packets are dropped. We use MoonGen to perform a binary search to find the maximum packet rate which leads to a loss smaller than 1%, starting with the maximum throughput that the NIC itself is capable of dealing (in this case, 10Gbps). This binary search

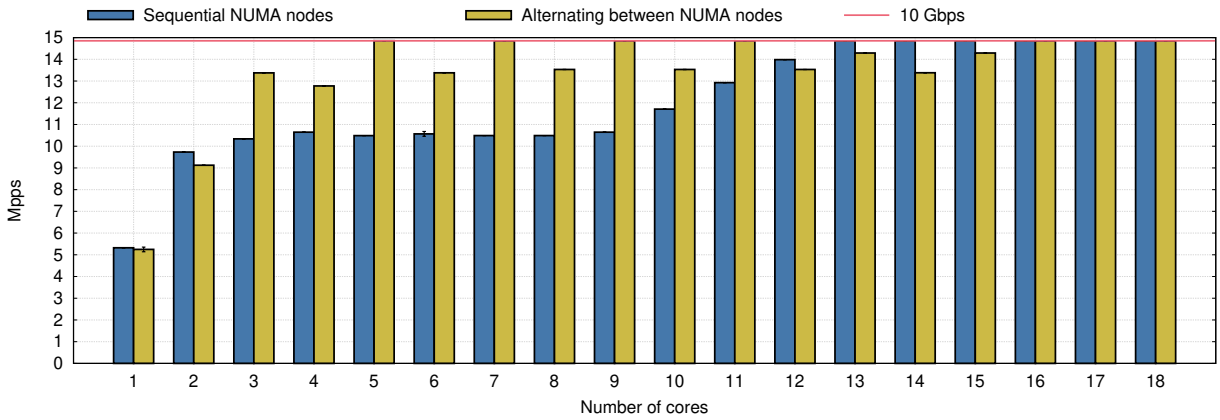


Figure 4.7: Throughput scalability comparison between alternating between NUMA nodes and sequentially using the cores per NUMA node.

uses 10 iterations of throughput measure to acquire the final NF's throughput.

When studying the core scalability of the NF, this method is used for acquiring the throughput value for the NF using an increasing number of cores, running 10 measures for each given NF configuration for statistical significance.

Latency. When dealing with throughput of tens of millions of packets per second, achieving measures of packet round trip time (RTT) with fine granularity becomes vital to obtain latency values. MoonGen, by using hardware timestamping following the Precision Time Protocol [48], is capable of retrieving latency values with a granularity of $10ns$ [49]. As such, the NF is subjected to timestamped probe packets, which are later received by the TG and used to calculate the latency.

4.3 NUMA considerations

Emmerich *et al.* [1], showed, in a NUMA architecture, how each possible combination of NICs, memory, and NF threads pinning to NUMA nodes can influence throughput. Our machines are equipped with a single NIC, and therefore both ingress and egress ports can only be pinned to a single CPU. Their findings show that, under these circumstances, pinning the packet buffers to the CPU associated with the NIC is optimal. However, a performance drop is expected when running the NF on cores of the CPU not connected to the NIC.

A question then arises—which combination of cores from each NUMA node should one use in order to achieve the best performance on a parallel NF? In order to answer this question, we used the parallel implementation of vignop as a test subject.

First, we measured its throughput scalability by running it with an increasing number of cores from the CPU pinned to the NUMA node connected to the NIC. When all the available cores from this node were used, we then continued to add cores from the other node, until those were exhausted.

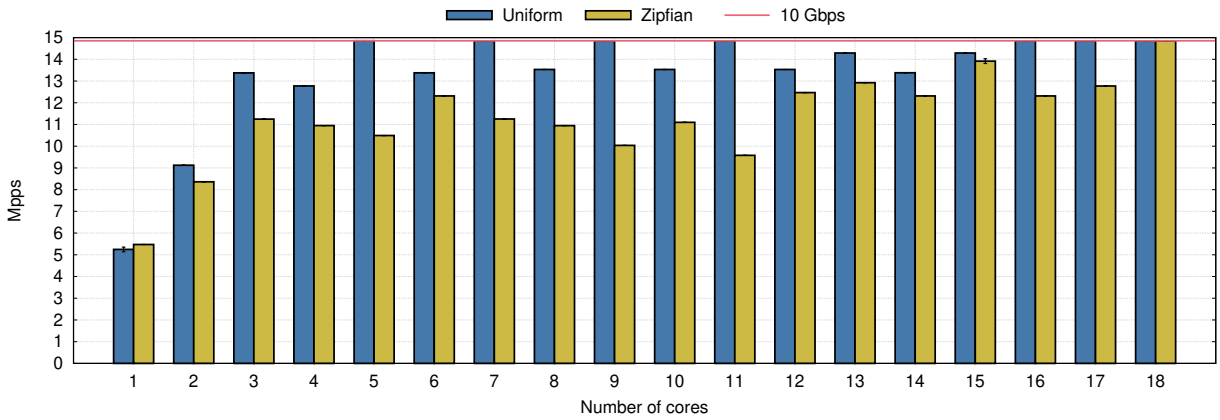


Figure 4.8: Variation of vignop's throughput under a uniform distribution and a Zipfian distribution.

The second scenario, instead of first choosing cores from one node before using cores from the other available node, alternates between the two nodes (i.e., running the NF with one core from node 0, with two cores from nodes 0 and 1, with three cores for which two are from the first and one is from the second, etc.).

The results are shown in fig. 4.7. We achieve better scalability with fewer cores in the sequential method, but quickly encounter a bottleneck preventing throughput increase with the number of cores. Only when all cores from the first node are used and cores from the second node are added the throughput increases, relating to a memory controller bottleneck [1]. When alternating between nodes, we achieve the 10 Gbps limit imposed by the NIC when using five cores (three from the node connected to the NIC, two from the other).

The bottleneck achieved by the sequential implementation, as seen when using three to nine cores (all from the same NUMA node), can be associated with the memory controller. Dobrescu *et al.* [50], in their study of contention in parallel packet processing applications, showed that the dominant contention factor is the cache. More specifically, contention for Intel Data Direct I/O (DDIO) resources, shown by Manousis *et al.* [51] to be one of the three sources of contention across the memory subsystem. Using DDIO, the packets coming from the NIC are directly placed in the last level cache (LLC) of the NUMA node. Contention, as exposed by the authors, happens when the number of concurrent packets exceeds the available reserved space for I/O in the LLC. We can see an increase in throughput when using 10 cores in fig. 4.7, as the added core is associated with a different NUMA node, and therefore a new LLC.

Future measures of throughput presented in this dissertation take these results into consideration, measuring scalability by alternating between NUMA nodes.

4.4 Traffic skew

The throughput measures presented in the previous section relate to an NF subjected to a traffic with a uniform distribution of flows. Although it suited the scenario under study, it did not correspond to

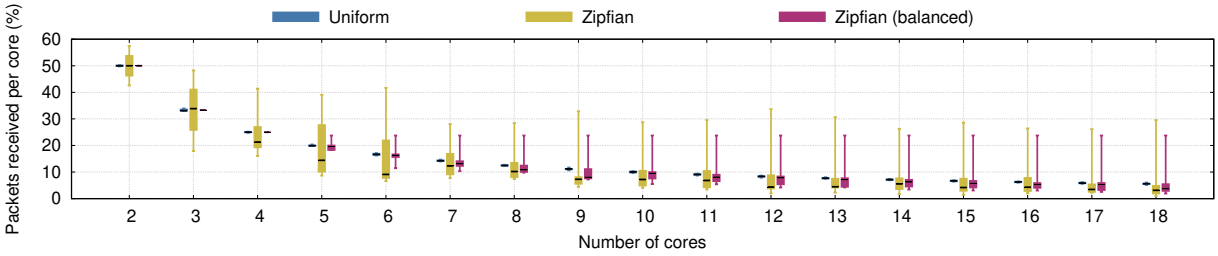


Figure 4.9: Skew per number of cores with vignop. For every number of cores, we show three boxplots representing the distribution of packets per core when the NF is submitted to a uniform distribution (left-most boxplot), Zipfian distribution (center boxplot), and Zipfian distribution with a balanced indirection table (right-most boxplot). A narrower boxplot indicates that each core received a similar amount of packets, and a wider boxplot indicates that some cores received more packets than the others.

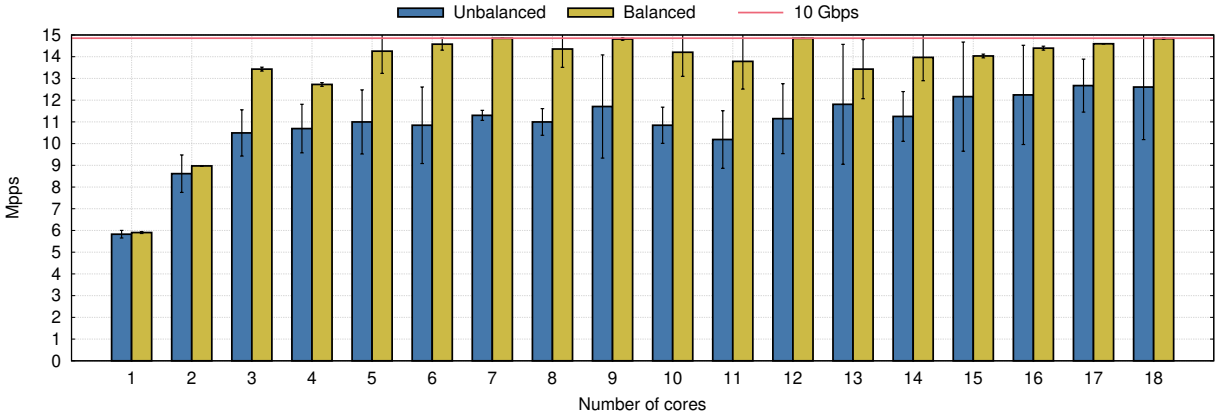


Figure 4.10: Throughput variation for vignop with Zipfian traffic using unbalanced and balanced indirection tables.

a distribution often found in the real-world. The expression ”mice and elephants” is typically used to describe the distribution of flows on the internet [32, 33]. This corresponds not to a uniform distribution of flows, where there is an equal probability of two packets arriving with the same flow IDs, but to a Zipfian distribution, where a large percentage of arriving packets share the same destination address.

As a traffic with uniform distribution leads to packets being uniformly distributed to cores, traffic following a Zipfian distribution can overload a subset of cores. This performance difference is shown in fig. 4.8, which demonstrates how the parallel vignop's throughput varies according to the traffic distribution to which it is subjected. The Zipfian traffic was generated using the Zipfian parameters found by Pedrosa *et al.* [35], which were found by analyzing real-world traffic samples from a University network [34].

4.4.1 Balancing the indirection table

Both the RSS key and the indirection table can influence throughput. More collisions produced by the RSS hash lead to more packets sent to the same core. Maestro exploits these collisions to shard packets according to certain packet fields in order to maintain semantics, but unintended collisions can still happen. Even if two packets generate different hashes they can still be sent to the same core, so

long as the values stored in the indirection table contain the same values.

As previously shown, traffic can influence the performance of an NF. In a uniformly distributed traffic, all the entries of the indirection table are expected to be accessed with an equal probability, leading to a uniform distribution of packets throughout the cores if the indirection table contains an equal number of all the possible values. However, in a Zipfian distribution, which corresponds to a more realistic model of packet flow distribution, the higher density of certain flows leads to a higher number of accesses to certain indirection table entries, overloading a subset of the available cores.

RSS++ [36] fixes this distribution problem by dynamically adjusting the indirection table according to the traffic. It balances the indirection table by swapping entries associated with overloaded cores for ones associated with underloaded ones.

Maestro uses the RSS++ balancing algorithm to generate parallel NFs which balance the indirection table according to the subjected traffic. Figure 4.9 shows three scenarios of different variations of packets received by each core, using a vignop—under a uniform distribution, under a Zipfian distribution, and finally under a Zipfian distribution but with a balanced indirection table. The uniform distribution, as expected, leads to minor variations of packets received by each core. The presence of elephant flows in the Zipfian distribution leads to core overloading, as some cores receive more traffic than others. Using a balanced indirection table achieved by the previously mentioned algorithm does not always, as can be seen, lead to a uniform distribution of packets throughout the cores. Taking the load of an overloaded core requires assigning its indirection table entries to an underloaded core. However, rebalancing has the granularity of individual flows. If a single elephant overloads a core, then it can not be rebalanced optimally.

Even if a balanced indirection table is unable to achieve a distribution as good as the one achieved by subjecting the NF to a uniform distribution, we can still detect throughput improvements. The throughput variations using both an unbalanced and a balanced indirection table, under the Zipfian traffic, can be seen in fig. 4.10. This data was extracted by configuring the vignop with three different RSS keys, and measuring its throughput with both an unbalanced indirection table, and a balanced one. The error bars represent the throughput variations between the configurations with different RSS keys.

4.5 Performance benchmarking

After Maestro, whenever possible, generated parallel lock-free implementations of each available Vigor NF, we studied their performance when configured to run with different numbers of cores. Each execution with a different number of cores was subjected to the NUMA considerations mentioned previously by alternating the added cores from each available NUMA node. Moreover, they were subjected to Zipfian traffic, and the NIC's indirection table was previously balanced under both the RSS key used and the Zipfian traffic sample.

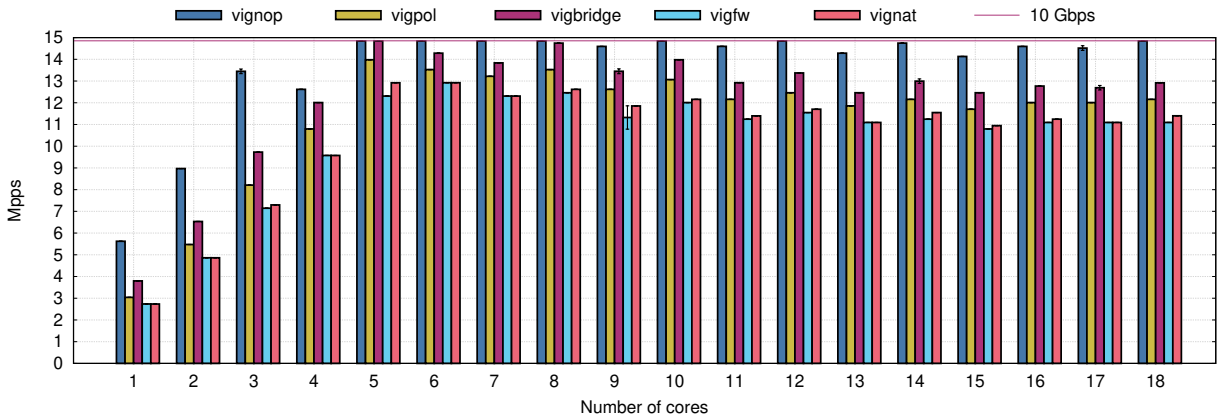


Figure 4.11: Throughput with a shared-nothing model.

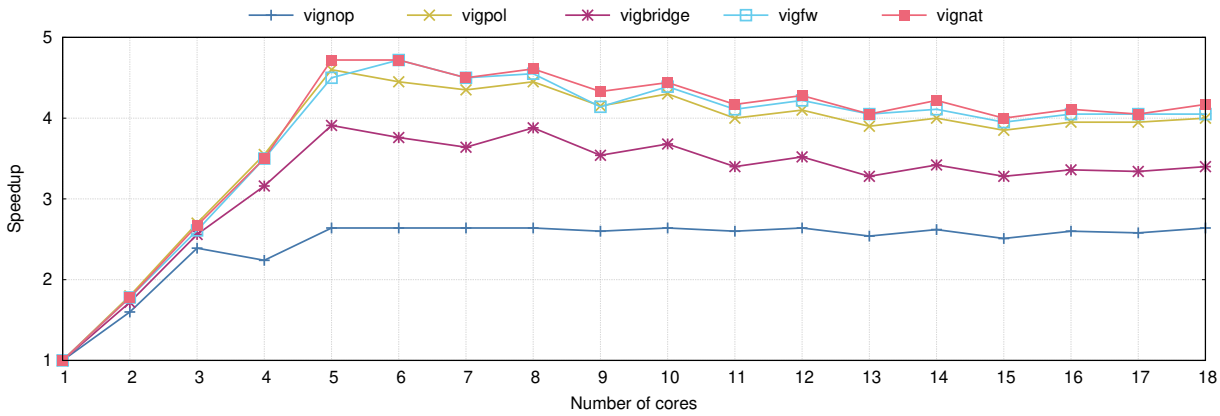


Figure 4.12: Throughput speedup per NF.

The retrieved throughput scalability is shown in fig. 4.11. The maximum throughput is achieved when using five cores for vigpol and vigbridge, and when using six cores for vigfw and vignat. Note that the values shown with single core performance differ from the ones shown in the Vigor paper [3]. This can be attributed to differences in the CPU, as their evaluation was done using a CPU with a higher clock frequency. Moreover, the vigbridge referenced by the authors is dynamic, whilst ours is static.

The speedup values for each NF are shown in fig. 4.12. Overall, no NF was capable of achieving $5\times$ the throughput of the single core execution, even with 18 available cores. We can associate this limitation to bottlenecks imposed by the memory controller, the PCIe bus, and core distribution (as balancing the indirection table does not lead to the same performance as one achieved by using uniform traffic). Vignop and vigbridge were the only NFs limited by the 10Gbps maximum rate imposed by the NIC, as expected from NFs that performs no writes to their state, and are configured with random RSS keys and all the available RSS-compatible packet fields.

The latency associated with each parallel NF was also measured, and the results can be seen in fig. 4.13. The dots in the plots correspond to the median latency, and the bottom and top errors bars correspond to the 5^{th} and 95^{th} percentiles, respectively. Overall, we detected no noticeable differences

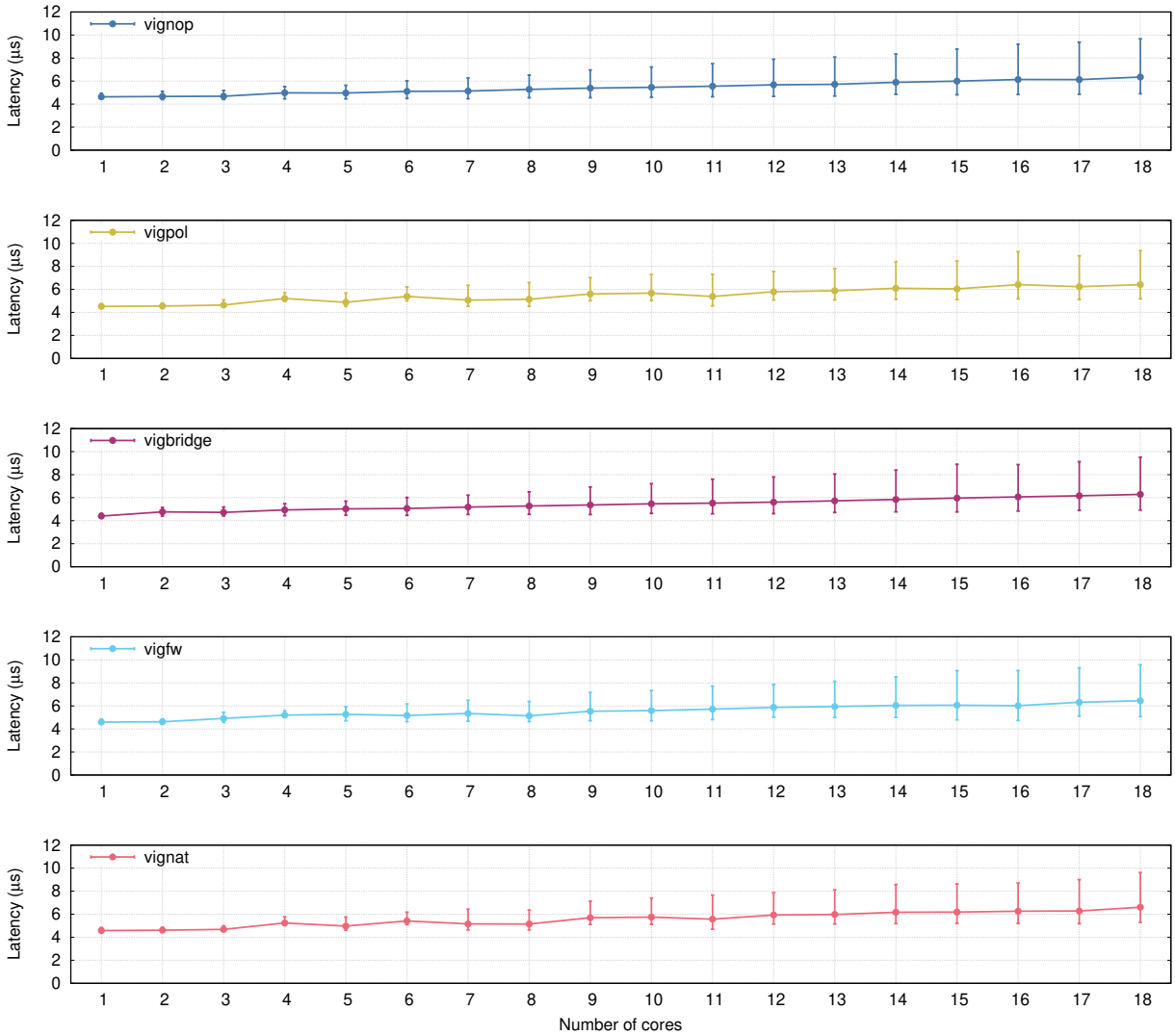


Figure 4.13: Latency of each NF with the number of used cores.

in latency between the NFs. When running with a single core, every NF was able to provide around 4 to 5 μs of latency. As they used more cores, the latency slowly increased, reaching 6 μs with 18 cores, with the 95^{th} percentile reaching 10 μs. The slight increase in latency with the number of cores can be associated with an increasing use of cores in the NUMA node not pinned to the NIC, as the packets need to travel to other NUMA node to be processed.

4.5.1 Benchmarking parallel lock-based NFs

In order to compare the performance achieved by a parallel NF with a shared-nothing architecture with a lock-based one, we also measured the performance achieved by each NF under the latter implementation. The measured throughput of each NF is shown in fig. 4.14. The latency associated with the

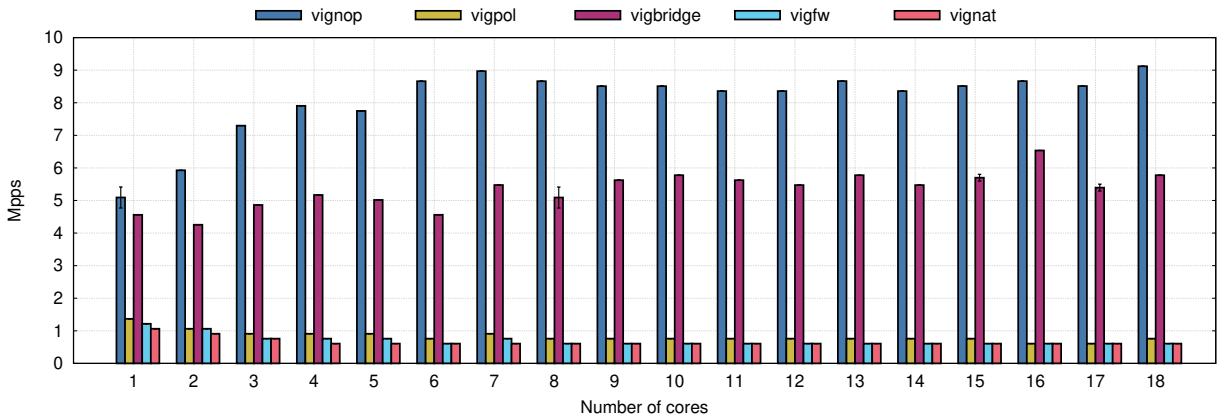


Figure 4.14: Throughput with locking mechanisms.

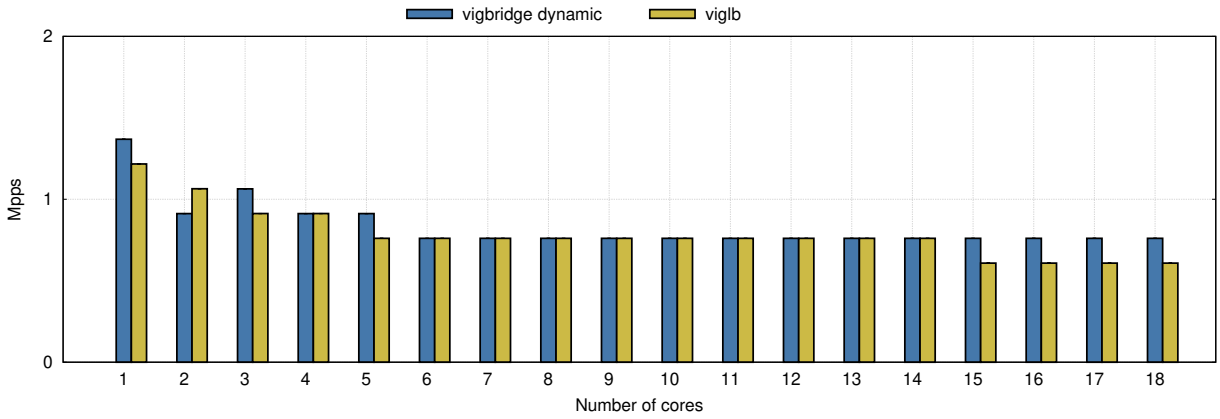


Figure 4.15: Throughput with locking mechanisms of NFs that can not be parallelized with a shared-nothing model.

NFs using locking mechanisms show no change when compared (if applicable) to their shared-nothing modeled counterparts. Nonetheless, the latency results can be found in appendix A (fig. A.1).

As expected, vignop and vigbridge achieve the highest values of throughput, as they constitute read-only NFs. As for the others, we can observe a decrease in throughput from an increasing number of used cores. This is justified by the fact that all packets in these NFs constitute write-packets, as they need to update timestamps to avoid being expired. As each write-packet triggers a sequential packet processing scenario, this leads to a worse than sequential performance, as different cores need to communicate with each other to orderly process the packets.

In order to achieve a shared-nothing model, vigbridge had to be modified to contain a read-only table of MAC addresses. Using locking mechanisms, it becomes possible to parallelize the original vigbridge, with its dynamic table. Similarly, Vigor's implementation of a load balancer—viglb—can now also be parallelized. The throughput achieved by each of these NFs is presented in fig. 4.15. The latency measures, as the ones associated with the previous NFs, can be found in appendix A (fig. A.1).

5

Conclusion

Contents

5.1 Conclusions

This dissertation contributes to the state of the art of NFV by introducing Maestro, a tool that automatically generates parallel shared-nothing or lock-based implementations of NFs from their respective sequential implementations.

Using the Vigor framework and its verification byproducts, Maestro is able to analyze how an NF maintains its state. It uses this information to shard its data structures, producing a parallel implementation that allows all its concurrent instances to freely process incoming packets without communicating with each other. This is made possible my finding an RSS configuration that redirects packets satisfying a set of conditions to the same core. That respective core keeps all the necessary information needed to process the packets redirected to it, and no information related to those packets are stored in other cores.

Additionally, we present a C library capable of finding RSS configurations that allow for this careful packet redirection, named libR3S. An NF developer can use libR3S to discover RSS configurations that redirect packets following the given conditions to the same core. This enables the developer to build their own parallel implementations, similar to the ones achieved by Maestro, without the use of our Vigor dependent tool.

When Maestro is unable to find a shared-nothing parallel solution that maintains semantics with the sequential implementation, it can be allowed to generate a lock-based parallel implementation. However, there is a significant performance penalty involved.

We measured the performance achieved by six different NFs, using their respective Maestro generated parallel implementations. We recorded throughput speedups from $2.5\times$ (the NOP NF, a stateless NF that redirects all the received traffic without making any operations), to $4.7\times$ (a NAT). The best throughput speedup values were achieved using five cores from two nodes in a NUMA architecture. We also recorded that latency was mostly unaffected in the parallel NFs, showing only slight increases with the number of cores used due to the added overhead of transiting across NUMA nodes.

5.2 System Limitations and Future Work

5.2.1 Synthesizer

At the current time, the Synthesis engine does not yet fully automate the push-to-parallelize process. However, fully automating the process is (immediate) future work.

5.2.2 Formal verification of the generated implementations

Although Maestro uses formally verified sequential implementations of NFs built in Vigor, its generated parallel implementations are not formally verified. Verification is not simple, and is outside the scope of this thesis. As such, it is left as future work.

5.2.3 RSS limitations

The Maestro's ability to generate its parallel implementations is limited by the available RSS configurations. As shown in the previous chapter, some NFs need to be sharded by RSS-incompatible packet fields. Adding more packet fields to the RSS process would allow Maestro to generate shared-nothing parallel implementations for more NFs. However, this is intrinsically associated with the NIC's capabilities, leaving little room for improvement in the academic landscape with the current hardware. This limitation could potentially be uplifted by either implementing RSS in software, or redirecting packets to cores with other mechanisms, which we leave to future work.

5.2.4 Following Maestro's example

Maestro is the proof that the functional representation of an NF extracted by Vigor (call paths) can be used to synthesize modified versions of its NF. In this thesis, we generate parallel implementations that run with DPDK under x86 machines, but it is far from the only use associated with this capability. This automated synthesis can potentially be used for, for example, adding new functionality, porting to new devices, and exploring other potential performance optimizations.

Bibliography

[1] P. Emmerich, M. Pudelko, S. Bauer, and G. Carle, "User space network drivers," *ANRW 2018 - Proceedings of the 2018 Applied Networking Research Workshop*, pp. 91–93, 2018.

[2] E. Papadogiannaki, L. Koromilas, G. Vasiliadis, and S. Ioannidis, "Efficient software packet processing on heterogeneous and asymmetric hardware architectures," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1593–1606, 2017.

[3] A. Zaostrovnykh, S. Pirelli, R. Iyer, M. Rizzo, L. Pedrosa, K. Argyraki, and G. Candea, "Verifying software network functions with no verification expertise," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, 2019, pp. 275–290.

[4] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 203–216. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[5] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 13–24, Aug. 2012. [Online]. Available: http://doi.acm.org/10.1145/2377677.2377680

[6] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng *et al.*, "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action," in *SDN and OpenFlow World Congress*, oct 2012. [Online]. Available: https://portal.etsi.org/NFV/NFV_White_Paper.pdf

[7] European Telecommunications Standards Institute (ETSI), "Network Functions Virtualisation (NFV)," retrieved 2019-12-29. [Online]. Available: https://www.etsi.org/technologies/nfv

[8] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys and Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[9] ONF, “Software-defined networking: The new norm for networks,” Open Networking Foundation, Tech. Rep., April 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[10] Network Functions Virtualisation (NFV) ETSI Industry Specification Group (ISG), “ETSI GS NFV 002 - V1.2.1 - Network Functions Virtualisation (NFV); Architectural Framework,” 2014, retrieved 2019-12-03. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf

[11] Network Functions Virtualisation (NFV) ETSI Industry Specification Group (ISG), “ETSI GS NFV 001 - V1.2.1 - Network Functions Virtualisation (NFV); Use Cases,” 2017, retrieved 2019-12-03. [Online]. Available: https://www.etsi.org/deliver/etsi_gr/NFV/001_099/001/01.02.01_60/gr_nfv001v010201p.pdf

[12] L. Rizzo, “NetMap: A novel framework for fast packet I/ O,” *Atc ’12*, no. 257422, pp. 101—-112, 2012. [Online]. Available: https://www.usenix.org/conference/usenixfederatedconferencesweek/netmap-novel-framework-fast-packet-io

[13] T. Barbette, C. Soldani, and L. Mathy, “Fast userspace packet processing,” *ANCS 2015 - 11th 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 5–16, 2015.

[14] D. Scholz, “A look at intel’s dataplane development kit,” in *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Summer Semester 2014*, ser. Network Architectures and Services (NET), G. Carle, D. Raumer, and L. Schwaighofer, Eds., vol. NET-2014-08-1. Munich, Germany: Chair for Network Architectures and Services, Department of Computer Science, Technische Universität München, Sep. 2014, pp. 115–122. [Online]. Available: http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2014-08-1/NET-2014-08-1_15.pdf

[15] J. García-Dorado, F. Mata, J. Ramos, P. Santiago del Río, V. Moreno, and J. Aracil, *High-Performance Network Traffic Processing Systems Using Commodity Hardware*, 01 2013, pp. 3–27.

[16] Intel, “Data Plane Development Kit,” 2010, retrieved 2019-12-15. [Online]. Available: https://www.dpdk.org

[17] N. Bonelli, S. Giordano, and G. Procissi, “Network Traffic Processing with PFQ,” *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 6, pp. 1819–1833, 2016.

[18] S. McCanne and V. Jacobson, “The bsd packet filter: A new architecture for user-level packet capture.” in *USENIX winter*, vol. 46, 1993.

[19] Fleming, Matt, “A thorough introduction to eBPF,” retrieved 2020-12-28. [Online]. Available: https://lwn.net/Articles/740157/

[20] L. Deri, “Improving Passive Packet Capture : Beyond Device Polling,” *Proceedings of SANE*, vol. 2004, p. 85, 2004.

[21] M. Trevisan, A. Finamore, M. Mellia, M. Munafo, and D. Rossi, “Traffic analysis with off-the-shelf hardware: Challenges and lessons learned,” *IEEE Communications Magazine*, vol. 55, no. 3, pp. 163–169, 2017.

[22] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, “MoonGen: A scriptable high-speed packet generator,” *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, vol. 2015-Octob, pp. 275–287, 2015.

[23] X. Wu, P. Li, Y. Ran, and Y. Luo, “Network measurement for 100 GbE network links using multicore processors,” *Future Generation Computer Systems*, vol. 79, pp. 180–189, 2018.

[24] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, “A formally verified nat,” *SIGCOMM 2017 - Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, pp. 141–154, 2017.

[25] H. Sutter, “The free lunch is over: A fundamental turn toward concurrency in software,” *Dr. Dobb’s journal*, vol. 30, no. 3, pp. 202–210, 2005.

[26] L. Dagum and R. Menon, “OpenMP: An Industry-Standard API for Shared-Memory Programming,” Tech. Rep., 1998. [Online]. Available: www.openmp.org.

[27] M. Vesović, A. Smiljanić, and M. Tomašević, “Speeding up ip lookup procedure in software routers by means of parallelization,” *Telfor Journal*, vol. 9, no. 1, pp. 2–7, 2017.

[28] F. Fusco and D. Luca, “High speed network traffic analysis with commodity multi-core systems,” *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, pp. 218–224, 2010.

[29] Microsoft Inc., “Introduction to Receive Side Scaling,” retrieved 2019-12-29. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling

[30] H. Krawczyk, “LFSR-based hashing and authentication,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 839 LNCS, pp. 129–139, 1994.

[31] S. Woo and K. Park, “Scalable TCP session monitoring with symmetric receive-side scaling,” pp. 1–7, 2012. [Online]. Available: http://www.ndsl.kaist.ac.kr/{~}shinae/papers/TR-symRSS.pdf

[32] L. Guo and I. Matta, “The war between mice and elephants,” in *Proceedings Ninth International Conference on Network Protocols. ICNP 2001*. IEEE, 2001, pp. 180–188.

[33] K.-c. Lan and J. Heidemann, “A measurement study of correlations of internet flow characteristics,” *Computer Networks*, vol. 50, no. 1, pp. 46–62, 2006.

[34] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010, pp. 267–280.

[35] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki, “Automated synthesis of adversarial workloads for network functions,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 372–385.

[36] T. Barbette, G. P. Katsikas, G. Q. Maguire Jr, and D. Kostić, “Rss++ load and state-aware receive side scaling,” in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019, pp. 318–333.

[37] Intel, “DPDK: Poll Mode Driver,” retrieved 2019-12-24. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html

[38] M. Jamshed, J. Lee, S. Moon, D. Kim, S. Lee, and K. Park, “Kargus : A Highly-scalable Software-based Intrusion Detection System Categories and Subject Descriptors,” *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 317–328, 2012.

[39] J. Nam, M. Jamshed, B. Choi, D. Han, and K. S. Park, “Haetae: Scaling the performance of network intrusion detection with many-core processors,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9404, pp. 89–110, 2015.

[40] M. Dobrescu, N. Egi, K. Argyraki, B. G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, “RouteBricks: Exploiting parallelism to scale software routers,” in *SOSP’09 - Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, 2009, pp. 15–28.

[41] N. Egi, A. Greenhalgh, M. Handley, G. Iannaccone, M. Manesh, L. Mathy, and S. Ratnasamy, “Improved forwarding architecture and resource management for multi-core software routers,” *NPC 2009 - 6th International Conference on Network and Parallel Computing*, pp. 117–124, 2009.

[42] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 209–224, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855756

[43] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[44] "Libr3s source code," https://github.com/fchamicapereira/libr3s, 2020.

[45] B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki, "Local search algorithms for partial maxsat," *AAAI/IAAI*, vol. 263268, 1997.

[46] Z. Fu and S. Malik, "On solving the partial max-sat problem," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2006, pp. 252–265.

[47] S. Bradner and J. McQuaid, "Benchmarking methodology for network interconnect devices," Internet Requests for Comments, RFC Editor, RFC 2544, 03 1999. [Online]. Available: https://tools.ietf.org/rfc/rfc2544.txt

[48] K. B. Lee and J. Eldson, "Standard for a precision clock synchronization protocol for networked measurement and control systems," in *2004 Conference on IEEE 1588, Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, 2004.

[49] M. Primorac, E. Bugnion, and K. Argyraki, "How to measure the killer microsecond," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 5, pp. 61–66, 2017.

[50] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," in *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 141–154.

[51] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 270–282.

A

Appendix

Listing A.1: Complex LVA example.

```
========================================
Access 210
  file         test000021.call_path
  src device   0
  dst device   1
  interface    map_put
  operation    WRITE
  object       1073838032
  read         (Concat w120 (Read w8 (w32 50) packet_chunks)
                 (Concat w112 (w8 0)
                   (Concat w104 (w8 0)
                     (Concat w96 (Read w8 (w32 60) packet_chunks)
                       (Concat w88 (Read w8 (w32 59) packet_chunks)
                         (Concat w80 (Read w8 (w32 58) packet_chunks)
                           (Concat w72 (Read w8 (w32 57) packet_chunks)
                             (Concat w64 (Read w8 (w32 56) packet_chunks)
                               (Concat w56 (Read w8 (w32 55) packet_chunks)
                                 (Concat w48 (Read w8 (w32 54) packet_chunks)
                                   (Concat w40 (Read w8 (w32 53) packet_chunks)
               ))))))))))
  layer        3
  protocol     2048
  fragments    offset 41 expression (ReadLSB w160 (w32 41) packet_chunks)
               offset 82 expression (Read w8 (w32 82) packet_chunks)
  dependencies 9
               19
               18
               17
               16
               15
               14
               13
               12

  layer        4
  protocol     17
  fragments    offset 123 expression (ReadLSB w32 (w32 123) packet_chunks)
  dependencies 3
               2
               1
               0

  layer        4
  protocol     6
  fragments    offset 123 expression (ReadLSB w32 (w32 123) packet_chunks)
  dependencies 3
               2
               1
               0

  write        (ReadLSB w32 (w32 0) new_index_1)
========================================
```

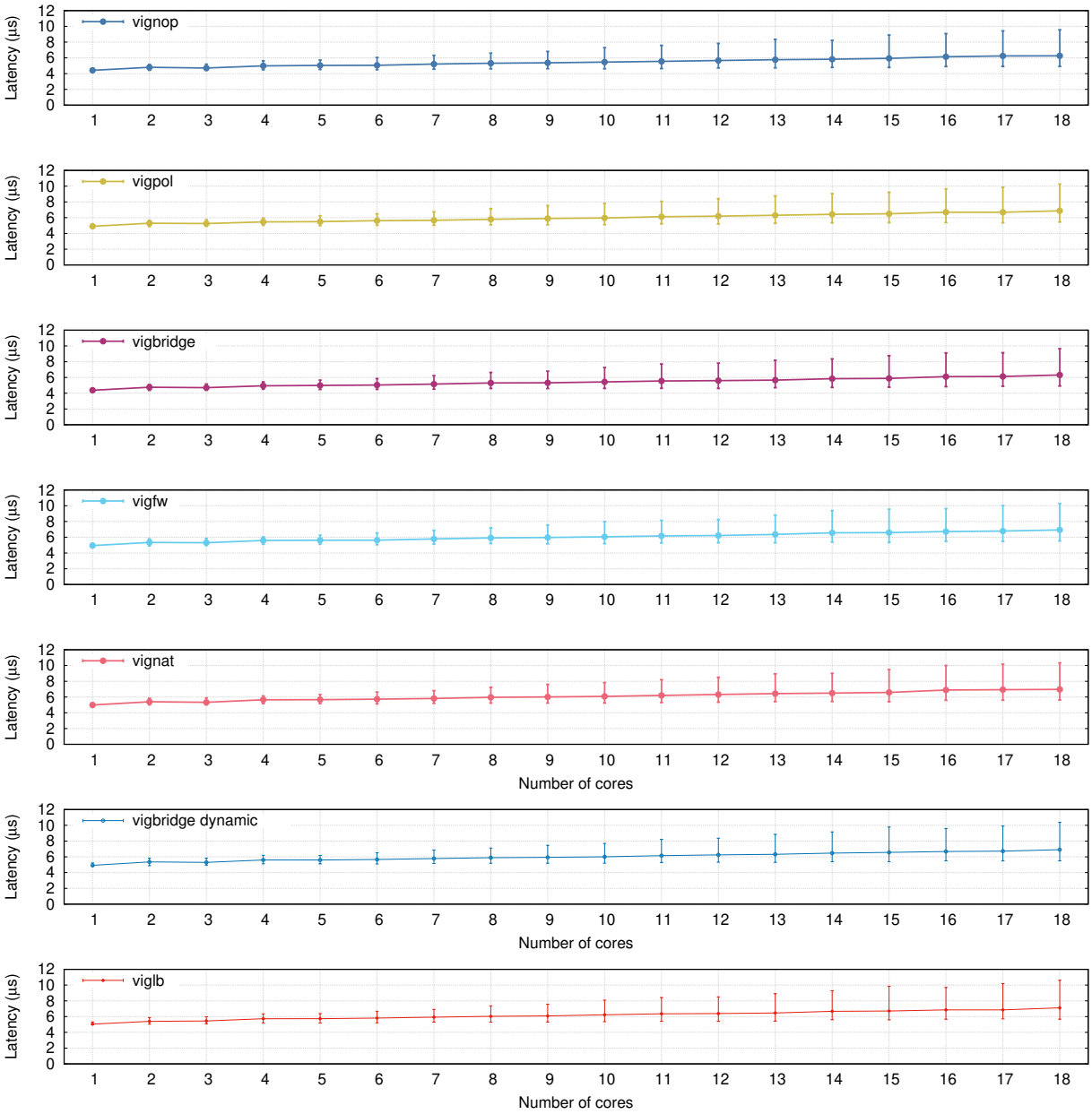


Figure A.1: Latency of generated parallel NFs using locking mechanisms.

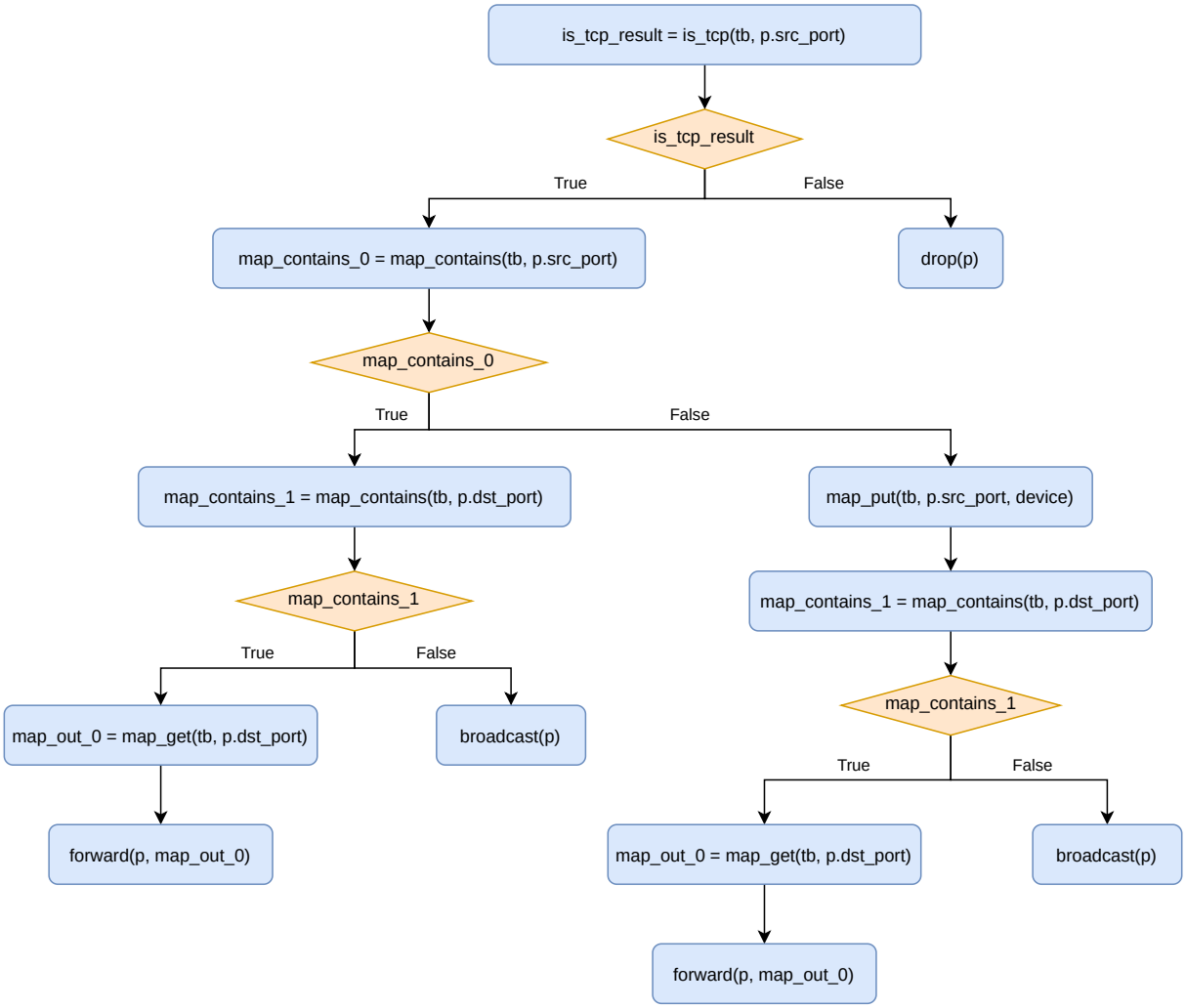


Figure A.2: Generated binary decision tree for the call paths in listing 2.3.

Listing A.2: Use of libR3S to find an RSS key that sends packets of the same session to the same core.

```
#include <r3s.h>

Z3_ast R3S_cnstr_symmetric_tcp_ip(R3S_cfg_t cfg,
                                  R3S_packet_ast_t p1,
                                  R3S_packet_ast_t p2)
{
    Z3_ast p1_ipv4_src, p1_ipv4_dst, p1_tcp_src, p1_tcp_dst;
    Z3_ast p2_ipv4_src, p2_ipv4_dst, p2_tcp_src, p2_tcp_dst;
    Z3_ast and_args[4];

    // extracting the packet fields into the Z3_ast objects

    R3S_packet_extract_pf(cfg, p1, R3S_PF_IPV4_SRC, &p1_ipv4_src);
    R3S_packet_extract_pf(cfg, p1, R3S_PF_IPV4_DST, &p1_ipv4_dst);
    R3S_packet_extract_pf(cfg, p1, R3S_PF_TCP_SRC,  &p1_tcp_src);
    R3S_packet_extract_pf(cfg, p1, R3S_PF_TCP_DST,  &p1_tcp_dst);

    R3S_packet_extract_pf(cfg, p2, R3S_PF_IPV4_SRC, &p2_ipv4_src);
    R3S_packet_extract_pf(cfg, p2, R3S_PF_IPV4_DST, &p2_ipv4_dst);
    R3S_packet_extract_pf(cfg, p2, R3S_PF_TCP_SRC,  &p2_tcp_src);
    R3S_packet_extract_pf(cfg, p2, R3S_PF_TCP_DST,  &p2_tcp_dst);

    // making the necessary equalities

    and_args[0] = Z3_mk_eq(cfg->ctx, p1_ipv4_src, p2_ipv4_dst);
    and_args[1] = Z3_mk_eq(cfg->ctx, p1_ipv4_dst, p2_ipv4_src);
    and_args[2] = Z3_mk_eq(cfg->ctx, p1_tcp_src, p2_tcp_dst);
    and_args[3] = Z3_mk_eq(cfg->ctx, p1_tcp_dst, p2_tcp_src);

    // building the final constraint

    return Z3_mk_and(cfg->ctx, 4, and_args);
}

int main () {
    R3S_status_t status;
    R3S_cfg_t    cfg;
    R3S_key_t    k;

    R3S_cfg_init(&cfg);
    R3S_cfg_set_number_of_keys(cfg, 1);

    R3S_cfg_load_opt(cfg, R3S_OPT_NON_FRAG_IPV4_TCP);

    status = R3S_keys_fit_cnstrs(cfg, &R3S_cnstr_symmetric_tcp_ip, &k);

    if (status == R3S_STATUS_SUCCESS)
        printf("%s\n", R3S_key_to_string(k));

    R3S_cfg_delete(cfg);
}
```