UN*i*S: A User-space Non-intrusive Workflow-aware Virtual Network Function Scheduler

by

Anthony Anthony

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

Author's Declaration

Statement of Contributions

Most parts of this thesis appeared in this publication:
A. Anthony, S. R. Chowdhury, T. Bai, R. Boutaba, and J. Francois, "UNiS: A User-space Non-intrusive Workflow-aware Virtual Network Function Scheduler", in the 14th Conference on Network and Service Management (CNSM), 5-9 November 2018.

Abstract

Network Function Virtualization (NFV) has gained a significant research interest in both academia and industry since its inception in the late 2012. One of the key research issues in NFV is the development of systems for building *Virtual Network Functions (VNFs)* capable of meeting the performance requirements of enterprise and telecommunication networks. New packet processing models leveraging kernel bypass I/O and poll-mode processing have gained popularity for building high performance VNFs because of their simple programming model and very low I/O overhead. However, a major drawback of such poll-mode processing is the inefficient use of CPU resources. Existing CPU schedulers are ill-suited for VNFs due to their inability to capture the actual processing cost of a poll-mode VNF, hence, cannot rightsize the CPU allocation. This is further exacerbated by their inability to consider VNF processing order when VNFs are chained to form Service Function Chains (SFCs).

The state-of-the-art solutions proposed for VNF scheduling are *intrusive*, *i.e.*, requiring the VNFs to be built with scheduler specific libraries or having carefully selected scheduling checkpoints. This highly restricts the VNFs that can properly work with such schedulers. To address these issues, we developed UN*i*S: a User-space Non-intrusive work-flow aware VNF Scheduler. Unlike existing approaches, UN*i*S does not require VNF modifications and treats the poll-mode VNFs as a black box, hence, is *non-intrusive*. UN*i*S is also *workflow-aware*, *i.e.*, maintains SFC processing order while scheduling the VNFs. Testbed experiments show that UN*i*S is able to achieve a throughput within 90% (for synthetic traffic load) and 98% (for real data center traffic trace) of the achievable throughput using an intrusive co-operative scheduler.

Acknowledgements

I take this opportunity to specially thank my supervisor Professor Raouf Boutaba for his invaluable guidance in this project, his continuous support, and the tremendous opportunities given to me throughout my Master's studies. It was a true privilege for me to work with him.

I am very grateful to my family, especially my mother, for the unconditional love and support during my stay here.

My special thanks to Shihabur Rahman Chowdhury for his constant support along the course of this project both technical and moral, Tim Bai for his involvement in the early stage of this project, and Haibo Bian for his contribution in the NFV platform. Thanks to everyone in the Network Lab and everyone in DC-3552 for the friendship. It was a truly enjoyable and enriching experience I had through the ups and downs in the last two years.

I would also like to thank my thesis committee members, Khuzaima Daudjee and Ali Mashtizadeh, for their time and constructive feedback. Also, I would like to express my gratitude to Jérôme François for his valuable advice in this project during my visit at INRIA.

Dedication

This is dedicated to my family.

Table of Contents

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Motivation 2

1.2 Design Goals and Challenges 4

1.3 Contributions 5

1.4 Thesis Organization 5

2 Background **7**

2.1 Network Function Virtualization 7

2.2 Kernel Bypass Network I/O Technologies 8

2.3 Packet Processing with DPDK 9

2.4 Process Scheduling in Linux 9

2.5 Related Work on NFV Scheduling 10

2.5.1 Analytical Models for NFV Scheduling 11

2.5.2 Systems for NFV Scheduling 11

3 UN*i*S: A User-space Non-intrusive Workflow-aware Virtual Network Function Scheduler **13**

3.1 Assumptions 13

3.2 System Design 14
3.2.1 System Architecture 14
3.2.2 Scheduling Algorithm 16
3.3 Implementation 18
3.3.1 The Reference NFV Platform 19
3.3.2 UN*i*S Components 20
3.3.3 Alternative Implementation on a Different Environment 22

4 Evaluation **24**
4.1 Experiment Setup 24
4.1.1 Testbed 24
4.1.2 VNF Types 25
4.1.3 Workloads 25
4.1.4 Compared Approach 26
4.1.5 Evaluation Metrics 26
4.2 Evaluation Scenarios and Results 27
4.2.1 SFC with fixed and uniform cost VNFs 27
4.2.2 SFC with fixed but non-uniform cost VNFs 30
4.2.3 SFC with variable cost (traffic dependent) VNFs 32
4.2.4 Multiple SFCs and Multiple CPU Cores 32
4.3 Investigation into UN*i*S's Throughput Gap 33
4.4 The Impact of Interface Occupancy based Optimization 37
4.5 Discussion: Cost vs. Benefit 41

5 Conclusion and Future Work **42**
5.1 Conclusion 42
5.2 Future Work 43

References **44**

List of Tables

4.1 Results for Multiple SFCs across Multiple CPUs 33
4.2 Extra throughput from the reduced context switches (compared to kbatch-8) 37

List of Figures

1.1 Packet Processing Performance of SFCs using Linux Schedulers 2

3.1 System Architecture 14
3.2 UNiS Cycle Estimator 20

4.1 SFC with fixed and uniform cost VNFs 28
4.2 Latency with Medium VNFs 29
4.3 Latency results for Light and Heavy VNFs 30
4.4 VNF density on a single core with fixed cost VNFs in an SFC 31
4.5 SFC Composed of VNFs with fixed but non-uniform processing cost 31
4.6 SFC composed of VNFs with variable processing cost (function of packet size) under real traffic load from [1] 32
4.7 SFC length 5 with uniform cost VNFs 34
4.8 Context switches and CPU cycles in SFC length 5 with uniform light VNFs 36
4.9 Ratio of cache misses with light VNFs 37
4.10 Context switches and CPU cycles for an SFC with uniform (medium) VNFs 38
4.11 Impact of occupancy based optimization in variable cost SFC 39
4.12 Impact of Interface Occupancy based Optimization 40

Chapter 1

Introduction

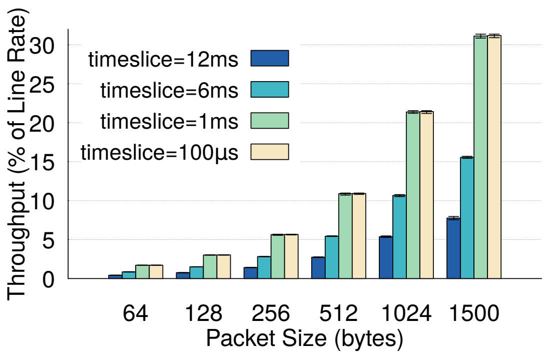
Network operators ubiquitously deploy proprietary, purpose-built and expensive hardware *middleboxes* [2] (*e.g.*, Firewalls, Intrusion Detection Systems, WAN optimizers, *etc.*) to realize various network services [3]. These middleboxes are a significant source of capital and operational expenditures for network operators because of their proprietary, vertically integrated and inflexible nature. This motivated the *Network Function Virtualization (NFV)* movement in the late 2012, which proposed to decouple Network Functions (NFs) from purpose-built hardware and run them as *Virtual Network Function (VNFs)* on commodity servers [4].

Moving NFs from specialized hardware to VNFs running on commodity servers comes with several challenges [5]. One key challenge is to achieve the same level of packet processing performance as that of the specialized hardware. A significant body of research has been dedicated to designing and implementing VNFs capable of line rate processing at tens of Gbps even for the smallest size packets [6–11]. A fundamental building block for these research works is the recently emerged fast packet processing libraries, such as netmap [12], Intel Data Plane Development Kit (DPDK) [13] that allow user-space programs to read/write packets directly from/to the Network Interface Card (NIC) bypassing the OS kernel, thus incurring very low I/O overhead.

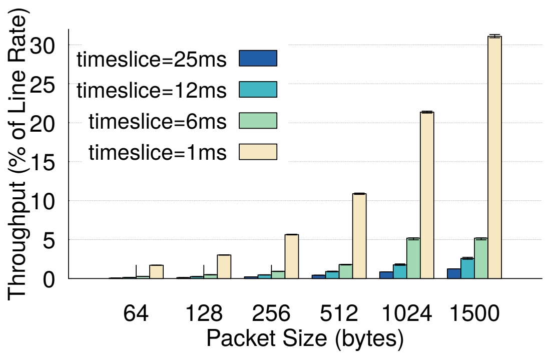
The most popular programming model for developing high performance VNFs leveraging these packet processing libraries is *poll-mode*, *i.e.*, VNFs continuously poll the NIC for incoming packets. Poll-mode VNF development has gained popularity in the last few years because it is simple to implement and incurs lower I/O overhead compared to a traditional interrupt driven model [8–11]. However, one caveat of this model is that the VNFs always utilize 100% CPU due to the continuous polling, even when there are no packets to process.

This makes it hard to relate CPU utilization of poll-mode VNFs to their packet processing cost. This continuous polling also renders the traditional kernel schedulers less effective since they heavily rely on CPU usages for taking scheduling decisions. Another drawback of existing kernel schedulers is that there is no interface to specify the desired processing order of VNFs. This is particularly important for scheduling VNFs sharing a CPU, since most network services are realized by steering packets through an ordered sequence of chained VNFs, known as a Service Function Chain (SFC) [14]. Due to these reasons, it is very common to see that most research leveraging poll-mode VNFs suggest to pin the VNFs to dedicated CPU cores [6] [7] [9]. This limits the number of VNFs that can be deployed on a machine to the number of available CPU cores. In this thesis, we address the problem of scheduling poll-mode VNFs on shared CPU cores in a way that maximizes the number of VNFs on a shared CPU core while maintaining high packet processing throughput.

1.1 Motivation



(a) CFS



(b) RT Scheduler

Figure 1.1: Packet Processing Performance of SFCs using Linux Schedulers

We perform an experimental study to demonstrate that out of the box OS schedulers fall short of efficiently scheduling VNFs in an SFC competing for the same CPU core. Note that this experimental study complements the motivational experiment presented in [15] by considering a VNF chain as opposed to individual VNFs sharing a core. We developed a lightweight DPDK-based VNF for this study, which performs bare-minimal packet processing (swaps the source and destination MAC addresses) to ensure that its processing overhead is not a performance bottleneck. The VNFs are chained by using a shared-memory based zero-copy packet exchange mechanism built using DPDK `rte_ring` library [16]. We deploy an SFC with three such VNFs, where the first two VNFs are pinned to the same CPU core and the third is pinned to a different one. The third VNF sends the packets out to the NIC, hence, was kept isolated from the other two to ensure there is no interference.

The machine used for this experiment is equipped with a 4 cores 3.3Ghz Intel Xeon E3-1230v3 CPU and a 10Gbps NIC, connected directly with a traffic generator. We generate traffic with varying packet sizes using pktgen-dpdk [17]. We evaluate two Linux schedulers, namely Completely Fair Scheduler (CFS) [18] and Real Time (RT) scheduler [19]. The throughput of the SFC is expressed as the percentage of the throughput of the same SFC with each VNF pinned to a different CPU (which was measured to be 10Gbps line rate for the smallest packet size of 64 bytes).

The results of this experiment are presented in Fig. 1.1. The first bar in each packet size represents the result obtained with the default scheduler parameters. For both CFS and RT schedulers, throughput is significantly low. For 64B packets, the throughput is $\approx$1% of line rate and even with Maximum Transport Unit (MTU) sized packets of 1500 bytes, it does not exceed $\approx$30% of line rate. Such poor performance can be explained as follows. In the case of CFS, the default configuration results in a `time_slice` of 12ms allocated to each VNF during a scheduling period, which we found to be too long. During this allocated time, a VNF fills up its outgoing interface very quickly. Since the outgoing interface becomes full, all the packets processed afterwards by the VNF are dropped, wasting the work already done by the current and previous VNFs. We also tune the `time_slice` allocated to VNFs by changing CFS parameters. We experimentally found that the default CFS does not support allocating less than 100μs `time_slice` to a process. As we can see from Fig. 1.1(a), even though throughput increases with reduced `time_slice`, it is still far from reaching line rate. Similar performance is also observed for the RT scheduler. Tuning RT scheduler parameters does not help much since we also experimentally found that the default minimum value of the parameters is in sub-millisecond precision. Moreover, it is important to note that neither CFS nor RT scheduler are able to enforce VNF execution order according to the SFC. This experimental study motivates a further examination of scheduling in NFV context.

A recent related work, NFVNice [15], addressed poll-mode VNF scheduling by proposing a mechanism to assign packet processing cost proportional CPU shares to VNFs. It also proposes to re-adjust assigned CPU shares to VNFs in an SFC when packets start dropping along the chain. However, NFVNice requires VNFs to be built using scheduler provided libraries to be able to monitor packet drops. Another VNF scheduling approach is to build the VNFs that can co-operate with other VNFs sharing a CPU by voluntarily yielding CPU at some carefully placed scheduling checkpoints in the code [20]. However, these solutions are *intrusive*, *i.e.*, require modifications to the VNFs to make them compatible with the scheduler, thus limiting the generality of the solution. To alleviate this limitation, we address VNF scheduling using a non-intrusive black box approach and design our scheduler to be workflow-aware, *i.e.*, preserving the VNF execution order in an SFC.

1.2 Design Goals and Challenges

Our goals for designing the scheduler and the associated challenges in achieving them are as follows:

- **Generic**. As mentioned in Section 1.1, the state-of-the-art approaches for scheduling poll-mode VNFs are *intrusive*, *i.e.*, require VNFs to be built with scheduler specific libraries that allow the scheduler to have insight into the VNF. Clearly, this approach benefits from more accurate VNF's internal information (*e.g.*, packet drop events, processing delay, *etc.*) that comes with lesser monitoring overhead. However, such requirement limits the generality of the solution and severely limits the type of VNFs that can work properly with the scheduler. We believe that a better and more generic approach is to be *non-intrusive*, *i.e.*, does not require VNFs to be built with any scheduler specific library or carefully placed scheduling checkpoints inside their code. However, the major challenge in devising a non-intrusive scheduler is to work with limited information about the events happening inside the VNFs, such as packet transmissions, packet drops, actual processing cost, *etc.*

- **High usability**. Our goal is not only to eliminate the requirement of modifying the VNFs to be compatible with the scheduler, but also to require minimal change to the operating system on which these VNFs are running. One option for implementing VNF scheduler is to make it part of the OS kernel by extending the existing OS scheduler or make it as another stand-alone scheduler in the kernel. However, this approach might take long time to be approved and integrated to the stable kernel release, otherwise it requires users to change purposely their OS kernel. Therefore, we choose to implement the scheduler in the user-space, increasing its usability. However, the challenge is to maintain the precision of our scheduler in spite of the additional overhead in accessing low level OS/hardware components via system-calls.

- **SFC-aware**, It is fundamental to ensure that VNFs are scheduled in the order they appear in the SFC, otherwise, it increases the chances of wasting CPU cyles due to not having sufficient work to perform when a VNF is scheduled. As we build our scheduler in the user-space we are still leveraging the existing OS scheduler to some extent. As discussed in the previous section, the underlying Linux schedulers can not guarantee the order of execution of VNFs sharing a core. Adapting the existing OS schedulers to enforce VNF execution order is a non-trivial task. Therefore, the challenge here is to devise a mechanism allowing VNFs to be scheduled according to their workflow in the SFC.

1.3 Contributions

The main contributions of this thesis are as follows:

- **System Architecture.** We propose UN*i*S a User-space Non-intrusive Workflow-aware Virtual Network Function Scheduler for poll-mode VNFs. To the best of our knowledge, UN*i*S is the first system that addresses poll-mode VNF scheduling problem in a non-intrusive manner, *i.e.*, without requiring the VNFs to be built with specific scheduling checkpoints or with any scheduler specific library.
- **System Implementation.** Closely following our proposed system architecture, we design and implement UN*i*S in C++ for scheduling DPDK-based VNF running as user-space processes on Linux operating system. Note that this is just one possible implementation of UN*i*S. The design of UN*i*S is generic and is not tied to any specific implementation.
- **Testbed Evaluation.** We perform an extensive evaluation of UN*i*S using multiple performance metrics, different types of VNFs and workloads under various deployment scenarios. Our key finding is that UN*i*S, in spite of its black box scheduling approach, is able to achieve a throughput within 90% (synthetic traffic) and 98% (real traffic) of that achieved using the intrusive cooperative scheduler.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

- Chapter 2 provides necessary background for this thesis. This chapter provides an overview of Network Function Virtualization, kernel bypass networking, DPDK-based packet processing and how process scheduling normally works in Linux kernel.

- Chapter 3 presents the UNiS system architecture, key components, scheduling algorithm, and implementation in details.
- Chapter 4 presents the results of our evaluation study of UNiS, and their analysis.
- Chapter 5 concludes this thesis with a summary of the main research contributions of this work, and an outline of the future work.

Chapter 2

Background

In this chapter, we first introduce the concept of Network Function Virtualization in Section 2.1. Then, we discuss the kernel bypass network I/O technologies in Section 2.2, followed by a brief overview of packet processing using intel DPDK in Section 2.3. We describe how the default Linux schedulers work in Section 2.4. Finally, related work in NFV scheduling is discussed in Section 2.5.

2.1 Network Function Virtualization

Traditional *Network Functions (NFs)* are deployed as vertically integrated, proprietary hardware network appliances or middleboxes to offer various network services. Examples of these middleboxes include Intrusion Detection System, WAN Optimizer, Load Balancer, *etc*. A recent study shows that the number of different middleboxes is comparable to the number of routers deployed in enterprise and data center networks. Although middleboxes have become an integral part in modern networks, they come with high Capital Expenditure (CAPEX) and Operational Expenditure (OPEX) because they are usually expensive, vendor specific, vertically integrated and require trained personnel for deployment and maintenance.

Network Function Virtualization (NFV) proposes to decouple Network Function from purpose-built and vertically integrated hardware middleboxes and run it as software on high volume commodity servers, known as *Virtual Network Function (VNF)*. These VNFs can be located in datacenters, network nodes or in the end user premises [4]. Shifting from hardware middleboxes to NFV enables network operators to consolidate multiple VNFs onto the same commodity servers. The ability to consolidate VNFs lower the network operators' upfront investment required for deploying new network services and reduces their CAPEX. Furthermore, NFV reduces operator's OPEX by enabling on-demand service provisioning and scaling which results in more efficient use of the infrastructures [5].

The application areas of NFV include but are not limited to mobile networks (*e.g.*, Evolved Node B, Home Location Register/Home Subscriber Server, *etc.*), traffic analysis (Deep Packet Inspection, QoE measurement), service assurance, SLA monitoring, application performance enhancement (*e.g.*, Content Delivery Networks, load balancer, WAN optimizer, *etc.*), and security functions (*e.g.*, firewalls, intrusion detection system, spam protection, etc). Furthermore, NFV is highly complementary to Software Defined Networking (SDN), and they are not dependent on each other. The separation of control and data plane provided by SDN, combined with the softwarized network functions provided by NFV can enhance performance, simplify deployments and facilitate better operation and maintenance procedures. [4]

2.2 Kernel Bypass Network I/O Technologies

In a general server environment, network packets arriving at the Network Interface Card (NIC) are first stored in the circular ring buffer by Direct Memory Access (DMA), and software interrupts notify the kernel network stack to process the packets according to the protocol and transfer them to socket receive buffer, and finally are copied to the user-space application. Normally, this requires at least one memory copy to move data and metadata from kernel to user space and multiple interrupts are raised to notify CPU about the reception or transmission of packets [12]. This traditional kernel network stack operation incurs high overhead due to memory management, packet copying and interrupt handling, making it impossible to provide high packet processing performance for NFV. Experimental studies in [12] [21] show that packet forwarding with kernel network stack achieves very poor throughput of 0.49 to 0.78 Mpps for 64B packet size.

Another option is to implement NFs in the kernel-space, eliminating the packet copy and other overhead. However, it has several negative consequences including (1) the slow process of making something part of the kernel; (2) the current kernel networking itself is already complicated and bloated with features to support different protocols; (3) lastly, users are required to change the kernel whenever a new NF is implemented. Therefore, user-space is deemed to be the right place for implementing the NFs, but the problem is the kernel networking being the bottleneck.

As a result, fast packet processing libraries such as netmap [12], OpenOnload [22], Intel Data Path Development Kit (DPDK) [13] and FD.io [23] emerged recently as enabling technologies for realizing NFV. These libraries facilitate a rapid development of user-space programs that can read/write packets directly from/to the NIC bypassing the OS kernel. Bypassing the kernel eliminates the overhead of packet copying, interrupts and system-calls, thus resulting in a very low I/O overhead and higher performance.

2.3 Packet Processing with DPDK

Intel Data Path Development Kit (DPDK) is an open source library to facilitate fast packet processing in user-space. DPDK contains libraries for kernel-bypass packet I/O, lockless multi-producer multi-consumer circular queues (`rte_ring` library), and memory management (`rte_mempool` library) among others. The ring library can be used to create shared memory based abstractions for zero-copy packet exchange. DPDK also ships with a set of NIC specific poll-mode drivers (PMDs) that enable user-space program to bypass the OS kernel network stack and continuously poll the NIC for incoming packets. Under high incoming packet rate, one advantage of poll-mode I/O over interrupt driven I/O is that the former can perform packet I/O with significantly less CPU cycles per packet, while the latter incurs more cycles due to the high frequency of expensive interrupt handling. This causes the poll-mode I/O to have substantially higher throughput. However, the major drawback of this model is that packet processors result in 100% CPU utilization for polling the NIC, even if there are no incoming packets.

2.4 Process Scheduling in Linux

Completely Fair Scheduler (CFS) is the default process scheduler since the Linux kernel version 2.6.23. CFS ensures fair allocation of CPU time to the processes competing for a CPU core. CFS achieves this by maintaining the notion of virtual runtime for each competing process and schedules the process with the least used virtual time to run next. Once a process is scheduled, it is allocated `time_slice` amount of time to run until it is preempted. In CFS, the time allocated to a process depends on some configurable kernel parameters [24], namely: (i) `sched_min_granularity_ns`: minimum amount of time a process is allowed to be run on a CPU core before being preempted, (ii) `sched_latency_ns`: minimum period after which CFS takes a scheduling decision.

The scheduling period (`sched_period`), *i.e.*, the period after which CFS takes scheduling decisions is set to `sched_latency_ns` if the number of competing processes for a CPU (`n_tasks`) is less than (`sched_latency_ns/sched_min_granularity_ns`), otherwise, to (`n_tasks*sched_min_granularity_ns`). Each competing process then gets (`sched_period / n_tasks`) amount of CPU time within a scheduling period.

CFS performs context switches to ensure fairness among competing processes. An alternative scheduler in Linux kernel that is work conserving and causes lesser context switches is the Real Time (RT) scheduler. RT scheduler prioritizes the completion of individual processes, rather than ensuring fairness among competing processes. RT scheduler has two scheduling policies resulting in a process being preempted only after it has finished (first-in-first-out (FIFO) policy) or after its allocated time slice has expired (round-robin policy). Note that in the case of VNFs, processes running the VNFs are not expected to terminate by their own, but rather terminate based on external triggers (*e.g.*, end of service period). Therefore, the out of the box FIFO policy as currently implemented in the kernel is not suitable for VNF workload. RT scheduler with a round-robin policy has a number of tunable kernel parameters [24]. One relevant parameter is `sched_rr_timeslice_ms`, which determines the length of `time_slice` a process is allowed to run before the next one is scheduled in a round-robin fashion. However, we experimentally found that the minimum values for the `time_slice` value in both CFS and RT schedulers, *i.e.*, 100 μs and 1ms respectively, are still too large for VNF workload.

2.5 Related Work on NFV Scheduling

Scheduling has been extensively studied in various areas of systems and networking such as cluster scheduling [25–27], packet scheduling [28, 29], flow scheduling [30, 31] among others. What makes NFV scheduling different from other areas is that VNF processing cost depends on a multitude of factors including packet size, packet arrival rate, VNF configuration, and packet contents to name a few. In contrast, in other areas that are close to NFV scheduling (*e.g.*, packet/flow scheduling, joint compute-network scheduling) processing costs are much more predictable and are usually dependent on lesser number of variables (*e.g.*, flow completion time depends on the amount of data to transfer and available bandwidth). In this section, we discuss recent developments in scheduling with a particular focus on NFV and contrast UN*i*S with the state-of-the-art.

2.5.1 Analytical Models for NFV Scheduling

There has been substantial developments in addressing VNF scheduling from a theoretical point of view using different methodologies [32–37]. Riera *et al.*, presents one of the early integer program formulation for scheduling VNFs on a set of servers [32], which is limited in scalability. Mijumbi *et al.*, presents an optimization model to jointly map and schedule VNFs on physical machines [33]. They also propose to use a tabu search meta-heuristic to address the limited scalability of the optimization model. An extension to the previous problem that also jointly considers routing between VNFs was studied in [35] and [36]. Both proposals use a mixed integer linear program to optimally solve the problem and then use a tabu search meta-heuristic [35] and column generation [36] to improve the scalability of their solutions. Other variants of the VNF scheduling problem have been studied with different objectives (*e.g.*, minimizing service latency [34]) and have been solved using methods such as game theory [37]. The optimization models for different variations of VNF scheduling is focused on scheduling SFCs across multiple machines, considering the network topology, available compute and network resources *etc.* In contrast, UN*i*S's focus is to serve as a viable alternative to local OS schedulers for VNF scheduling. Moreover, the methods used in this line of research are more suitable for devising an offline execution plan rather than for online scheduling decision making at micro-second time, which is a key requirement for UN*i*S. An analytical model focusing on processor sharing among VNFs in a single server is presented in [38]. The objective of this work is to reduce the time an outgoing NIC remains idle. In contrast, our objective is to pack as many VNFs as possible on the CPUs and achieve comparable throughput to an intrusive scheduling mechanism.

2.5.2 Systems for NFV Scheduling

Flurries [39] and NFVNice [15] are two notable systems proposed for NFV scheduling. Flurries proposes a system for hybrid poll-mode and interrupt driven execution of DPDK based VNFs and combines that with using RT kernel scheduler. With this combination Flurries is able to significantly increase VNF density on a physical machine. In contrast, NFVNice [15] proposes a back-pressure based mechanism to slow down an SFC by setting Explicit Congestion Notification (ECN) bit inside packets when VNFs experience packet drops. However, both of these approaches are intrusive, *i.e.*, they require the VNFs to be built with scheduler provided library to get a better insight into the VNFs or assume usage of certain mechanisms by the VNFs (*e.g.*, set ECN bit in packet). Another approach is to write VNFs from scratch to co-operate with other VNFs for better scheduling (similar to [20]). This usually results in fewer context switches, however, requires carefully placed scheduling checkpoints inside the VNF code. These intrusive approaches limit the VNFs that can be used with a scheduler. In contrast, we adopt a black box approach in UNiS to work with a wider range of VNFs.

Chapter 3

UNiS: A User-space Non-intrusive Workflow-aware Virtual Network Function Scheduler

3.1 Assumptions

UNiS is designed for VNFs operating in a poll-mode, *i.e.*, continuously polling for incoming packets, rather than operating in an interrupt-driven manner. We assume UNiS to operate alongside a DPDK based NFV platform such as the one shown in Fig. 3.1. In this reference platform, the VNFs are chained using an abstract entity called *interface*. An interface is an abstraction over a finite storage with methods for pushing packets to and pulling packets from it in batches. A specific implementation of the interface can be based on virtual Ethernet (veth) pairs, shared-memory, *etc*. Our only assumption about the interface is that it can export the number of outstanding packets/bytes and the actual capacity of the underlying storage. This is a reasonable assumption since many existing system tools export similar information (*e.g.*, veth interfaces shaped by `tc` subsystem export queue occupancy information). Another abstract component, *flow classifier*, redirects incoming packets to the appropriate SFC. Flow classifier can be implemented in many ways such as in software or using specific NIC features [40]. The NFV platform considered here does not assume any specific implementation for the abstract components, hence, does not tie UNiS to any specific implementation.

As a first step to achieve non-intrusive workflow-aware VNF scheduling, we consider linear SFCs only and leave the case for general forwarding graphs for future extension.

Also, we assume VNF to CPU mapping for an SFC to be externally computed using one of many available algorithms [41].

UNiS is intended to be used as a local scheduler for VNFs deployed on a machine and does not consider a cluster-wide scenario. Indeed, a cluster-wide view will result in better scheduling decisions. However, being first to address VNF scheduling in a non-intrusive way, UNiS currently focuses on local scheduling (*i.e.*, an alternative to existing OS kernel and Intrusive schedulers) and leaves the cluster-wide case for future extensions.

3.2 System Design

3.2.1 System Architecture

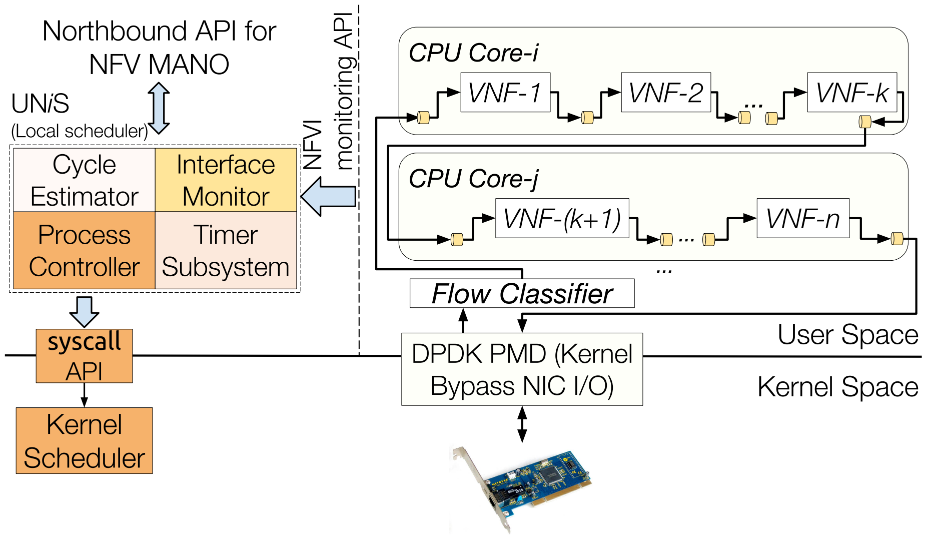


Figure 3.1: System Architecture

We design UNiS as a user-space VNF scheduler. This design choice has several benefits such as a faster development cycle and a high portability across different OSs. UNiS can also co-exist with existing OS schedulers, allowing them to schedule non-VNF processes. UNiS is expected to be part of every machine of an NFV infrastructure (NFVI). This way UNiS complements existing NFVI software [42] responsible for deploying and monitoring VNFs, and for creating VNF chains.

The system architecture of UNiS is presented in Fig. 3.1. UNiS exposes a northbound interface for the NFV Management and Orchestration (MANO) system (*e.g.*, similar to [43]) so that UNiS can be fed with SFC deployment information such as VNF to CPU core assignment, configuration of interfaces that connect the VNFs, *etc*. These information are typical to most NFV MANO systems, hence, do not restrict UNiS's generality. UNiS leverages the monitoring APIs exposed by existing NFVI software to monitor the interfaces connecting VNFs. This follows the ETSI NFV reference architecture [4]. Finally, UNiS uses OS provided system call APIs to interact with scheduling and process control subsystem in the kernel to control VNF execution states (*i.e.*, changing from running to waiting state). Apart from the different APIs for interaction, UNiS has the following key components:

Cycle Estimator

The cycle estimator component is responsible for profiling VNFs and estimate their processing cost in terms of the number of CPU cycles required to process each packet, which can also be translated to time dimension, *i.e.*, per packet processign latency. The processing cost of a VNF depends on a number of factors such as packet size, VNF configuration, packet content *etc*. [44–46]. An ideal cycle estimator should be able to take all such factors into account and provide an accurate estimate. The estimated cost of a VNF is used as an input to the scheduling algorithm for determining the `time_slice` allocated to that VNF.

Interface Monitor

To achieve the design goal of being generic, UNiS considers the VNFs as black boxes and relies on externally monitoring the interfaces that connect the VNFs. The interface monitor assumes that the underlying NFVI provides APIs to obtain the following statistics: (i) the number of outstanding packets in an interface connecting two VNFs; (ii) the maximum number of outstanding packets an interface can hold. These information are generic and are commonly exported by existing system tools. Note that with the availability of more types of statistics that the Interface Monitor could gather, *e.g.*, packet drop rate, incoming packet rate, *etc*., UNiS can potentially improve its scheduling decisions.

Timer Subsystem

The timer subsystem provides time accounting mechanism to UNiS to decide if a VNF has exhausted its allocated `time_slice`. The timer subsystem maintains a high precision timer in the user-space and is used for triggering events such as interface monitoring, VNF preemption, *etc*.. This subsystem is a key component in UNiS scheduling algorithm.

Process Controller

The process controller subsystem interacts with the underlying operating system to control the execution state of VNFs (*e.g.*, to start a waiting process or to preempt a running process). It should provide an efficient and reliable user-space mechanism that has low overhead and works under high frequency invocations. Since there are multiple ways to implement the process controller, even on the same OS, this subsystem should be able to support multiple implementations. As a result, it hides the underlying OS specific details from UNiS and porting UNiS to a different OS only requires adding another implementation with the corresponding system calls for the target OS to the existing process controller.

3.2.2 Scheduling Algorithm

At the core of UNiS, a scheduling algorithm makes scheduling decision for each CPU core. The scheduling algorithm leverages the components of UNiS to monitor the system, determines which VNF to run next and the `time_slice` allocation, and acts upon the VNFs to start/stop them. Some research has been dedicated to address VNF scheduling from a theoretical perspective [32–37]. However, these proposals are more suitable for devising an offline execution schedule and not for taking online scheduling decisions at the microsecond scale, which is a key requirement in UNiS. Therefore, we develop a lightweight yet effective scheduling algorithm for UNiS based on estimated `time_slice` allocation and interface occupancy between the VNFs in an SFC.

UNiS maintains a per CPU core wait queue of VNFs. When an external orchestrator invokes UNiS's northbound API with VNF to CPU mapping for a new SFC request, UNiS takes the VNFs in the order they appear in the SFC and places them in their corresponding CPU's wait queues.

The pseudo-code of UNiS's main scheduling loop is presented in Alg. 1. Before entering the main loop (line 3), it deploys the first VNF in each CPU's wait queue and creates the corresponding per core timer by leveraging UNiS's timer subsystem. `time_slice` allocated to a VNF v is computed as: $complexity(v) * \gamma * interface\_capacity(v.egress)$, where $complexity(.)$ gives us the estimated per packet processing time (profiled by UNiS's cycle estimator) required by v, and *interface_capacity(.)* gives us an interface's capacity to hold outstanding packets. This equation ensures that a VNF is given sufficient time to fill up its egress interface as close as possible to its full capacity, thereby maximizing throughput. $0 \leq \gamma \leq 1$ is a parameter used for leaving some head-room in the interface to

Algorithm 1: UNiS Scheduling Loop

```
   Input: $cores$ = Set of CPU cores; $\mathcal{T}$ = monitoring interval; $timer\_subsystem,$
          $process\_controller,$ $monitor$ = Handler to UN$i$S system components
1  function ScheduleVNFs()
2      timer_subsystem.monitoring_timer.start($\mathcal{T}$)
       /* The system is initialized by running the first VNF in every
          core's wait queue and creating corresponding per core timers.
          */
3      while $true$ do
           /* Take scheduling decision after every $\mathcal{T}$ $\mu$s                */
4          if $timer\_subsystem.monitoring\_timer.is\_expired() == $ false then continue
           /* Iterate over each core and check if a new VNF can be
              scheduled                                                    */
5          foreach $core \in cores$ do
6              $\mathcal{C} \leftarrow$ core.cur_vnf
7              if $core.timer.is\_expired()$ or $monitor.num\_pkts(\mathcal{C}.ingress) \leq \theta_{min}$ or
                $monitor.num\_pkts(\mathcal{C}.egress) \geq \theta_{max}$ then
                   /* Iterate over the wait queue ($\mathcal{WQ}$) and find a VNF that
                      has sufficient work to do                            */
8                  core.$\mathcal{WQ}$.push($\mathcal{C}$)
9                  $\mathcal{N} \leftarrow$ core.$\mathcal{WQ}$.pop()
10                 while $(\mathcal{C} \neq \mathcal{N})$ and $(monitor.num\_pkts(\mathcal{N}.ingress) \leq \theta_{min}$ or
                    $monitor.num\_pkts(\mathcal{N}.egress) \geq \theta_{max})$ do
11                     core.$\mathcal{WQ}$.push($\mathcal{N}$)
12                     $\mathcal{N} \leftarrow$ core.$\mathcal{WQ}$.pop()
13                 end
14             end
               /* If a candidate VNF is found, allocate a time_slice to it
                  */
15             if $\mathcal{C} \neq \mathcal{N}$ then
16                 core.timer.stop()
17                 time_slice $\leftarrow$ cost_estimator.get_cost($\mathcal{N}$) * $\gamma$ *
                    monitor.pkt_cap($\mathcal{N}.egress$)
18                 process_controller.deactivate($\mathcal{C}$)
19                 process_controller.activate($\mathcal{N}$)
20                 core.cur_vnf $\leftarrow \mathcal{N}$
21                 core.timer.reset(time_slice)
22             end
23         end
24     end
25     timer_subsystem.monitoring_timer.reset($\mathcal{T}$)
26 end
```

account for deviation of actual packet processing cost from the estimation. Then, UN*i*S monitors the system and takes a scheduling decision every $\mathcal{T}\mu$s.

During each scheduling interval, UN*i*S first checks if any of the CPUs has an expired timer, *i.e.*, the scheduled VNF needs to be preempted (line 7). Note that the incoming traffic rate is not considered during `time_slice` computation because the incoming rate of the SFC might be different from the incoming rate at each ingress interface of a VNF. Therefore, there can be situations where a VNF does not have sufficient packets to process (*i.e.*, its ingress interface has less than θ_{min} packets), or the outgoing interface is close to becoming full (*i.e.*, its egress interface has more than θ_{max} packets outstanding), even if the `time_slice` allocated to that VNF has not expired. We account for these conditions when determining if a VNF should be preempted or not (line 7). When such a VNF is found, we iterate over the CPU's wait queue and find a candidate VNF for scheduling that has more than θ_{min} packets in its ingress and less than θ_{max} packets in its egress interfaces (lines 8 – 14). Such selection criterias avoid wasted CPU cycles and unnecessary context switches by ensuring that the selected VNF has meaningful work to do when scheduled. We refer to this added optimization as the *Interface Occupancy based Optimization*. Once a proper candidate VNF is found, Alg. 1 interacts with the process controller to preempt the currently running VNF and schedule the next one.

3.3 Implementation

We have implemented a prototype of UN*i*S in C++ to work alongside a DPDK-based implementation of the reference NFV platform from Fig. 3.1. The reference NFV platform uses DPDK PMDs for packet I/O, `rte_ring` and `hugetlbfs` [47] to create shared memory between VNFs facilitating zero-copy packet exchange. This shared memory based interface has the capacity to hold 2048 packet references at a time and facilitates I/O in batches of up to 64 packets (*i.e.*, $batch\_size = 64$). In the implementation of UN*i*S scheduling algorithm, we set γ to 0.75 so that a newly scheduled VNF gets sufficient time to fill its egress buffer with a substantial number of packets and avoid overflow or packet drop due to any inaccuracy in packet processing cost estimation. We set θ_{min} parameter to ($batch\_size$ - 8), and θ_{max} parameter to $interface\_capacity(v.egress) * \gamma$ for a VNF v. In the following, we first describe the implementation of the NFV platform that UN*i*S is built on, followed by the implementation of UN*i*S system components to address the goals and challenges discussed in Section 1.2.

3.3.1 The Reference NFV Platform

UNiS is built on an NFV platform that we implemented in house from scratch. Our NFV platform is a generic high performance DPDK-based system providing interfaces to instantiate chains of VNFs. DPDK allows packets received by the NIC to be brought directly into the user-space, specifically, to the hugepages in the *hugetlbfs* [47] that is made accessible to the VNFs.

Our NFV platform reads the configuration files of an SFC containing multiple VNFs, allocates the necessary memory, creates shared-memory based abstraction for packet exchange and instantiates the VNFs. Each deployed VNF is a user-space process running in a poll-mode model. As described in Section 2.3, these VNFs exchange packets between them via a lockless multi-producer multi-consumer circular queue provided by DPDK `rte_ring` library. Note that this `rte_ring` data structure does not contain the entire packet data, instead, it stores only pointers to the beginning of the data structure containing the actual packet. Therefore, packet movement between VNFs does not require copying whole packet content rather is performed through copying only the pointers. The circular queue data structure in our implementation can hold up to 2048 packet pointers.

A VNF starts polling its ingress interface and reads a batch of maximum 64 packets at a time. Inside the VNF we add an optimization to pipeline packet processing by prefetching a cacheline worth of packet data into L1 cache, *i.e.*, when packet i from a batch is being processed, a non-blocking cache prefetch instruction is issued to prefetch a cacheline worth of data from packet $i + m$ from the batch and warm up the L1 cache. After processing a batch, the VNF pushes these packets to its egress interface. However, if the egress interface is full, packets are dropped. Then the VNF loops back to the beginning and pulls the next batch of packets from its ingress interface.

In our NFV platform, we also have the provision to include scheduling checkpoints in the VNFs as an optional feature. Without the scheduling checkpoint, the VNF will continuously execute the poll-mode loop of pulling, processing and pushing packets. With the scheduling checkpoint, the running VNF will yield its CPU after processing and pushing a certain number of batches of packets. As soon as a VNF yields its CPU, one of the other VNFs pinned to the same core will be scheduled.

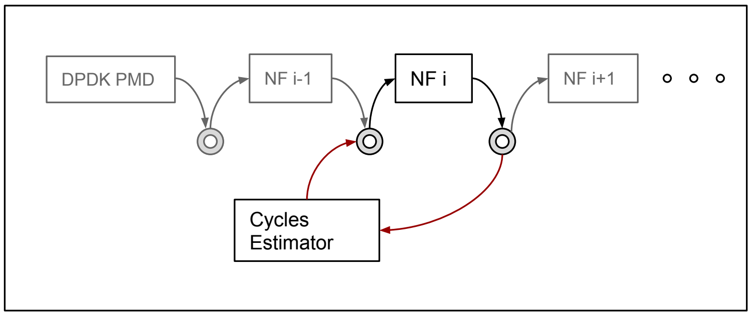


Figure 3.2: UN*i*S Cycle Estimator

3.3.2 UN*i*S Components

Cycle Estimator

We have implemented the Cycle Estimator to statically profile a VNF by pushing a batch of 64B packets into its ingress interface and then polling the egress interface to capture the batch of packets back and measures the time elapsed in between, as shown in Fig. 3.2. During the estimation process, the VNF is given a dedicated CPU core without other VNFs sharing that core. We are aware that this estimated cost is not the ideal representation of actual processing cost since I/O from and to the interfaces is included in the estimated cost. Furthermore, the actual cost depends on many factors such as packet size, VNF configuration, content of the packets, *etc.* To overcome this inaccuracy issue, we introduce the interface occupancy based optimization mentioned in Alg. 1 to fine tune the CPU time allocated to a VNF and leave dynamic adaptation of processing cost as a future work.

Interface Monitor

As mentioned earlier, the underlying NFV platform uses a shared memory based zero-copy abstraction to implement the interfaces facilitating VNF chaining. The NFV MANO system provides UN*i*S with SFC information that contains the configuration of the interfaces (*e.g.*, name of the shared memory region created by the external orchestrator and the interface memory capacities). After the initialization phase, the Interface Monitor uses `rte_ring` library to periodically read the ring occupancy and updates UN*i*S internal data structure that keeps track of this occupancy information. The aforementioned mechanism does not limit the generality of our solution. Similar APIs also exist for other Linux subsystems, *e.g.*, interfaces controlled by Linux `tc` also export similar information.

Timer Subsystem

We leverage DPDK's `rte_timer` library for a high precision time keeping in the user-space. The library is able to maintain its high precision by using the High Precision Event Timer (HPET), a hardware timer developed by Intel and Microsoft that has been integrated to their chipsets since 2005. In the absence of HPET, the library will use the CPUs Time Stamp Counter (TSC) registers to provide a reliable time reference [48].

Timers created by the rte_timer library use an asynchronous callback mechanism that we use to set a variable indicating timer expiration for a particular CPU core. We periodically poll this variable to check for timer expiry and trigger the necessary scheduling events. Once a timer is found to be expired, the VNF process running on that core will be preempted, the next VNF process will be run, and the timer is reset accordingly to the `time_slice` of the newly scheduled VNF. Currently, we poll the timer every 1μs, hence, can trigger scheduling events at 1μs granularity.

Before settling on using `rte_timer`, we explored the possibility of integrating *libevent* which is another library for implementing the timer subsystem. However, our early experiments demonstrated that libevent is unable to provide the micro-second level precision that we need.

Process Controller

A key challenge in implementing process controller in the user-space is to ensure a low overhead in switching processes. We tried several mechanisms for implementing this subsystem including Linux control groups (cgroups), POSIX signals, and using scheduler priority parameter (*sched_priority*) in RT scheduling with round-robin policy. This led us to develop a process controller library to allow multiple process controller mechanisms to co-exist under a common API. This approach enables programmers to easily select different mechanisms and add new process controller implementations in the future.

Linux cgroup is a Linux kernel feature that can be used to limit and isolate resource (*e.g.*, CPU, memory, disk I/O, and network) usage of a collection of processes. The common way of using cgroup is to create and interact with its virtual file system directly. However, we found that the cgroup file system hierarchy does not fit our need since under each group, we are restricted to tune the default OS scheduler parameters that do not have small enough minimum values. The relative shares between process groups in cgroup are not suitable for our proposed algorithm that tries to allocate `time_slice` proportional to the VNF processing cost.

POSIX signal allows a user-space process to send notifications to another process or a thread. In our case, we found `SIGSTOP` and `SIGCONT` signals to be potential candidates. `SIGSTOP` instructs the operating system to stop a process for later resumption, and `SIGCONT` instructs the operating system to continue the execution of a process. Our early experiments with the POSIX signal were successful when scheduling two processes with a relatively large `time_slice`. However, this approach does not work properly with more than two processes and with smaller `time_slice` allocation. Our conjecture is that the POSIX signal can not handle the high frequency (a few microseconds interval) invocations of these signals.

The final mechanism we implemented and also our mechanism of choice for UN*i*S is the following. We set the kernel to use the Linux RT scheduler with round robin policy (*i.e.*, *SCHED_RR*). With this setup, RT scheduler schedules the process with the highest priority at any moment and puts the other lower priority competing processes in the waiting state. When a different process is given the highest priority, RT scheduler swaps out the current process with the new highest priority process. In this way, we are able to control the execution state of VNFs by changing the *sched_priority* parameter. Note that VNFs are switched after every `time_slice` or less, which is computed by UN*i*S and is much smaller than the one assigned by RT scheduler. Therefore, there is no side-effect in using RT scheduler.

3.3.3 Alternative Implementation on a Different Environment

Although the implementation details elaborated so far are meant for Linux, it does not imply that the design and implementation of UN*i*S is specific to the Linux environment. In designing UN*i*S to achieve high usability, we do not leverage any specific implementation that only exist in Linux operating system. Therefore, we believe that the same concepts of building a user-space non-intrusive workflow-aware VNF scheduler can also be implemented for different environments. We implemented a prototype of UN*i*S in Linux because it is widely used in server deployments, and open-sourced with a rich ecosystem of development resources.

We conducted a study showing the feasibility of implementing UN*i*S for Windows server since it is also a commonly used OS for commercial servers. We found that official support for DPDK on Windows has already been initiated through `windpdk`, an experimental branch of DPDK. As demonstrated in DPDK Summit 2017 [49], `windpdk` already has the essential libraries, *i.e.*, `rte_eal, rte_mempool, rte_ring, etc.`, compiled and working on Windows Server 2016. Once the DPDK libraries required by UN*i*S are available in `windpdk`, the Cycle Estimator, Interface Monitor, Timer Subsystem components in UN*i*S can be directly used without any modification. For the Process Controller component, Windows has three options (*i.e.*, *system scheduler*, *user-mode scheduling (UMS)*, and *fibers*) to allow user-space scheduling. The most relevant one is *system scheduler*, a priority based scheduling mechanism that decides which of the competing threads receives the next processor time slice. The system assigns time slices in a round-robin fashion to all threads with the highest priority [50]. Additionally, the APIs for adjusting the priority for the specified process are also available [51]. Based on this study, we are confident that UN*i*S can be ported to Windows without requiring any changes in the design.

Chapter 4

Evaluation

We evaluate the performance of UNiS through testbed experiments. In the following, we first describe our experiment setup in Section 4.1. Then, we present our evaluation results on the effectiveness of UNiS's scheduling based on the following scenarios: (i) SFC with fixed and uniform cost VNFs (Section 4.2.1), (ii) SFC with fixed but non-uniform cost VNFs (Section 4.2.2), and (iii) SFC with variable cost (traffic dependent) VNFs (Section 4.2.3), and (iv) one or more SFCs deployed across multiple CPU cores (Section 4.2.4). Furthermore, we investigate factors that affect the throughput of UNiS in Section 4.3, and the impact of our interface occupancy based optimization in Section 4.4. We conclude this section with a discussion on cost vs. benefit of using intrusive and non-intrusive approaches in Section 4.5.

4.1 Experiment Setup

4.1.1 Testbed

Our testbed consists of two physical machines with identical configuration connected back to back without any interfering switch. One machine acting as the device under test hosts the VNFs and UNiS, while the other one is used for traffic generation. Each machine is equipped with a DPDK compatible Intel X710-DA 10Gbps NIC, 16GB of memory, and 3.3Ghz 4-core Intel Xeon E3-1230v3 CPU with L1 data and instruction caches each 4x32KB 8-way set associative, 4x256KB 8-way set associative L2 cache, and 8MB 16-way set associative L3 cache. When running UNiS, we isolate all the CPU cores except core

0 from the kernel scheduler and use them for VNF deployment, this way eliminating any conflict between the kernel scheduler and UNiS.

4.1.2 VNF Types

We use two types of VNFs in our experiments: (i) a fixed cost VNF whose packet processing cost is fixed and does not depend on packet size, (*e.g.*, similar to a layer 2-4 firewall), and (ii) a variable cost VNF whose packet processing cost is a function of packet size (*e.g.*, a WAN optimizer performing payload compression). For fixed cost VNFs, we use the same lightweight VNF used in Section 1.1 which reads and writes the ethernet addresses of a packet and add some imitated CPU intensive workload to emulate three different levels of packet processing cost, namely, *light* (50 cycles/packet), *medium* (150 cycles/packet), and *heavy* (250 cycles/packet). We profile the fixed cost VNFs by pushing smallest size packets and measuring the packet processing latency, and use this as their cost during scheduling. For the variable cost VNF, we implement a VNF whose packet processing cost is a step function of the packet size with the minimum cost similar to the cost in light VNF and the maximum cost equals to the cost in a heavy VNF. We profile the variable cost VNF using varying packet sizes ranging from 64 bytes to MTU size (1500 bytes) and consider the average packet processing latency over all sizes as their cost.

4.1.3 Workloads

We use pktgen-dpdk [17] and Moongen [52] for throughput and latency measurements, respectively. For throughput measurement, we generate traffic with different packet sizes, *i.e.*, ranging from smallest size (64 bytes) to MTU sized (1500 bytes) packets with pktgen-dpdk. We also use a real data center traffic trace (UNI1 trace [53] from [1]) to evaluate the effectiveness of UNiS under realistic traffic load. UNI1 trace is captured from a data center consisting of 500 servers and 22 switches, and exhibits a bi-modal packet size distribution centering around 200 bytes and 1400 bytes. During latency measurement with synthetic workload, we set the packet size to 128 bytes to provide some room for Moongen to embed a timestamp in some of the generated packets. We also set packet transmission rate to 80% of the maximum sustainable throughput in that deployment scenario to avoid the timestamped packets sent being dropped by the VNFs.

4.1.4 Compared Approach

First of all, we do not compare UNiS to the default OS scheduler because the performance of the default OS schedulers is too low as discussed in Section 1.1. We wanted to compare UNiS with NFVNice, however, the complete source code of NFVNice is not publicly available. Additionally, the performance results reported by NFVNice are very limited in terms of the system evaluation scenarios. They only provide an overall throughput report comparing NFVNice to OS schedulers with SFCs consisting of 3 VNFs with different processing costs. The fact that the heaviest VNF in their chain costs 550 cycles per packet limits their throughput to less than 2.5 Mpps, which is only 17% of their input traffic of 14.88 Mpps (10 Gbps line rate). We believe that the ideal VNF throughput should be close to line rate and able to allow more than 3 VNFs sharing one CPU core. Therefore, we compare UNiS with an intrusive co-operative scheduling approach similar to [20]. In this intrusive approach, the VNF is designed to voluntarily yield the CPU for other competing VNFs after processing k batches of packets. To decide the value for this parameter k, we run some experiments tuning the value and found that $k = 8$ is a good choice. With the voluntary yield after processing only 8 batches, the `time_slice` allocated to VNFs by OS scheduler does not have any impact on the VNF performance. Note that cooperative scheduling does not always guarantee VNF execution order according to an SFC. Therefore, we repeat each experiment with the intrusive approach for 5 times and report average result across all runs. The simplicity of intrusive co-operative scheduling offers very low overhead and provides a very efficient scheduling that has been used in embedded systems.

4.1.5 Evaluation Metrics

Throughput and Latency We measure the throughput and packet processing latency for both UNiS and the intrusive approach. We represent throughput as packets per second (pps) when using fixed packet size, or bits per second (bps) when using a mix of different packet sizes. For latency, we report the per packet average with 5th and 95th percentile latency values in μs. In addition to providing the result of these metrics, we also conduct an in-depth study investigating several operating system events, such as the number of context switches, cache misses ratio and CPU cycles consumed by a VNF to explain the observations from our experiments.

VNF density *VNF density* of a scheduling approach is measured by fixing a target throughput and determining the maximum length of an SFC chain (*i.e.*, number of VNFs) that can be deployed on a single CPU to sustain that throughput. This metric demonstrates a scheduling approaches ability to pack as many VNFs to a CPU core without degrading performance.

4.2 Evaluation Scenarios and Results

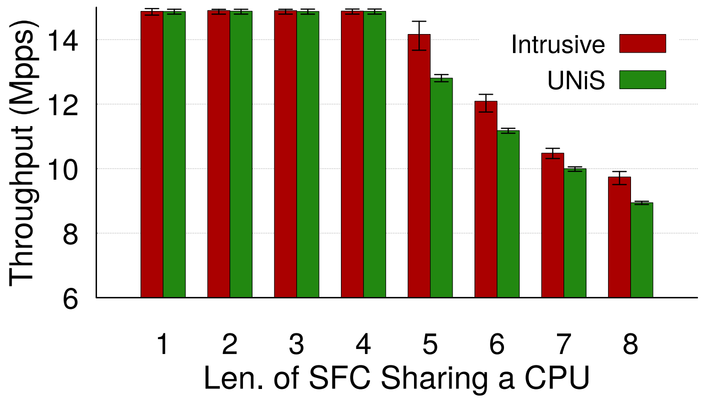
4.2.1 SFC with fixed and uniform cost VNFs

Our first set of experiments measure how much does the non-intrusive scheduling approach deviates from the intrusive approach in terms of throughput. We deploy SFCs of different lengths composed of identical VNFs with fixed packet processing cost (all light VNFs) on a single CPU core and present throughput results for the smallest (*i.e.*, 64 bytes) packet size in Fig. 4.1(a). Up to an SFC of length 4, both the intrusive approach and UN*i*S are able to sustain line rate. From length 5 and beyond, packet processing throughput drops below line rate and UN*i*S is not able to match that of the Intrusive approach. However, the deviation from the Intrusive approach was no more than 10% over all chain lengths. Note that the lighter the VNF the more the impact of accurate `time_slice` allocation. Therefore, this scenario with light VNFs measures the worst case performance deviation. In reality, with increasing VNF processing cost we expect the gap to be smaller. We confirm this hypothesis through another set of experimental results presented in Fig. 4.1(b) where we have the identical setup as before but use medium VNFs instead of the light ones. Since the VNFs are heavier, they cannot reach line rate processing in any case. However, the key observation here is that with increased packet processing cost UN*i*S's performance deviation from the intrusive approach is almost negligible (<2.5%).

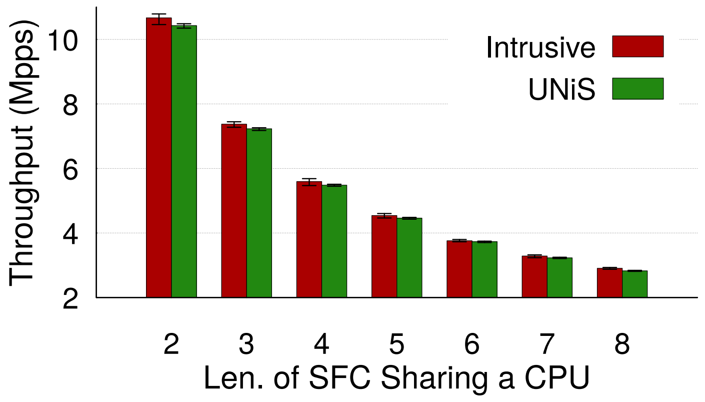
We designed UN*i*S prioritizing high throughput over low latency. However, we still perform a set of experiments to measure the extent of latency incurred by the packets. For SFC consisting of 3 medium VNFs (Fig. 4.2), the latency of UN*i*S is very close to the intrusive approach. However, for longer chain lengths, packets experience between 14% to 58% more latency on average in UN*i*S compared to the intrusive approach. The lower latency in the intrusive approach is because the CPU is voluntarily released after processing only 8 batches of packets, which avoids queue buildups. Note that the large error bar in the intrusive approach with longer SFCs is attributed to the inability to set the desired execution order of the VNFs sharing a CPU core.

The latency results for the light and heavy VNFs (Fig. 4.3) show different trends compared to the latency result of the medium VNFs. In the case of light VNFs scheduled by

UNiS, packets experience an average of 14 μs additional latency compared to the intrusive approach as shown in Fig. 4.3(a). We attribute this increase in latency to the following factors: (1) the impact of out of order execution of VNFs is not evident in SFC with light VNFs. (2) the difference between the estimated packet processing cost with the actual processing cost that causes `time_slice` allocation is not as intended; (3) since the light VNFs process packets faster, queue buildups in the interface connecting VNFs are growing faster and can cause packet drop. This could happen when the allocated `time_slice` is too long and the Interface Monitor has not updated its statistics.



(a) Throughput with Light VNFs



(b) Throughput with Medium VNFs

Figure 4.1: SFC with fixed and uniform cost VNFs

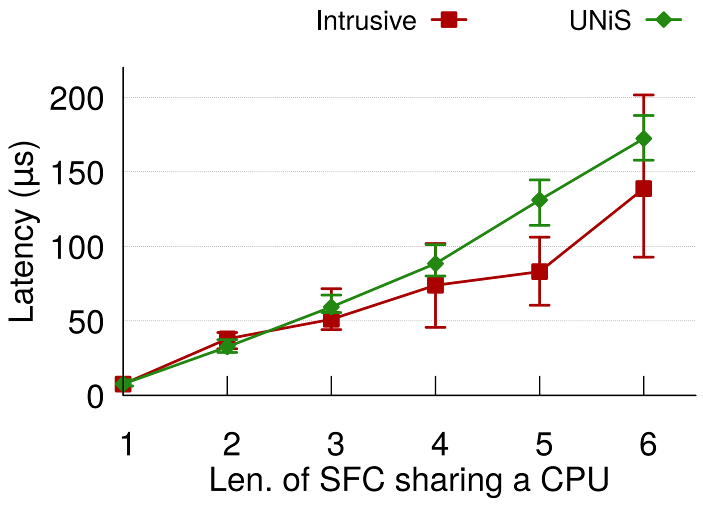
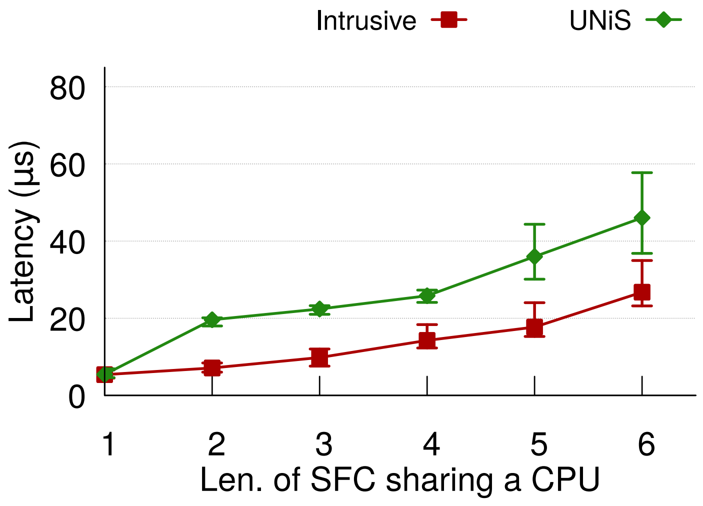


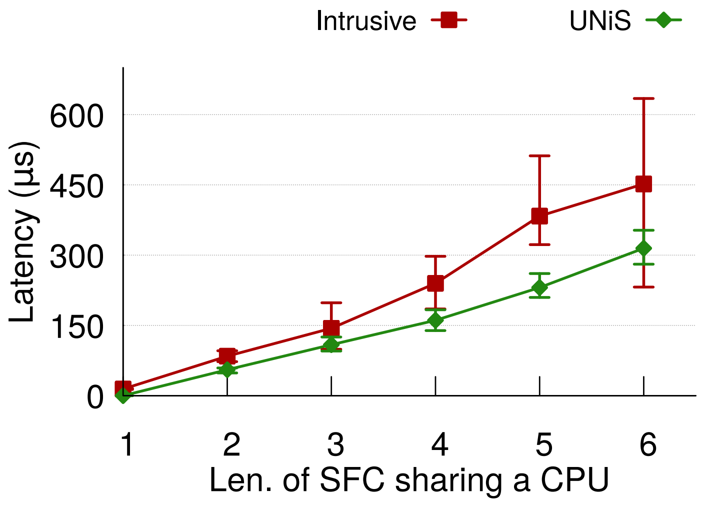
Figure 4.2: Latency with Medium VNFs

For SFC composed of heavy VNFs, we observe an opposite pattern, *i.e.*, average packet processing latency is lower when VNFs are scheduled by UN*i*S compared to using intrusive approach. This is due to the fact that the cooperative intrusive approach is unable to always guarantee VNF processing order according to the SFC. Because of out of order execution, packets have to stay longer in the interfaces until the appropriate VNF is scheduled and processes them. Unlike light and medium VNFs, heavy VNFs have higher packet processing cost leading to larger CPU time allocation during each scheduling round, which amplifies the penalty of such out of order execution. For instance, consider 5 VNFs chained together forming a SFC of VNF1 $\rightarrow$ VNF2 $\rightarrow$ VNF3 $\rightarrow$ VNF4 $\rightarrow$ VNF5, the worst execution order in this case is when VNFs are scheduled in the reverse order from VNF5 down to VNF1. The reason is that packets will come first to the VNF1, and then will be processed and placed to the interface connecting VNF1 and VNF2. However, the next scheduled VNF is VNF5 instead of VNF2, thus, VNF2 has to wait until VNF5, VNF4, VNF3 are scheduled and yielded the CPU. This also shows that UN*i*S being workflow-aware, provides additional advantage in terms of latency in scheduling more computational heavy VNFs.

We also present VNF density by varying the target throughput in Fig. 4.4. We conduct the experiment by using all three flavors of VNFs, *i.e.*, light, medium, and heavy. UN*i*S has identical VNF density in most cases compared to the intrusive approach. Even when UN*i*S packed less number of VNFs than the intrusive approach, the difference is less than 10%.



(a) Latency with Light VNFs



(b) Latency with Heavy VNFs

Figure 4.3: Latency results for Light and Heavy VNFs

4.2.2 SFC with fixed but non-uniform cost VNFs

In our next scenario, we deploy SFCs of different lengths with an alternating sequence of medium and heavy VNFs, *i.e.*, the VNFs at odd positions are the medium ones and

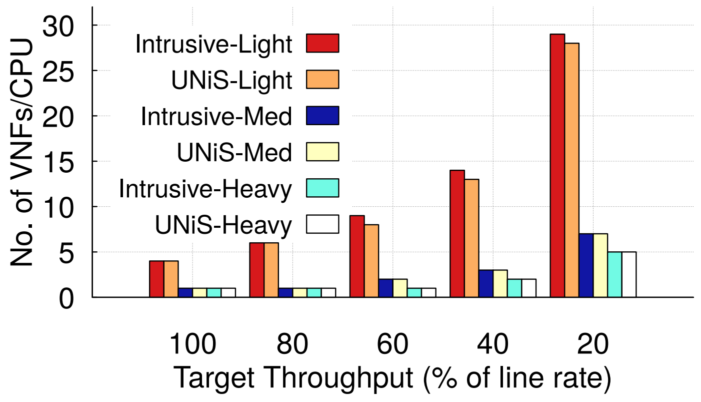


Figure 4.4: VNF density on a single core with fixed cost VNFs in an SFC

at even positions are the heavy ones. The goal of this experiment is to demonstrate the effectiveness of UNiS in handling heterogeneity in an SFC. The results of this experiment are presented in Fig. 4.5. As a result, UNiS is able to sustain a throughput that only deviates less than 2% from that of the intrusive approach for all chain lengths.

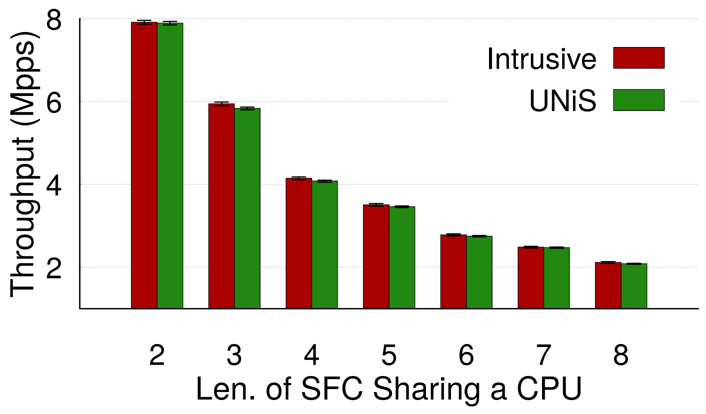


Figure 4.5: SFC Composed of VNFs with fixed but non-uniform processing cost

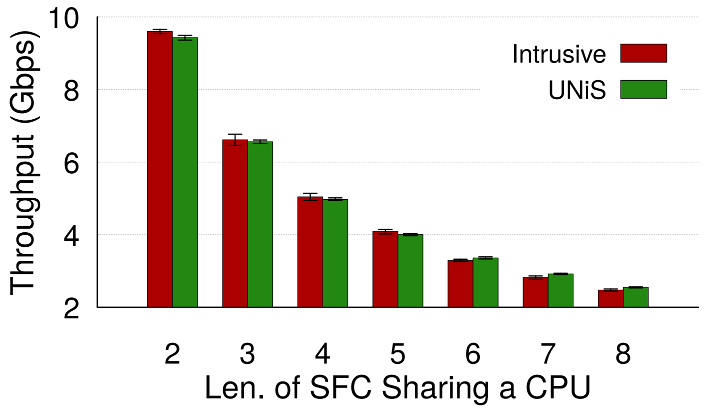


Figure 4.6: SFC composed of VNFs with variable processing cost (function of packet size) under real traffic load from [1]

4.2.3 SFC with variable cost (traffic dependent) VNFs

Previous experiments have not considered variable processing cost of a VNF based on traffic. However, many VNFs that operate on payloads can exhibit different processing costs depending on the packet size. To demonstrate the effectiveness of UN*i*S for such cases, we deploy SFCs composed of chains of variable cost VNFs described in Section 4.1.3. We play a real traffic trace containing packets of different sizes [1] and report the throughput in Fig. 4.6. UN*i*S performs very close to the intrusive approach with less than 2% deviation. The interface occupancy based optimization in Alg. 1 helps UN*i*S in this scenario to fine tune the `time_slice` allocated to the VNFs, which was computed using UN*i*S Cycle Estimator in the first place. We will further discuss the effectiveness of our interface occupancy based optimization in Section 4.4.

4.2.4 Multiple SFCs and Multiple CPU Cores

This evaluation scenario is intended to validate if UN*i*S causes any starvation while scheduling one or more SFCs spanning multiple CPUs. We deploy two SFCs (indicated by S1 and S2) consisting of all medium VNFs (*i.e.*, CPU limited) using the configurations described in Table 4.1. In Scenario (a), there are multiple SFCs deployed on a single core. With the intrusive approach, both SFCs achieve equal throughput of 5.31Mpps for 64B packets. We also observe a near equal throughput distribution across S1 and S2 for UN*i*S, indicating no SFC is starving for CPU. Scenario (b) has two SFCs deployed across two cores and each core hosts VNFs from two SFCs. Similar to (a), the intrusive approach shows equal throughput for both SFCs. We also observe similar behavior in this case for UN*i*S, validating the fact that no starvation is occurring when CPU cores are hosting VNFs from multiple SFCs and SFCs are deployed across multiple cores. Finally, scenario (c) deploys one SFC across multiple cores and here we see UN*i*S achieving a throughput within 1.3% of the intrusive approach.

# VNFs in SFC	**# VNFs on Core-1**	**# VNFs on Core-2**	**Int. Thput. (Mpps)**	**UN*i*S Thput. (Mpps)**

(a) S1 = 3
S2 = 1
	S1 = 3
	S2 = 1
		–	S1 = 5.31
			S2 = 5.31
				S1 = 5.30
				S1 = 5.21

(b) S1 = 4
S2 = 4
	S1 = 3
	S2 = 1
		S1 = 1
		S2 = 3
			S1 = 5.24
			S2 = 5.24
				S1 = 5.10
				S2 = 5.14

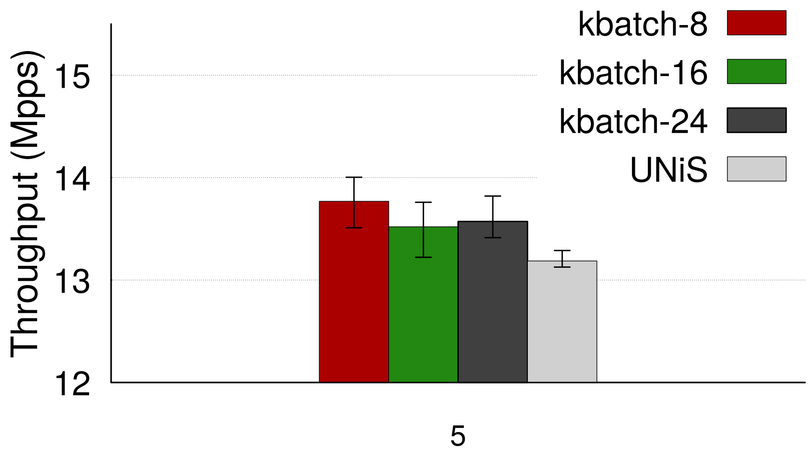
(c) S1 = 8	S1 = 4	S1 = 4	S1 =5.41	S1 = 5.34

Table 4.1: Results for Multiple SFCs across Multiple CPUs

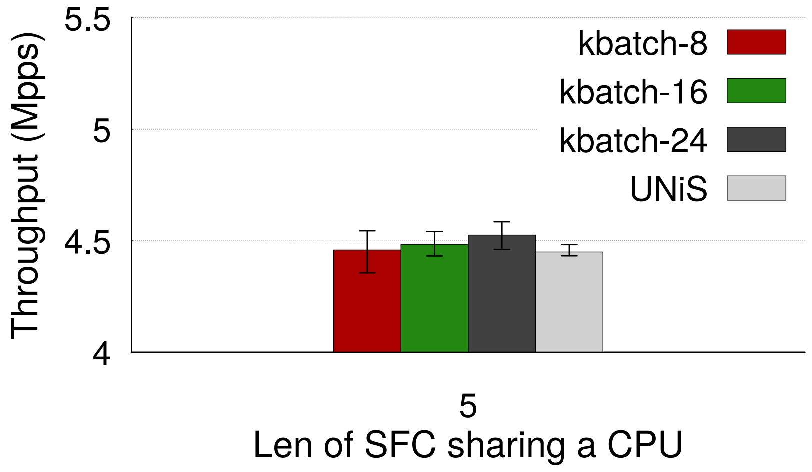
4.3 Investigation into UN*i*S's Throughput Gap

We observed a throughput gap of up to 10% between UN*i*S and the intrusive approach for an SFC composed of light VNFs sharing a CPU core (Section 4.2.1). To better understand the reason behind this gap, we measure several system-level metrics including the number of context switches, cache miss ratio and CPU cycles used by VNFs, which are known to have a major impact on system performance. We conducted experiments with varying SFC lengths up to 5 and observed similar trends for all lengths. Therefore, we only report our findings for SFCs of length 5.

In our compared intrusive approach from (Section 4.1.4), the VNF voluntarily yields CPU after processing every k batches of packets, which was set to 8 across all the previous experiments. However, the value of k heavily influences the number of times a VNF process switches context and also the cache access pattern. To better compare UN*i*S with the intrusive approach we also vary the value of k and show the performance difference with UN*i*S as well. In the following, we use the term kbatch-n to refer to an intrusive scheduling scenario with the value of k set to n. Note that kbatch-24 has the closest behavior to UN*i*S since both of these approaches try to fill up 75% capacity of the interface connecting adjacent VNFs.



(a) Light VNFs



(b) Medium VNFs

Figure 4.7: SFC length 5 with uniform cost VNFs

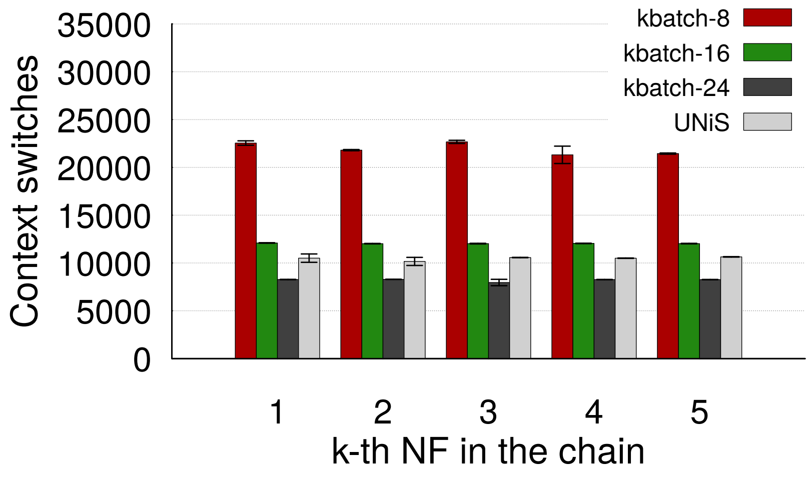
In Fig. 4.7, we present throughput measurements for different kbatch-n scenarios and UNiS, and notice different trends for SFCs composed from light and medium VNFs when scheduled with different kbatch-n scenarios. In Fig. 4.7(a), we see that kbatch-16 and kbatch-24 scenarios have lower throughput than kbatch-8, *i.e.*, our default intrusive scheduling scenario. Whereas in Fig. 4.7(b), increasing the value of k results in a higher throughput. This suggests that there are different key factors affecting the performance in the light and medium flavor VNFs. Note that in both cases UNiS has lower throughput than kbatch-8 scenario and throughput drop is more evident in the case of SFC composed of light VNFs compared to the one with the medium VNFs.

Our first step in investigating this throughput gap is to look at the number of context switches experienced by each VNF along the chain. As shown in Fig. 4.8(a), UNiS has less than half context switches than the default intrusive approach. This finding is rather counter-intuitive since we expect better throughput with lesser number of context switches. Therefore, this result alone does not explain the throughput gap and we continue looking for other factors in the OS.

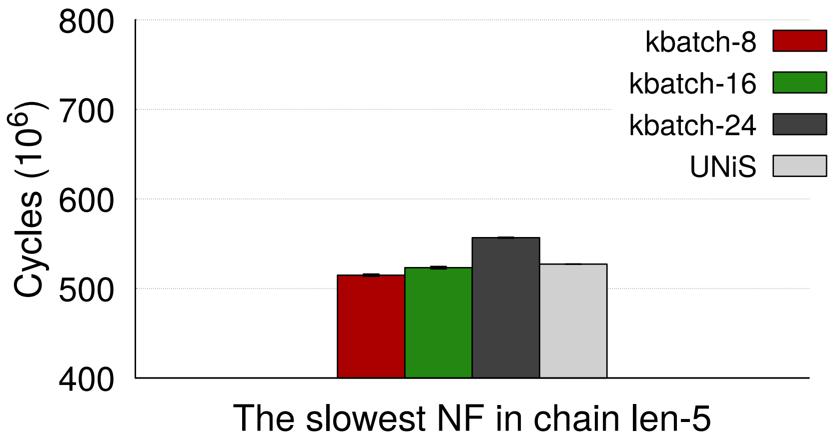
We then measure the actual CPU cycles consumed by each of the VNFs within a 1-second time window using *perf* tool [54]. Here, we focus on the slowest VNF in the chain since the throughput of the SFC is limited by the slowest VNF. The results are presented in Fig. 4.8(b). Despite the lesser number of context switches, the slowest VNF in UNiS, kbatch-16, and kbatch-24 get marginally more CPU cycles than the default intrusive scheduling scenario within the same 1-second time window. This suggests that the savings from the reduced number of context switches is not significant enough to substantially improve packet processing throughput.

Finally, we analyze the cache access pattern of the light VNFs in the SFC and present the results in Fig. 4.9 in terms of the percentage of cache misses over total cache references for each of the VNFs along the chain. Note that the synthetic workload with 64B packet generated at 10Gbps line-rate is used here, and for each packet, the VNF only accesses the packet header occupying 2 cache lines (128 B). We observe a trend where UNiS and intrusive approach with larger k exhibit more cache misses across the VNFs along the chain. The first three VNFs in the chain have higher cache-miss percentage because they are still warming up the cache. A closer look to the last VNF in the chain reveals that UNiS's cache-miss percentage (9.8%) is almost double that of kbatch-8 (4.9%), *i.e.*, the default intrusive scheduling scenario. This behavior is attributed to UNiS processing more batches of packets compared to the default intrusive case. Processing more batches also increases the chances of evicting previously cached packets from the CPU cache hierarchy. This increased percentage of cache misses combined with not so significant savings in CPU cycles from context switches contribute to reducing packet processing throughput of UNiS.

Although the throughput gap in the SFC with medium VNFs is almost negligible, we still conduct a study on context switches and CPU cycles to find a proper explanation and better understand the behavior of our system. The context switches behavior observed in



(a) Context switches in 1 second



(b) CPU cycles consumed in 1 second

Figure 4.8: Context switches and CPU cycles in SFC length 5 with uniform light VNFs

SFC with medium VNFs (Fig. 4.10(a)) is similar to the light VNF case discussed above. Fig. 4.10(b) shows that the CPU cycles consumed by the slowest VNF in a 1-second time window in UNiS is marginally more than that of the kbatch-8 scenario. This implies again that the CPU cycles saved by the reduced number of context switches are not significant

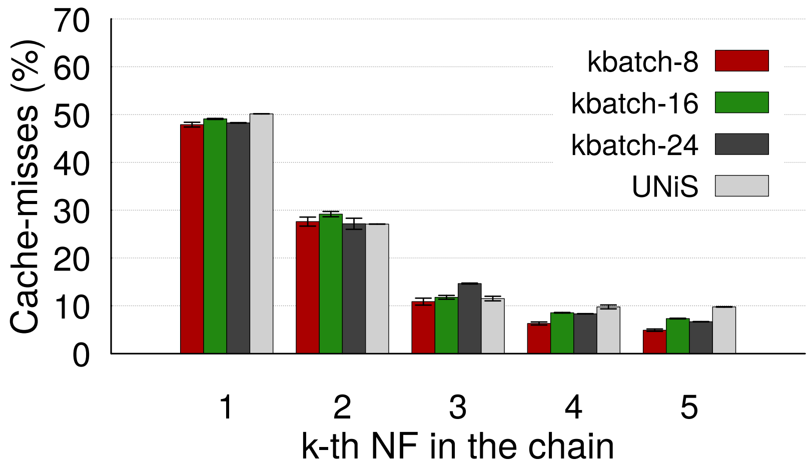


Figure 4.9: Ratio of cache misses with light VNFs

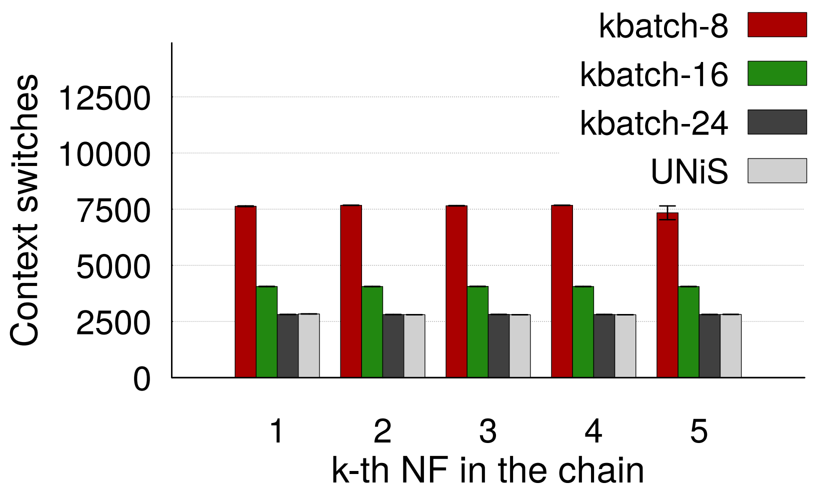
compared to the CPU cycles required to process a batch of packets. We calculate the expected increase in throughput from these extra cycles and present them in Table 4.2. In the case of UNiS the extra cycles translate to just 0.067 Mpps throughput increase, which is insignificant.

	Extra cycles per sec	**Increase in TP (mpps)**
UNiS	10,080,998	0.067
kbatch-16	27,235,101	0.181
kbatch-24	30,343,564	0.202

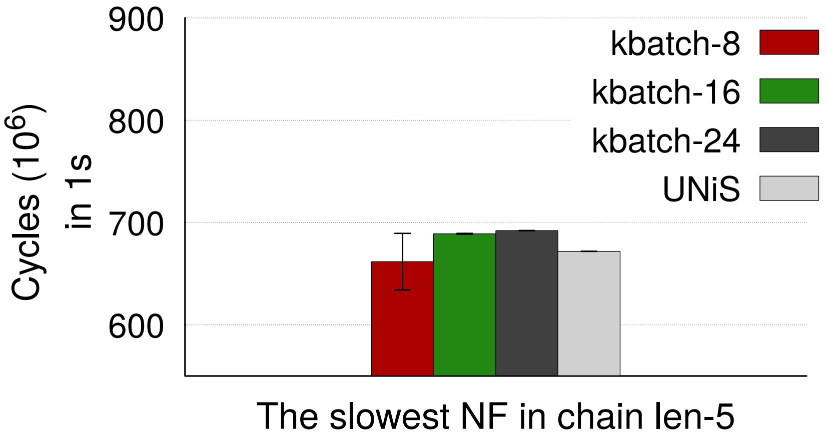
Table 4.2: Extra throughput from the reduced context switches (compared to kbatch-8)

4.4 The Impact of Interface Occupancy based Optimization

As mentioned in Section 3.3 and Section 3.2.2, the allocated `time_slice` to a VNF should be large enough to fill the VNF's egress buffer close to its full capacity, yet not too much to cause packet drops. The γ parameter used in `time_slice` calculation allows us to adjust the `time_slice` allocation, which is intended to compensate the inaccuracy introduced



(a) Context switches in 1 second



(b) CPU cycles consumed in 1 second

Figure 4.10: Context switches and CPU cycles for an SFC with uniform (medium) VNFs

because of the static profiling approach taken by UN*i*S's cycle estimator. In this section, we investigate the impact to the overall system throughput when the allocated `time_slice` is too small or too large and how the interface occupancy based optimization helps in alleviating the impact of inaccurate `time_slice` allocation.

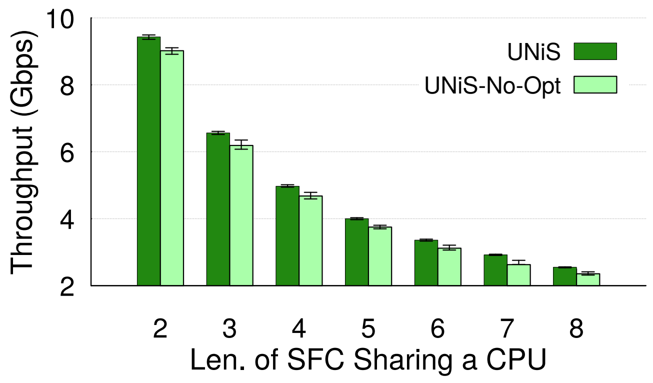
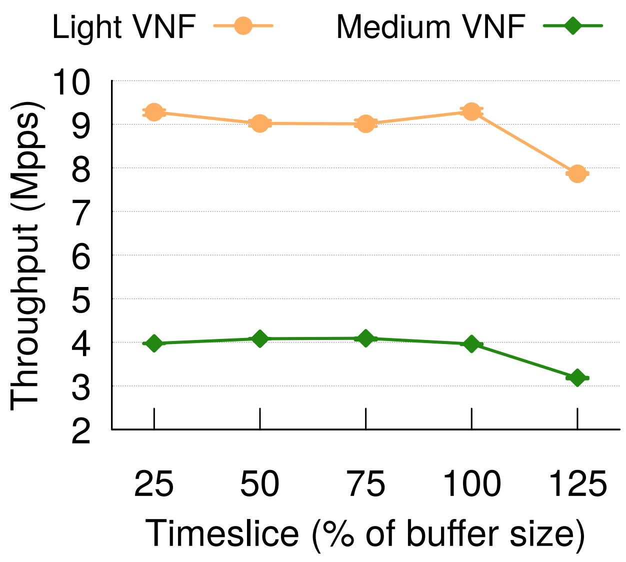


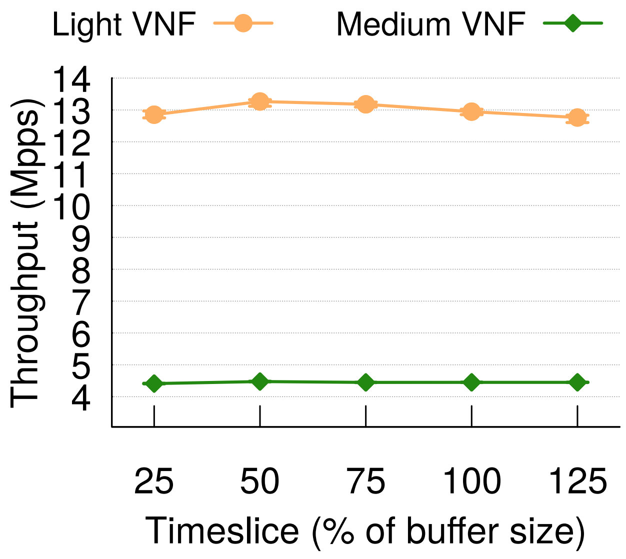
Figure 4.11: Impact of occupancy based optimization in variable cost SFC

We perform experiments with and without the interface occupancy based optimization and show its impact using an SFC consisting of variable cost VNFs with real traffic trace in Fig. 4.11. In this case, the allocated `time_slice` can sometimes overflow the interface when the processing cost is low (small packet size), or produce less batches of packets when the processing cost is high (large packet size). The added optimization results in as much as ≈10% performance improvement, which is relatively significant in absolute terms when packets are being processed at a rate of tens or hundreds Gbps.

We perform another experiment where we deploy an SFC composed of 5 VNFs pinned on a single CPU core and vary the allocated `time_slice` from the smallest value that is enough to fill only 25% of the interface to the largest value that can fill 125% the interface, *i.e.*, will cause packet drop. We evaluate UN*i*S with and without the interface occupancy based optimization for both light and medium VNFs. The experiment result with UN*i*S without occupancy based optimization in Fig. 4.12(a) shows that smaller value or `time_slice` does not have any significant impact on the throughput. However, when the allocated `time_slice` is larger than the time it takes to fill the whole interface (125% interface capacity), the throughput of both the SFC consisting light VNFs and the SFC consisting medium VNFs experiences a sharp drop of 12% and 22%, respectively. In the case of UN*i*S with the interface occupancy based optimization (Fig. 4.12(b)), there is no sharp performance drop when the allocated `time_slice` is either too small or too large for both light and medium VNFs. Therefore, the interface occupancy based optimization is effective in offsetting the impact of incorrect `time_slice` allocation.



(a) UNiS without occupancy based optimization



(b) UNiS with occupancy based optimization

Figure 4.12: Impact of Interface Occupancy based Optimization

4.5 Discussion: Cost vs. Benefit

Our experimental results suggest that even with a non-intrusive approach, UNiS is able to schedule VNFs in an SFC to achieve a comparable performance to that of an intrusive approach. Intrusive approaches such as co-operative scheduling and the one described in [15] have the benefit of lower monitoring overhead. For instance, a co-operative VNF will have carefully designed scheduling points where it yields the CPU to other VNFs, thus alleviating the need for continuously monitoring it. Another example is, for a method similar to [15], the VNF can notify the scheduler about packet drop events, therefore, event based monitoring can be performed instead of continuous monitoring. However, the price to pay here is the lack of generality of the approach. In contrast, for an effective non-intrusive approach, the system needs to be monitored at a finer time-scale, resulting in additional resource consumption. For instance, we needed to dedicate a CPU core in UNiS for high-precision time keeping and monitoring. This is the cost paid for achieving a generic scheduler capable of working with a wider range of poll-mode VNFs.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Network Function Virtualization (NFV) with its potential in reducing CAPEX/OPEX has attracted researchers and industries to tackle challenges in realizing NFV platforms capable of provisioning and managing high performance VNFs. Scheduling VNFs sharing a CPU core efficiently has become an important research problem. The state-of-the-art VNF scheduling systems can not support generic poll-mode VNFs and require VNFs to be built with schedulers' specific libraries. In this thesis, we presented UN*i*S, a user-space non-intrusive workflow-aware VNF scheduler. UN*i*S does not require any kernel modification, treats poll-mode VNFs as a black box, and considers VNF execution order in an SFC for scheduling. The proposed system architecture and components of UN*i*S are designed to be generic and not tied to a specific implementation. As a proof of concept, we implemented UN*i*S in C++ on Linux operating system with a DPDK-based NFV platform. In the evaluation, we compared our implementation of UN*i*S with an intrusive co-operative scheduler on a testbed.

Testbed experiments show that UN*i*S is able to achieve a throughput within 90% to 98% of achievable throughput using an intrusive co-operative scheduler, depending on the types of VNFs and the workload. In terms of latency, UN*i*S incurs up to 58% additional latency for SFC consisting of medium VNFs. However, for SFC consisting of heavy VNFs, packets scheduled by UN*i*S experience 12% lower latency on average than using the intrusive approach. Moreover, our extensive study of the relation between cache misses, context switches and the consumed CPU cycles shows that a cache intensive (light) VNF with longer `time_slice` is more sensitive to throughput degradation due to the impact of more cache misses. Additionally, we found that without the interface occupancy based optimization, UNiS experience a throughput degradation between 12% to 22% when the allocated `time_slice` is too large. While with the optimization, the throughput degradation is reduced to only 0.8% to 3%. This shows the effectiveness of our interface occupancy based optimization in UNiS. In summary, the promising experimental results demonstrate that even with a black box approach, UNiS is able to perform very close to the intrusive scheduling method.

5.2 Future Work

Building on these promising results, several possible extensions of this work are as follows:

Multi-node deployment. Currently, UNiS is a local scheduler that is only responsible for scheduling VNFs that are provisioned on a single node. In the case of SFCs that are deployed across multiple nodes, potential research avenues are on how to place these VNFs among the multiple nodes with possibly heterogenous hardware cofigurations and how to schedule these VNFs in this scenario considering factors that did not exist in a single node deployment, e.g., inter-node latency.

Consideration for arbitrary SFC structure. In the design and evaluation of UNiS, we only considered linear SFCs. A more complex graph structure interconnecting the VNFs consisting of multiple branches could be considered in the future extension of UNiS.

Dynamic adjustment. Although our interface occupancy based optimization can workaround the problem of inaccurate packet processing cost estimation, an interesting future extension will be to consider factors such as packet size, packet content, and packet inter-arrival time for adjusting the `time_slice` or other scheduling decisions dynamically.

Achieving specified SLO. There are VNFs or SFCs, such as load balancer and web accelerator in Content Delivery Networks that have different levels of utilization during different times of the day. In this situation, there might be higher Service Level Objective (SLO) during the day, compared to the SLO during the night where the entire network traffic is lower. Therefore, as opposed to maximizing the throughput of the CPU sharing VNFs, an extension of UNiS could be to devise a mechanism for meeting certain SLOs while minimizing resource usage.

References

[1] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of ACM IMC.* ACM, 2010, pp. 267–280.

[2] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and issues," Internet Requests for Comments, RFC Editor, RFC 3234, February 2002. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3234.txt

[3] 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," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[4] "Network Functions Virtualisation – Introductory White Paper," White paper, Oct 2012. [Online]. Available: https://portal.etsi.org/nfv/nfv_white_paper.pdf

[5] 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 & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[6] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *Proceedings of USENIX NSDI.* USENIX Association, 2014, pp. 459–473.

[7] J. Hwang, K. Ramakrishnan, and T. Wood, "NetVM: high performance and flexible networking using virtualization on commodity platforms," in *Proceedings of USENIX NSDI.* USENIX Association, 2014, pp. 445–458.

[8] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A software nic to augment hardware," *Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155*, 2015.

[9] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of nfv." in *Proceedings of USENIX OSDI.* USENIX Association, 2016, pp. 203–216.

[10] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "OpenNetVM: A platform for high performance network service chains," in *Proceedings of ACM HotMiddlebox.* ACM, 2016, pp. 26–31.

[11] Z. A. Qazi, M. Walls, A. Panda, V. Sekar, S. Ratnasamy, and S. Shenker, "A high performance packet core for next generation cellular networks," in *Proceedings of ACM SIGCOMM.* ACM, 2017, pp. 348–361.

[12] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *Proceedings of USENIX ATC*, 2012, pp. 101–112.

[13] "Intel data path development kit," https://www.dpdk.org/. [Online]. Available: https://www.dpdk.org/

[14] J. Halpern and C. Pignataro, "Service function chaining (sfc) architecture," Internet Requests for Comments, RFC Editor, RFC 7665, October 2015.

[15] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "NFVnice: Dynamic backpressure and scheduling for nfv service chains," in *Proceedings of ACM SIGCOMM.* ACM, 2017, pp. 71–84.

[16] "Dpdk rte_ring," https://dpdk.readthedocs.io/en/v16.04/prog_guide/ring_lib.html. [Online]. Available: https://dpdk.readthedocs.io/en/v16.04/prog_guide/ring_lib.html

[17] "pktgen-dpdk," http://git.dpdk.org/apps/pktgen-dpdk/. [Online]. Available: http://git.dpdk.org/apps/pktgen-dpdk/

[18] "Linux cfs," https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt. [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

[19] "linux-rt," https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt. [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt

[20] "DPDK L-thread Subsystem Example," http://doc.dpdk.org/guides-18.05/sample_app_ug/performance_thread.html#lthread-subsystem. [Online]. Available: http://doc.dpdk.org/guides-18.05/sample_app_ug/performance_thread.html#lthread-subsystem

[21] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*. IEEE, 2015, pp. 5–16.

[22] "openonload," https://www.openonload.org/. [Online]. Available: https://www.openonload.org/

[23] "Fd.io: The universal data plane," https://fd.io/. [Online]. Available: https://fd.io/

[24] "Tuning the task scheduler," https://doc.opensuse.org/documentation/leap/tuning/html/book.sle.tuning/cha.tuning.taskscheduler.html. [Online]. Available: https://doc.opensuse.org/documentation/leap/tuning/html\/book.sle.tuning/cha.tuning.taskscheduler.html

[25] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 265–278.

[26] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós, "On scheduling in map-reduce and flow-shops," in *Proceedings of ACM SPAA*, 2011, pp. 289–298.

[27] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of ACM EuroSys*. ACM, 2015.

[28] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *Proceedings of ACM SIGCOMM*. ACM, 2016, pp. 44–57.

[29] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *Proceedings of USENIX NSDI*. USENIX Association, 2016, pp. 501–521.

[30] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *Proceedings of USENIX NSDI*. USENIX Association, 2010.

[31] M. Chowdhury, Y. Zhong, and I. Stoica, “Efficient coflow scheduling with varys,” in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 443–454.

[32] J. F. Riera, X. Hesselbach, E. Escalona, J. A. Garcia-Espin, and E. Grasa, “On the complex scheduling formulation of virtual network functions over optical networks,” in *Proceedings of ICTON*. IEEE, 2014, pp. 1–5.

[33] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. D. Turck, and S. Davy, “Design and evaluation of algorithms for mapping and scheduling of virtual network functions,” in *Proceedings of IEEE NetSoft*, April 2015, pp. 1–9.

[34] L. Qu, C. Assi, and K. Shaban, “Delay-aware scheduling and resource optimization with network function virtualization,” *IEEE Transactions on Communications*, vol. 64, no. 9, pp. 3746–3758, 2016.

[35] H. A. Alameddine, L. Qu, and C. Assi, “Scheduling service function chains for ultra-low latency network services,” in *Proceedings of IEEE/ACM/IFIP CNSM*. IEEE, 2017, pp. 1–9.

[36] H. A. Alameddine, S. Sebbah, and C. Assi, “On the interplay between network function mapping and scheduling in vnf-based networks: A column generation approach,” *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 860–874, Dec 2017.

[37] C. Pham, N. H. Tran, and C. S. Hong, “Virtual network function scheduling: A matching game approach,” *IEEE Communications Letters*, vol. 22, no. 1, pp. 69–72, Jan 2018.

[38] G. Faraci, A. Lombardo, and G. Schembra, “An analytical model to design processor sharing for sdn/nfv nodes,” in *Proceedings of ITC*, vol. 02, Sept 2016, pp. 28–34.

[39] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, “Flurries: Countless fine-grained nfs for flexible per-flow customization,” in *Proceedings of ACM CoNeXT*. ACM, 2016, pp. 3–17.

[40] “Intel ethernet flow director,” https://software.intel.com/en-us/articles/setting-up-intel-ethernet-flow-director. [Online]. Available: https://software.intel.com/en-us/articles/setting-up-intel-ethernet-flow-director

[41] J. G. Herrera and J. F. Botero, "Resource allocation in nfv: A comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[42] "Network Functions Virtualisation (NFV); Infrastructure Overview," White paper, Jan 2015. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/001/01.01.01_60/gs_NFV-INF001v010101p.pdf

[43] "Network Functions Virtualisation (NFV); Management and Orchestration," White paper, Dec 2014. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf

[44] S. Lange, A. Nguyen-Ngoc, S. Gebert, T. Zinner, M. Jarschel, A. Köpsel, M. Sune, D. Raumer, S. Gallenmüller, G. Carle *et al.*, "Performance benchmarking of a software-based lte sgw," in *Proceedings of IEEE/ACM/IFIP CNSM*. IEEE, 2015, pp. 378–383.

[45] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "Nfv-vital: A framework for characterizing the performance of virtual network functions," in *Proceedings of IEEE NFV-SDN Conference*. IEEE, 2015, pp. 93–99.

[46] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "Resq: Enabling slos in network function virtualization," in *Proceedings of USENIX NSDI*. Renton, WA: USENIX Association, 2018, pp. 283–297.

[47] "hugetlbfs kernel documentation." [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt

[48] "Dpdk rte_timer," https://doc.dpdk.org/guides/prog_guide/timer_lib.html. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/timer_lib.html

[49] "Dpdk summit 2017," https://www.dpdk.org/wp-content/uploads/sites/35/2018/06/Making-networking-apps-scream-on-Windows-with-DPDK.pdf. [Online]. Available: https://www.dpdk.org/wp-content/uploads/sites/35/2018/06/Making-networking-apps-scream-on-Windows-with-DPDK.pdf

[50] "Windows system scheduler," https://docs.microsoft.com/en-us/windows/desktop/procthread/scheduling. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/procthread/scheduling

[51] "Windows process thread api," https://docs.microsoft.com/en-ca/windows/desktop/api/processthreadsapi/. [Online]. Available: https://docs.microsoft.com/en-ca/windows/desktop/api/processthreadsapi/

[52] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of ACM IMC*. ACM, 2015, pp. 275–287.

[53] "Data set for imc 2010 data center measurement," http://pages.cs.wisc.edu/˜tbenson/IMC10_Data.html. [Online]. Available: http://pages.cs.wisc.edu/~{}tbenson/IMC10_Data.html

[54] "Linux perf tool." [Online]. Available: https://perf.wiki.kernel.org/index.php/Tutorial

[55] G. Faraci, A. Lombardo, and G. Schembra, "A processor-sharing scheduling strategy for nfv nodes," *Journal of Electrical and Computer Engineering*, vol. 2016, pp. 1:1–1:1, Jan. 2016.

[56] M. Yoshida, W. Shen, T. Kawabata, K. Minato, and W. Imajuku, "Morsa: A multi-objective resource scheduling algorithm for nfv infrastructure," in *Proceedings of AP-NOMS*, Sept 2014, pp. 1–6.

[57] L. Qu, C. Assi, and K. Shaban, "Network function virtualization scheduling with transmission delay optimization," in *Proceedings of IEEE/IFIP NOMS*, April 2016, pp. 638–644.

[58] C. Zeng, F. Liu, S. Chen, W. Jiang, and M. Li, "Demystifying the performance interference of co-located virtual network functions," in *Proceedings of IEEE INFOCOM*, 2018.

[59] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-net: Effective gpu sharing in nfv systems," in *Proceedings of USENIX NSDI*. USENIX Association, 2018.