

Non-intrusive and Workflow-aware Virtual Network Function Scheduling in User-space

Anthony Anthony, Shihabur Rahman Chowdhury, Tim Bai, Raouf Boutaba, Jérôme François

▶ **To cite this version:**

Anthony Anthony, Shihabur Rahman Chowdhury, Tim Bai, Raouf Boutaba, Jérôme François. Non-intrusive and Workflow-aware Virtual Network Function Scheduling in User-space. IEEE Transactions on Cloud Computing, 2020, 10.1109/TCC.2020.3024232 . hal-02996459

HAL Id: hal-02996459
https://inria.hal.science/hal-02996459

Submitted on 9 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Non-intrusive and Workflow-aware Virtual Network Function Scheduling in User-space

Anthony, Shihabur Rahman Chowdhury, *Student Member, IEEE*, Tim Bai, Raouf Boutaba, *Fellow, IEEE*, and Jérôme François

Abstract—The simple programming model and very low-overhead I/O capabilities of emerging packet processing techniques leveraging kernel-bypass I/O and poll-mode processing is gaining significant popularity for building high performance software *middleboxes* (*aka Virtual Network Functions (VNFs)*). However, existing OS schedulers fall short in rightsizing CPU allocation to poll-mode VNFs due to the schedulers' shortcoming in capturing the actual processing cost of these VNFs. This issue 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 VNF schedulers proposed as an alternative to OS schedulers are *intrusive*, 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 these schedulers. In this paper, we present UN*i*S, a **U**ser-space **N**on-**i**ntrusive work-flow aware VNF **S**cheduler. Unlike existing approaches, UNiS is non-intrusive, *i.e.*, does not require VNF modifications and treats poll-mode VNFs as black boxes. UNiS is also workflow-aware, *i.e.*, takes SFC processing order into account while scheduling VNFs. Testbed experiments show that UN*i*S is able to achieve a throughput within 90% and 98% of that achievable using an intrusive co-operative scheduler for synthetic and real data center traffic, respectively.

Index Terms—Network Function Virtualization, Service Function Chaining, Scheduling

1 Introduction

Network Function Virtualization (NFV) proposes to decouple *Network Functions (NFs)* from purpose-built and expensive hardware *middleboxes* and run them as *Virtual Network Functions (VNFs)* on inexpensive commodity servers [1]. This paradigm shift from hardware-centric to software-centric architecture enables the network operators to lower their capital expenditure by consolidating multiple NFs on the same commodity hardware and reduce operational expenditure by enabling on-demand service provisioning leveraging cloud orchestration technologies [2], [3]. Since its inception in the late 2012, NFV is experiencing increasing adoption in data centers owned by large-scale online service providers [4], [5] as well as in the telecommunications central offices and Internet Service Provider (ISP) Point-of-Presences (PoPs) transformed into edge-clouds [6], [7] supporting a wide-range of use-cases.

A significant effort in early NFV research and development has been dedicated to close the performance gap between hardware middleboxes and VNFs running on commodity servers [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. A fundamental building block of these high performance VNFs is the emerging fast packet processing libraries such as Intel DPDK [18]. DPDK and similar libraries facilitate rapid development of user-space programs that can perform direct packet I/O on the Network Interface Card (NIC) bypassing the OS kernel, thus incurring very low I/O overhead.

These packet processing libraries adopt a poll-mode programming model where the VNFs need to continuously poll the NIC for incoming packets. Poll-mode VNF development has gained popularity in the last few years because of its simplicity and lower I/O overhead compared to traditional interrupt driven I/O model [10], [11], [13], [14]. However, a major 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 difficult to relate CPU utilization of poll-mode VNFs to their packet processing cost [19]. Another implication of continuous polling is that the CPU schedulers in existing OSs become ineffective for VNF scheduling since they heavily rely on CPU usages for making scheduling decisions. Furthermore, existing OS schedulers do not have any interface for specifying the desired processing order of VNFs. This is particularly important for scheduling VNFs on a shared CPU core since network services are typically realized by steering packets through an ordered sequence of VNFs, also known as a *Service Function Chain (SFC)* [20]. Due to these reasons, it is very common to see that poll-mode VNFs are pinned to dedicated CPU cores, *limiting the number of VNFs that can be deployed on a machine*.

Recently, NFVNice [21], has proposed a poll-mode VNF scheduling mechanism that assigns CPU shares to VNFs in proportion to their packet processing cost. NFVNice also proposes to re-adjust CPU shares allocated 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

- *Anthony, Shihabur Rahman Chowdhury, Tim Bai, and Raouf Boutaba are with the David R. Cheriton School of Computer Science, University of Waterloo, ON N2L 3G1, Canada.*
  Email: {a3anthon, sr2chowdhury, tim.bai, rboutaba}@uwaterloo.ca
- *Jérôme François is with the RESIST team, INRIA - Nancy Grand Est, 57600, France. Email: jerome.francois@inria.fr*

yielding CPU at some carefully placed scheduling checkpoints in the code [22]. However, these solutions are *intrusive*, *i.e.*, require modifications to the VNFs to make them compatible with the scheduler, thus *highly restricting the type of VNFs that can work properly with the scheduler*.

Motivated by the gaps in the state-of-the-art, we set out to answer the following question: *how can we devise a scheduling mechanism for poll-mode VNFs that does not require VNF modifications and maximizes the number of VNFs sharing a CPU core while maintaining a high packet processing throughput?* We answer this question by presenting the design and implementation of UN*i*S: a **U**ser-space **N**on-**i**ntrusive Workflow-aware VNF **S**cheduler that is:

- **User-space**: works in the user-space and does not require any kernel modification;
- **Non-intrusive**: takes a black-box scheduling approach and does not require VNFs to be built with any UN*i*S specific library or to implement any specific scheduling logic; and
- **Workflow-aware**: takes SFC processing order into consideration while scheduling VNFs.

In this paper, we present the design and implementation of UN*i*S. We have implemented UN*i*S in C++ to work alongside a DPDK-based VNF platform. We have performed extensive testbed experiments using both synthetic and real network traces for evaluating UN*i*S and compare with an intrusive co-operative VNF scheduler similar to [22]. Our key finding is that UN*i*S, despite its black box scheduling approach, is able to achieve a throughput within 90% (synthetic traffic) and 98% (real traffic) of that achieved using the co-operative intrusive scheduler.

This work builds on our initial work in [23] and extends it in several aspects. First, we describe our design goals and their rationale in a greater detail, and elaborate on the challenges associated with achieving these goals. In order to better describe different scheduling scenarios, we augment the description of scheduling algorithm with illustrative examples. Then, we elaborate on UN*i*S's implementation, discussing the possible alternatives for implementing different UN*i*S system components and our rationale for choosing a specific one when applicable. We also extend our evaluation focusing on characterizing the cache access behavior and context switch overhead of VNFs scheduled by UN*i*S. We provide a better explanation of the performance gap between UN*i*S and the compared intrusive scheduling approach with the aid of these new results. We conduct another set of new experiments demonstrating the benefits of the optimization that we have introduced as part of UN*i*S. Finally, we extend our discussion of the related works and contrast UN*i*S with the state-of-the-art.

The rest of the paper is organized as follows. We begin with a brief overview of DPDK based packet processing and process scheduling in Linux kernel in Section 2. Then, Section 3 presents a motivating experiment demonstrating the ineffectiveness of existing OS schedulers for scheduling VNFs in an SFC. We present the design and implementation of UN*i*S in Section 4 followed by the experimental results in Section 5. Section 6 contrasts UN*i*S with the state-of-the-art approaches. Finally, we conclude with some future research directions in Section 7.

2 Background

2.1 Packet Processing with DPDK

Intel Data Path Development Kit (DPDK) [18] is a set of libraries to facilitate fast packet processing in the user-space. DPDK contains libraries for kernel-bypass packet I/O, lockless multi-producer multi-consumer circular queues (DPDK `rte_ring` library), and memory management (DPDK `rte_mempool` library) among others. The ring library can be used to create shared memory based abstractions between packet processors for zero-copy packet exchange. DPDK also ships with a set of NIC specific poll-mode drivers (PMDs) for packet I/O to/from the NIC.

VNFs built using DPDK run in the user-space, bypasses the kernel and continuously poll the NIC for incoming packets. Poll-mode I/O in DPDK is a departure from the traditional interrupt driven I/O, where I/O operations engage the CPU only when packets become available at the NIC. When an I/O interrupt occurs, the CPU switches context from that of the currently running process to that of the interrupt handler in the kernel, performs packet I/O and copies the packets from the kernel-space to the user-space. In contrast, poll-mode I/O always engages a CPU and performs zero copy I/O from the user-space whenever packets become available. In this way poll-mode I/O obliterates the need for context switching, executing interrupt handlers, and copying packets from the kernel- to the user-space. By eliminating these overhead among others introduced by interrupts [24], [25], polling incurs very low CPU overhead and low latency, significantly increasing packet processing throughput compared to interrupt driven I/O [9]. However, the major drawback of poll-mode I/O is that packet processing applications always keep the CPU busy for polling the NIC, even when there are no incoming packets.

2.2 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 run time 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. The time allocated to a process depends on some configurable parameters [26], namely: (i) `sched_min_granularity_ns`: minimum duration 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`$\times$`sched_min_granularity_ns`). Each competing process then gets (`sched_period / n_tasks`) amount of CPU time within a scheduling period.

CFS performs frequent 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,

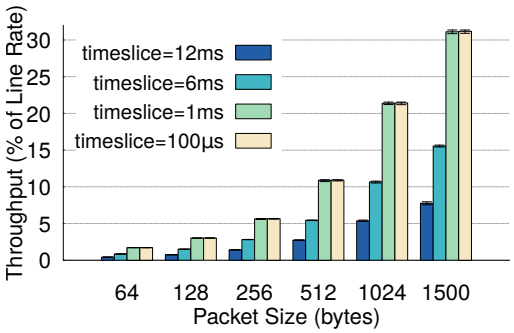
rather than ensuring fairness among competing processes. RT scheduler has two scheduling policies resulting in a process being preempted only after it has finished (first-in-first-out (FIFO) policy) or after its allocated time slice has expired (round-robin policy). Note that in the case of VNFs, processes 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, FIFO policy as currently implemented in the kernel will keep running a VNF instance on a shared CPU core indefinitely and starve the other VNFs sharing the same CPU core. Therefore, RT scheduler with FIFO policy is not a viable option to use for scheduling VNFs on a shared CPU core. RT scheduler with a round-robin policy has a number of tunable kernel parameters [26]. We are interested in the `sched_rr_timeslice_ms` parameter, which determines the length of `time_slice` a process is allowed to run before the next one is scheduled in a round-robin fashion.

3 Motivation

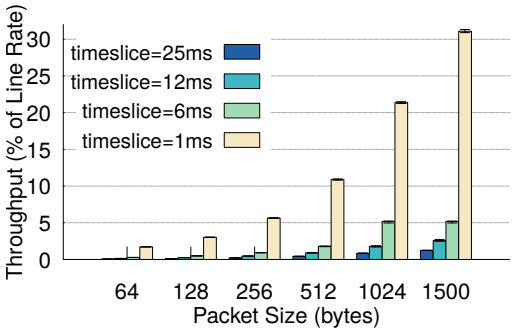
We perform an experimental study to demonstrate that existing OS schedulers fall short of efficiently scheduling VNFs in an SFC competing for the same CPU core. Note that this experimental study complements the motivational experiment presented in [21] by considering a VNF chain as opposed to individual VNFs sharing a core. For this study, we developed a lightweight VNF on a DPDK-based NFV platform [17]. This VNF 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 (similar to [27]). 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 3.3 Ghz Intel Xeon E3-1230v3 CPU and a 10 Gbps NIC, connected directly with a traffic generator without any interfering switch. We generate traffic with varying packet sizes using pktgen-dpdk [28]. We use both CFS and RT scheduler for this study. We express the throughput of the SFC as the percentage of throughput of obtained from a baseline scenario. Our baseline scenario uses the same SFC with each VNF pinned to a different CPU (which was found to be 10Gbps line rate for the smallest packet size (*i.e.*, 64B packets)). Since each VNF gets its own dedicated CPU in the baseline scenario, therefore, this scenario indicates the maximum achievable throughput of the SFC.

The results of this experiment are presented in Fig. 1. The first bar for each packet size represents the result obtained with the default scheduler parameters. For both CFS and RT scheduler, throughput is significantly low. For 64B packets, the throughput is ≈1% of line rate and with MTU size packets, it does not exceed ≈30% of line rate. Note that bigger packets result in more bytes transferred per second (bps), hence, the increase in observed throughput. Such poor performance can be explained as follows. In the case of



(a) CFS



(b) RT Scheduler

Fig. 1. Packet Processing Performance of SFCs using Linux Schedulers

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 from that point. One solution to this problem is to increase the size of shared memory backing the interface between VNFs. However, to avoid packet drop during a VNF's allocated `time_slice`, several megabytes of memory are required for each interface. This is indeed one possible solution but will severely increase packet processing latency.

We also tune the `time_slice` allocated to VNFs by changing CFS parameters described in Section 2.2. However, CFS does not support allocating less than 100μs *time_slice* to a process. As we can see from Fig. 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 it is limited to sub-millisecond *time_slice*. Moreover, it is important to note that neither CFS nor RT scheduler are able to enforce the VNF execution order according to the SFC.

This experimental study motivates a further examination of scheduling in NFV context. The state-of-the-art in NFV scheduling proposes to build VNFs by linking scheduler provided libraries [21] or writing the VNF code in a way that allows VNFs to cooperate together [22]. The main reason for being intrusive is to provide the scheduler with a better insight into and more control over VNFs. However, at the same time being intrusive limits the generality of the solution. To alleviate this limitation, we address VNF scheduling

using a *non-intrusive black box approach* and design UN*i*S to be also *workflow-aware*, *i.e.*, preserve VNF execution order in an SFC for better CPU usage.

4 UN*i*S: Design and Implementation

In this section, we present the design of UN*i*S and describe how the system components are implemented. We begin by outlining the assumptions we make about the underlying NFV platform (Section 4.1) followed by describing our design goals and the associated challenges (Section 4.2). Then, we give an overview of the system architecture and individual components (Section 4.3), present our scheduling algorithm (Section 4.4) and describe the implementation of each UN*i*S's system component (Section 4.5).

4.1 Assumptions

UN*i*S 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 UN*i*S to operate alongside a DPDK based VNF platform such as the one shown in Fig. 2 (*e.g.*, BESS [10], OpenNetVM [13], μNF [17], *etc.*). 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 and can be implemented in software [10] or using specific NIC features [29]. However, UN*i*S does not depend on any implementation specific features of the abstract components, hence, is not tied to any particular implementation.

As a first step to achieve non-intrusive workflow-aware VNF scheduling, we consider linear SFCs since they cover a wide-range of use cases [30], [31]. For general SFC forwarding graphs [32], we need to consider factors such as the traffic load on different paths of the forwarding graph besides considering the graph's structural properties. This merits a separate investigation that we leave for a future extension. Also, we assume VNF to CPU mapping for an SFC is externally computed using one of many available algorithms [33]. Furthermore, UN*i*S assumes that a VNF is not shared by multiple SFCs, which covers a wide range of use cases according to the NFV research literature [3].

UN*i*S is intended to be used as a local scheduler for VNFs deployed on a server inside a data center and does not consider a data center wide scenario. Indeed, a data center wide view will result in better scheduling decisions. However, being first to address VNF scheduling in a non-intrusive way, UN*i*S currently focuses on local scheduling (*i.e.*, an alternative to existing OS kernel and intrusive schedulers). The data center wide case is a challenging problem of its own and merits independent investigation.

4.2 Design Goals and Challenges

Our objective is to design and implement a scheduler for poll-mode VNFs, which maximizes the number of unmodified VNFs that can share a CPU while achieving the same level of performance as any intrusive VNF scheduler. A major challenge in achieving this objective is to devise a scheduling mechanism that minimize the wastage of CPU cycles. CPU cycles can be wasted in two possible ways: (i) a poll-mode VNF is scheduled on a CPU and there is no meaningful work to be done, hence, the VNF wastes CPU cycles by just polling its ingress interface; and (ii) a poll-mode VNF consumes CPU cycles processing packets, but there is not enough room in its egress interface, causing packet drops and wasting the CPU cycles spent in processing those dropped packets. Apart from addressing the aforementioned challenge, we have additional challenges stemming from the following design goals for UN*i*S:

Non-intrusive: Prior work on VNF scheduling took an intrusive approach requiring the VNFs to be built with scheduler specific libraries. Clearly, this approach has benefits such as the scheduler is able to monitor more parameters with lesser monitoring overhead (*e.g.*, event-based notification of packet drops, processing latency, *etc.*) provided by the VNF to the scheduler. However, such requirement limits the generality of the solution and highly restricts the types of VNFs that can work properly with the scheduler. We believe that a better and more generic approach is to be *non-intrusive*, which does not require VNFs to be built with scheduler specific libraries or have carefully placed scheduling checkpoints inside their code. However, the major challenge for designing and implementing a non-intrusive scheduler is that the scheduler has limited information about the events taking place inside the VNFs, such as packet drops, actual packet processing cost, *etc*.

High usability: Our design goal is not only to eliminate the requirement of modifying VNFs but also to require minimal changes to the OS on which these VNFs are running. One option for implementing UN*i*S is to make it part of the OS kernel by augmenting existing OS schedulers or make UN*i*S available as another stand-alone scheduler in the OS kernel. However, it has several negative consequences including the slow pace of adaptation for a new scheduler to existing OSs and the need for users to change their OS kernel. Therefore, we choose to implement UN*i*S in the user-space, increasing its usability. Indeed, UN*i*S requires OS support for certain operations, *e.g.*, bring a process into running state from waiting state. Such OS specific operations can be abstracted as method invocations to a driver with different implementations based on the available system calls for an OS. Moreover, being in user-space also allows us to isolate some CPU cores and let UN*i*S take control over them, leaving the rest of the cores to be used by OS schedulers. However, the challenge in implementing UN*i*S as a user-space scheduler is to meet the micro-second scale decision making requirement for VNF scheduling (as demonstrated in Section 3) in spite of the additional overhead in accessing low-level OS/hardware components via system-calls.

SFC-aware: It is crucial to ensure that when a VNF is scheduled on a CPU core, it has meaningful work to do and cause minimal waste of CPU cycles (*i.e.*, have enough pack-

ets to process and have enough room in the outgoing interface to push the processed packets). Not scheduling VNFs in the order they appear in an SFC can result in frequently scheduling them at times when there is no meaningful work to perform. Consequently, the chances of wasting CPU cycles increase. Therefore, one of our design goals is to be *SFC-aware*, *i.e.*, schedule VNFs with meaningful work to be done according to their order in the SFC. One approach to make scheduling SFC-aware is to adapt existing OS schedulers to enforce VNF execution order according to an SFC, which is a non-trivial task. The difficulty arises from the fact that existing OS schedulers (*e.g.*, in Linux) do not provide any interface for enforcing a specific execution order of VNFs sharing a CPU core. Therefore, VNF scheduling according to the workflow of an SFC poses a major challenge.

4.3 System Architecture

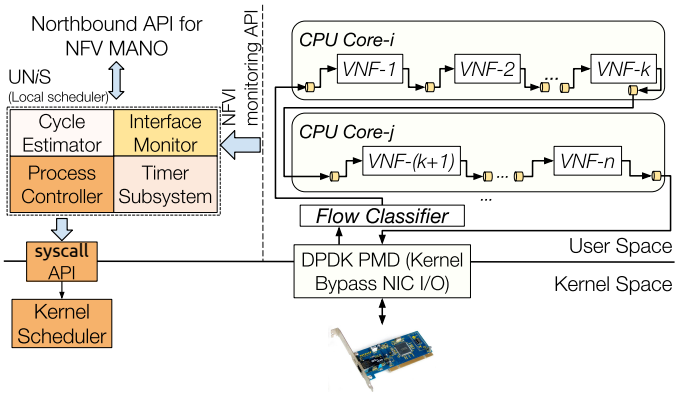


Fig. 2. System Architecture

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

The system architecture of UN*i*S is presented in Fig. 2. UN*i*S exposes a north-bound interface for the NFV Management and Orchestration (MANO) systems (*e.g.*, OSM [34]) so that UN*i*S can be fed with SFC deployment information such as VNF to CPU core assignment, configuration of interfaces that connect the VNFs, *etc*. This information is typical to most NFV MANO systems, hence, do not restrict UN*i*S's generality. UN*i*S leverages the monitoring APIs exposed by existing NFVI software (*e.g.*, OPNFV [35]) to monitor the interfaces connecting VNFs. This follows the ETSI NFV reference architecture [1]. Finally, UN*i*S uses OS provided system call APIs to interact with scheduling and process control subsystem in the kernel for controlling VNF execution states (*e.g.*, change from running to waiting state). Apart from the different APIs for interaction, UN*i*S has four key components as follows.

4.3.1 Cycle Estimator

The cycle estimator is responsible for profiling the VNFs and estimate their processing cost in terms of packet processing latency. Packet processing cost of a VNF depends on a number of factors such as packet size, VNF configuration, packet content, *etc*. [36], [37], [38]. An ideal cycle estimator should be able to take all such factors into account and provide an accurate estimate. Estimated cost of a VNF is then used as an input to the scheduling algorithm for determining the `time_slice` allocated to that VNF.

4.3.2 Interface Monitor

UN*i*S considers the VNFs as black box and relies on externally monitoring the interfaces connecting the VNFs to track a VNF's packet processing progress. The interface monitor assumes that the underlying NFVI exports the following statistics: (i) number of outstanding packets in an interface connecting two VNFs; (ii) maximum number of packets an interface can hold. This information is generic and are commonly exported by existing Linux system tools. Note that with the availability of a richer set of statistics from the interface monitor, *e.g.*, packet drop rate, incoming rate, *etc*., UN*i*S can improve its scheduling decisions.

4.3.3 Timer Subsystem

Besides continuously monitoring the interfaces at a regular interval, UN*i*S also requires time accounting mechanism to decide if a VNF has exhaused its allocated `time_slice`. The timer subsystem maintains a high precision timer in the user-space used for triggering events such as interface monitoring, VNF preemption, *etc*.

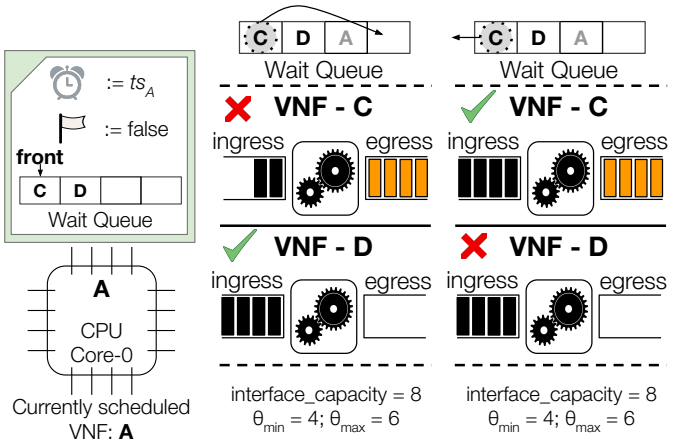
4.3.4 Process Controller

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

The cycle estimator and the interface monitor together help UN*i*S to achieve the first design goal of being non-intrusive. As mentioned earlier, UN*i*S is designed to be a user-space scheduler, in this way achieving the second design goal of being highly usable. The timer subsystem aids in achieving the same goal by enabling UN*i*S to make scheduling decisions in micro-second timescale. Finally, the UN*i*S leverages the process controller component to enforce a desired VNF execution order during scheduling, achieving the third design goal of being SFC-aware.

4.4 Scheduling Algorithm

At the core of UN*i*S, a scheduling algorithm makes scheduling decisions for each CPU core. The scheduling algorithm leverages the components of UN*i*S to monitor the system,



(a) State of data structures after initialization

(b) VNF at the queue front satisfies the selection criteria

(c) VNF at the queue front does not satisfy the selection criteria

Fig. 3. UN*i*S scheduling: an illustrative example

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 [39], [40], [41], [42], [43], [44] focusing on devising an offline execution schedule. However, they are not suitable for taking online scheduling decisions at micro-second time scale, which is a key requirement in UN*i*S. Therefore, we develop a lightweight yet effective scheduling algorithm for UN*i*S based on estimated `time_slice` allocation and occupancy of the interfaces connecting VNFs in an SFC. Before describing the algorithm in detail, we first formally define the VNF scheduling problem (Section 4.4.1). Then, we present the algorithm (Section 4.4.2) followed by its running time analysis (Section 4.4.3).

4.4.1 Problem Statement

Let, $\mathcal{C}$ represent the set of CPU cores on a machine. Each CPU core $i \in \mathcal{C}$ has a wait queue $\mathcal{WQ}_i$ for holding yet to be scheduled VNFs. $\mathcal{F}$ and $\mathcal{S}$ represent the set of VNFs and SFCs, respectively. As mentioned earlier, we consider each SFC, $s \in \mathcal{S}$, to be an ordered and linear sequence of VNFs from $\mathcal{F}$. An assignment function $A : \mathcal{F} \rightarrow \mathcal{C}$ maps VNFs from the SFCs to CPU cores on one or more machines. We assume that the wait queues are initialized with the VNFs mapped to the corresponding CPU cores.

The problem takes as input the set of CPU cores C, the initialized wait queues, $\forall i \in \mathcal{C} : \mathcal{WQ}_i$, and the interval $\mathcal{T}$. The VNF scheduling problem seeks to periodically decide (*i.e.*, every $\mathcal{T}$ time units) which VNF from each wait queue should be scheduled on the corresponding CPU core for execution and for how long. The objective is to maximize the throughput of the SFCs by minimizing wastage of CPU cycles due to dropped packets and idle VNFs.

4.4.2 Algorithm Description

When an external orchestrator invokes UN*i*S's northbound API with VNF to CPU mapping for an SFC request, UN*i*S takes the VNFs in the order they appear in the SFC and places them in their corresponding CPU's wait queues. We initialize the CPU wait queue in such order for enforcing SFC-awareness, which is one of our design goals as described in Section 4.2. In other words, when VNF f_a appears before VNF f_b in an SFC, then placing f_a before f_b in the same wait queue ensures that SFC execution order is maintained while scheduling VNFs from that wait queue. Note that we do not enforce any ordering between VNFs belonging to different SFCs since that is not required for enforcing SFC-awareness. It is possible that because of resource constraints and the placement algorithm used, the external orchestrator maps multiple non-consecutive VNFs from the SFC on the same CPU core. For instance, for an SFC **A** $\rightarrow$ **B** $\rightarrow$ **C** $\rightarrow$ **D**, the orchestrator can map non-consecutive VNFs **A**, **C**, **D** on CPU core 0. In such cases, UN*i*S will initialize the corresponding CPU's wait queue following the relative order of VNFs in the SFC, *e.g.*, in the order **A**, **C**, **D** for CPU core 0. Note that the initialization step is performed before UN*i*S enters the main scheduling loop presented in Algorithm 1. In the following, we will use this aforementioned SFC and the mapping on CPU core 0 as a running example.

Algorithm 1: UN*i*S Scheduling Loop

Input: $\mathcal{C}$ = 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(T)
       /* The system is initialized by running the
          first VNF in every CPU core's wait queue
          and creating corresponding per CPU core
          timers.                                   */
3      while true do
           /* Take scheduling decision every T μs   */
4          if timer_subsystem.monitoring_timer.is_expired()
            == false then continue
           /* Iterate over each CPU core and check if
              a new VNF can be scheduled            */
5          foreach core ∈ C do
6              f ← core.cur_vnf
7              if core.timer.is_expired() or
                monitor.num_pkts(f.ingress) ≤ θ_min or
                monitor.num_pkts(f.egress) ≥ θ_max then
                   /* Iterate over the wait queue (WQ)
                      and find a VNF that has
                      meaningful work to do         */
8                  core.WQ.push(f)
9                  N ← core.WQ.pop()
10                 while (f ≠ N) and
                    (monitor.num_pkts(N.ingress) ≤ θ_min or
                    monitor.num_pkts(N.egress) ≥ θ_max) do
11                     core.WQ.push(N)
12                     N ← core.WQ.pop()
               /* Allocate time_slice for the
                  candidate VNF.                    */
13             time_slice ← cost_estimator.get_cost(N) * γ
                * monitor.pkt_cap(N.egress)
14             if f ≠ N then
15                 process_controller.deactivate(f)
16                 process_controller.activate(N)
17                 core.cur_vnf ← N
18             core.timer.reset(time_slice)
19     timer_subsystem.monitoring_timer.reset(T)
```

The pseudo-code of UN*i*S's main scheduling loop

is presented in Algorithm 1. Before entering the main loop (line 3), it deploys the first VNF in each CPU's wait queue and creates corresponding per core timer by leveraging the `timer_subsystem`. The `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 required by v (profiled by UN*i*S's cycle estimator component), and *interface_capacity(.)* gives us an interface's capacity to hold 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. The parameter $\gamma \in [0, 1]$ is used for leaving some head-room in the interface to account for deviation of actual packet processing cost from the estimation. Once the initial VNFs are deployed, UN*i*S starts monitoring the system and takes scheduling decision every $\mathcal{T}\mu$s. Fig. 3(a) gives an example of the status of the timer and per CPU core wait queue after the first VNF **A** is scheduled on CPU core 0. In this case, the timer will expire after time ts_A, setting the expired flag to `true`. Until the timer expires, VNFs **C** and **D** will remain in CPU core 0's wait queue.

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 (*e.g.*, a firewall VNF dropping packets will change the incoming rate of subsequent VNFs in that SFC). Therefore, there can be situations where a VNF does not have sufficient packets to process (*i.e.*, ingress interface has less than θ_{min} packets), or the outgoing interface is close to becoming full (*i.e.*, egress interface has more than θ_{max} packets outstanding), even if the `time_slice` has not expired. We account for these conditions when determining if the currently scheduled VNF should be preempted or not (line 7).

When Algorithm 1 decides to preempt the currently scheduled VNFs on a CPU core, *i.e.*, when line 7 evaluates to true, it iterates over that CPU core's wait queue and finds 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 criteria avoids wasted CPU cycles and unnecessary context switches by ensuring that a scheduled VNF has meaningful work to do. We refer to this added optimization as the *interface occupancy based optimization* and experimentally show its benefits in Section 5.7. This optimization aids in achieving our design goal of being SFC-aware. Once Algorithm 1 finds a candidate VNF, it leverages the process controller to preempt the current VNF and schedule the next one.

Fig. 3(b) gives an example when VNF **C** at the front of the wait queue satisfies the selection criteria for getting scheduled next. In this case, the currently scheduled VNF **A** is pushed to the back of the wait queue, and VNF **C** is removed from the wait queue and scheduled on CPU core 0. Fig. 3(c) gives another example where VNF **C** at the front of the wait queue does not satisfy the selection criteria. In this case, we first push **A** to the back of the wait queue. Then we keep removing VNFs from the front of the queue and test if it satisfies the selection criteria. If the selection criteria is not satisfied then the removed VNF is pushed to the back of the wait queue. We repeat this process until a suitable VNF is found. In this example, VNF **D** is the first VNF that satisfies the selection criteria, hence, is scheduled on CPU core 0 by setting the appropriate timer and flag.

4.4.3 Algorithm Running Time

During each interval, Algorithm 1 iterates over all the VNFs in the wait queue of each CPU core to identify VNFs that can be scheduled. Therefore, the algorithm requires $O(|\mathcal{WQ}_i|)$ iterations for each CPU core $i \in \mathcal{C}$ or $O(\sum_{i=1}^{|C|} |\mathcal{WQ}_i|)$ total iterations. Since $\sum_{i=1}^{|C|} |\mathcal{WQ}_i| \leq \sum_{\forall s \in \mathcal{S}} |s|$, the running time of Algorithm 1 becomes $O(\sum_{\forall s \in \mathcal{S}} |s|)$.

4.5 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. 2. The reference NFV platform uses DPDK PMDs for packet I/O, `rte_ring` and `hugetlbfs` [45] to create shared memory between VNFs facilitating zero-copy packet exchange. In our implementation, 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 interface 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} to (`batch_size` - 8), and θ_{max} to $interface_capacity(v.egress) * \gamma$ for a VNF v. Note that we experimentally evaluate the impact of changing γ on SFC throughput in Section 5.7. In the following, we describe the implementation of UN*i*S system components in detail.

4.5.1 Cycle Estimator

We currently implement the Cycle Estimator to statically profile a VNF by pushing a batch of 64B packets into the ingress interface of a VNF and then polling the egress interface to capture the batch back, and measures the time elapsed in between. During the estimation process, the VNF is given a dedicated CPU core without other VNFs sharing it. This estimated cost is not the exact 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 offset the impact of inaccuracies in the estimated cost we introduce the interface occupancy based optimization in Algorithm 1 to fine tune the effective `time_slice` and leave dynamic adaptation of processing cost as a future work.

4.5.2 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 initialization, the Interface Monitor uses `rte_ring` library to periodically read the ring occupancy. 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.

4.5.3 Timer Subsystem

We leverage DPDK's `rte_timer` library [46] for high precision time keeping in the user-space. Under the hood, this library uses the High Precision Event Timer (HPET), a hardware timer integrated to majority of chipsets. In the absence of HPET, this library uses the CPUs Time Stamp Counter (TSC) registers to provide a reliable time reference. Timers created by `rte_timer` use a callback mechanism to set a shared variable indicating timer expiration. We periodically poll the shared variable to check for timer expiry and trigger the necessary scheduling events. Currently, we poll the timer every 1μs, hence, can trigger events at 1μs granularity. Before settling on `rte_timer`, we also explored integrating *libevent* [47], 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 needed.

4.5.4 Process Controller

A key challenge in implementing process controller in the user-space is to ensure a low overhead in switching processes. We explored a few different approaches for implementing the process controller including linux control group (cgroup), POSIX signals, and using scheduler priority parameter (`sched_priority`) in RT scheduling with round robin policy. In the following, we describe our experience with different implementation approaches and the rationale for resorting to using scheduler priority parameter for changing execution state of VNFs.

Linux cgroup is typically used to limit and isolate resource (*e.g.*, CPU, memory, disk I/O, and network) usage of a group of competing processes. However, we found that assigning relative CPU shares between processes under a control group did not provide a precise control over the execution duration of VNFs, which is fundamental to ensure VNFs do not execute for too long to start dropping packets.

POSIX signal allows a user-space process to send signals to another process or a thread. In our case, we found `SIGSTOP` and `SIGCONT` signals to be of particular interest. `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 experiment with the POSIX signal was successful when scheduling two processes with a relatively large `time_slice`. However, this approach did not work properly with more than two processes and with a smaller `time_slice`. Our conclusion is that POSIX signals can not handle the high frequency (a few μs interval) invocations required for UN*i*S.

The final mechanism we implemented and also our mechanism of choice for UN*i*S is the following. We set the kernel to use RT scheduler with round robin policy. With this setup, RT scheduler schedules the process with the highest priority at any given time and puts the rest in waiting state. When a different process is given the highest priority, RT scheduler swaps the current process with the new highest priority process. This way, we are able to control the execution state of VNFs. Furthermore, this approach was able to sustain the high frequency process switching as required by UN*i*S. Note that VNFs are switched after every `time_slice` or less, which is computed by UN*i*S and much smaller than the one assigned by RT scheduler. Therefore, RT scheduler does not have any side-effect on scheduling decisions taken by UN*i*S.

5 Performance Evaluation

We evaluate the performance of UN*i*S through testbed experiments. In the following, we first describe our experiment setup in Section 5.1. Then, we present our evaluation results on the effectiveness of UN*i*S's scheduling based on the following scenarios: (i) SFC with fixed and uniform cost VNFs (Section 5.2), (ii) SFC with fixed but non-uniform cost VNFs (Section 5.3), and (iii) SFC with variable cost (traffic dependent) VNFs (Section 5.4), and (iv) one or more SFCs deployed across multiple CPU cores (Section 5.5). We also present additional evaluation results that (i) give insight into why UN*i*S achieves lower throughput compared to the intrusive approach in certain cases (Section 5.6) and (ii) demonstrates the benefits of interface occupancy based optimization (Section 5.7). We conclude this section with a discussion on cost vs. benefit of using intrusive and non-intrusive approach (Section 5.8).

5.1 Experiment Setup

5.1.1 Testbed

Our testbed consists of two physical machines with identical configuration connected back to back without any interfering switch. One machine acts as the device under test and hosts the VNFs and UN*i*S, while the other one is used for traffic generation. Each machine is equipped with a DPDK compatible Intel X710-DA 10 Gbps NIC (flow-control disabled to prevent sending pause frames), 3.3 Ghz 4-core Intel Xeon E3-1230v3 CPU (CPU scaling governor set to *performance*), and 16 GB of memory (4GB allocated to hugepages). When running UN*i*S, we isolate all the CPU cores except core 0 from the kernel scheduler and use them for VNF deployment, in this way eliminating any conflict between the kernel scheduler and UN*i*S. We use DPDK version 17.05 on Ubuntu 16.04LTS with kernel version 4.10.0-42-generic. We also disable Address Space Layout Randomization to ensure consistent hugepage mapping across the VNFs.

5.1.2 VNFs and Workload

We use two types of VNF in our experiments: (i) **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) **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 described in Section 3 and add some imitated 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 (64B) packets and measuring the packet processing latency, and use this as their cost during scheduling. We profile the variable costs VNF using varying packet sizes ranging from the smallest size (64B) to MTU size (1500B) and consider the average packet processing latency over all sizes as their cost.

We use pktgen-dpdk [28] and Moongen [48] for throughput and latency measurements, respectively. For throughput measurement, we generate traffic with different packet sizes, *i.e.*, ranging from smallest size (64B) to MTU sized (1500B) packets with pktgen-dpdk. We also use a real data center traffic trace (UNI1 traces [49] from a campus data center study conducted as part of [50], exhibiting a bimodal packet size distribution) to evaluate the effectiveness of UN*i*S under realistic traffic load. During latency measurement, we set the packet size to 128B and packet rate to 80% of the maximum sustainable throughput for that deployment scenario.

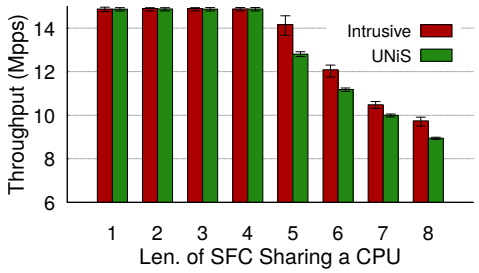
5.1.3 Compared Approaches

We compare UN*i*S with an intrusive co-operative scheduling approach similar to [22]. In the intrusive approach, the VNF is designed to voluntarily yield CPU to other competing VNFs after processing a certain number (k) of batches of packets (we experimentally found 8 to be a good choice for k). Due to the voluntary yields, the `time_slice` allocated to VNFs by RT scheduler does not have any impact on 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 to minimize impact of any such non-determinism. Indeed, NFVNice [21] is a better candidate for comparing with UN*i*S. However, neither the NFVNice scheduler source code nor the specilized libraries that VNFs should be built with for working with NFVNice (*i.e.*, *libnf*) were available at the time of writing this paper. The NFVNice paper also lacked implementation details required to properly implement it from scratch. Therefore, we resorted to comparing UN*i*S with the aforementioned intrusive cooperative scheduler.

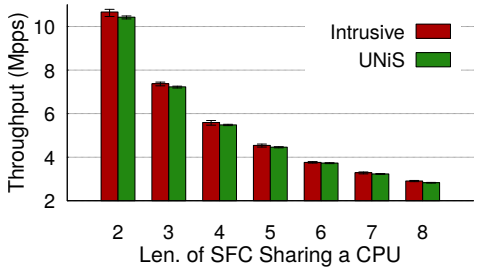
5.1.4 Evaluation Metrics

Throughput and Latency: We measure the packet processing throughput and packet processing latency of VNFs scheduled using both UN*i*S 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 average with 5th and 95th percentile values in μs.

VNF density: VNF density of a scheduling approach is measured by fixing a target throughput and determining the maximum length of an SFC (*i.e.*, number of VNFs) that can be deployed on a single CPU to sustain that throughput. This metric demonstrates a scheduling approach's ability to pack as many VNFs to a CPU core while maintaining a target throughput.



(a) Throughput with Light VNFs



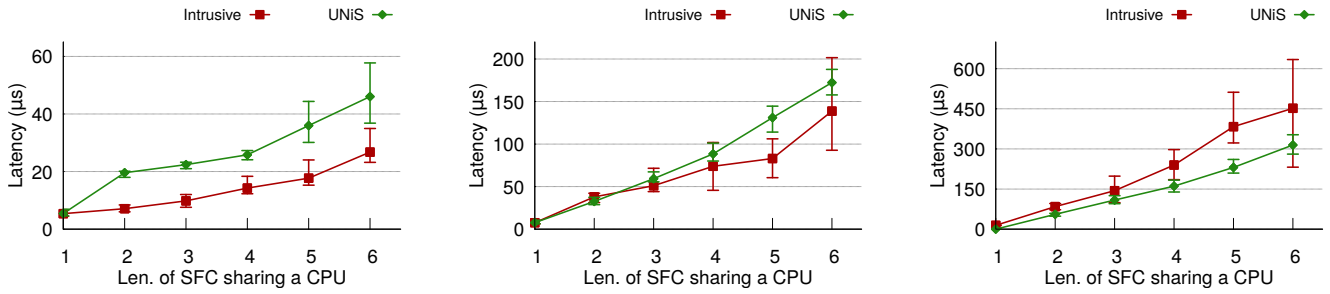
(b) Throughput with Medium VNFs

Fig. 4. Throughput of SFC with fixed and uniform cost VNFs

5.2 SFC with fixed and uniform cost VNFs

Our first set of experiments evaluate the deviation of the non-intrusive scheduling approach from the intrusive approach in terms of throughput. We deploy SFCs of different lengths composed from identical VNFs with fixed packet processing cost (all light VNFs) on a single CPU core and present throughput results for the smallest (*i.e.*, 64B) packet size in Fig. 4(a). Up to an SFC of length 4, both the intrusive approach and UN*i*S are able to sustain line rate throughput. From length 5 and beyond, 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 any inaccuracy in `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(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 for high throughput and not for low latency. However, we still perform a set of experiments to measure the extent of latency incurred by the packets and present the results in Fig. 5. For SFCs with light and medium VNFs we observe an increased packet processing latency when VNFs are scheduled by UN*i*S compared to using the intrusive approach. Because of yielding the CPU after processing small number of batches, the intrusive approach avoids queue buildups, hence, the lower latency compared



(a) Latency with Light VNFs (b) Latency with Medium VNFs (c) Latency with Heavy VNFs

Fig. 5. Latency of SFC with fixed and uniform cost VNFs

to UN*i*S. In case of light VNFs, packets experienced 14μs additional latency on average compared to the intrusive approach as shown in Fig. 5(a). SFC with medium VNFs also exhibit similar behavior, however, exhibiting a smaller latency gap between UN*i*S and the intrusive approach. This is because queue buildup in the interfaces connecting VNFs have higher sensitivity to allocated CPU time for light VNFs compared to medium VNFs. Since light VNFs process packets faster, the interfaces connecting VNFs tend to fill up faster as well. As a result, the currently chosen monitoring interval by the Interface Monitor combined with the discrepancy between estimated and actual packet processing cost can cause light VNFs to populate more than γ fraction of their interface capacity before they are preempted. This increases queuing delay in the interfaces. However, due to the higher packet processing cost, queue buildup is not as sensitive in medium VNFs, hence, the reduced gap between the intrusive approach and UN*i*S.

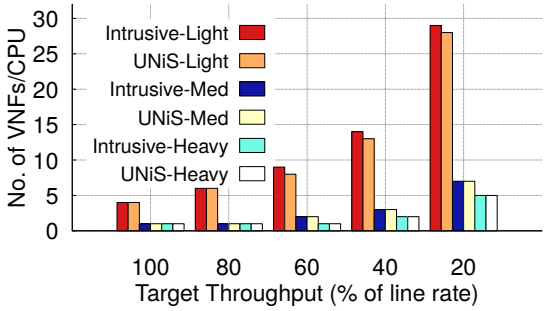


Fig. 6. VNF density on a Single Core with fixed cost VNFs in an SFC

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 because the intrusive approach is seldom schedules VNFs not according to their order in the SFC. Because of out of order execution, packets have to stay longer in the interfaces until the appropriate VNF is scheduled to process 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.

In Fig. 6, we present VNF density with varying target throughput. We conduct the experiment by using all three flavors of VNFs, *i.e.*, light, medium, and heavy. UN*i*S achieves nearly identical VNF density compared to the intrusive approach, deviating less than 10% in few scenarios.

5.3 SFC with fixed but non-uniform cost VNFs

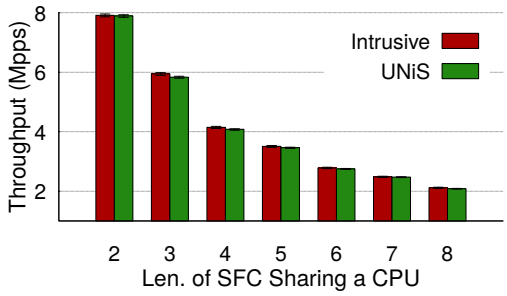


Fig. 7. SFC Composed with fixed but non-uniform cost VNFs

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 at even positions are the heavy ones. The goal of this experiment is to demonstrate the effectiveness of UN*i*S in handling heterogeneity in an SFC. The results of this experiment are presented in Fig. 7. Our observe that UN*i*S is able to sustain a throughput that only deviates less than 2% from that of the intrusive approach for all chain lengths.

5.4 SFC with variable cost (traffic dependent) VNFs

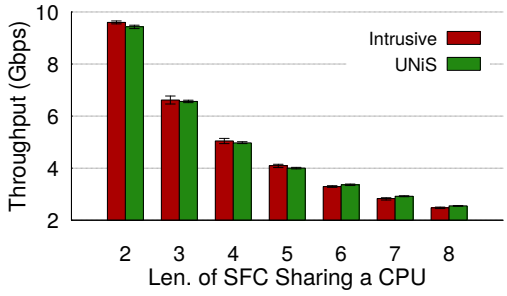


Fig. 8. SFC composed of VNFs with variable processing cost under real traffic load from [50]

TABLE 1
Results for Multiple SFCs across Multiple CPUs

Configuration	No. of VNFs in SFC	No. of VNFs on CPU core-1	No. of VNFs on CPU core-2	Throughput – Intrusive (Mpps)	Throughput – UN*i*S (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

Previous experiments have not considered variable processing cost of a VNF based on traffic characteristics. However, many VNFs that operate on payloads can exhibit different processing costs depending on the packet size (*e.g.*, WAN Optimizers [51], Application Firewalls [52]). To demonstrate the effectiveness of UN*i*S for such cases, we deploy SFCs composed of chains of VNFs whose packet processing cost is a function of packet size, *i.e.*, variable cost VNFs as described in Section 5.1.2. We play a real traffic trace containing packets of different sizes [50] and report the throughput in Fig. 8. For this scenario, UN*i*S performs very close to the intrusive approach, deviating less than 2% for all chain lengths. The packet sizes in the traffic trace follow a bi-modal distribution with most packets closer to 200 and 1400 bytes [50]. Consequently, the variable cost VNFs behave similar to medium and heavy VNFs in most cases. As observed earlier, UN*i*S's performance gap with the intrusive approach is smaller for medium and heavy VNFs compared to that for light VNFs. Hence, the small performance gap for real traffic traces on variable cost VNFs.

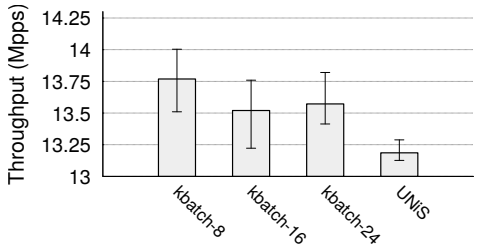
5.5 Multiple SFCs and Multiple CPUs

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 1. In configuration (a), there are multiple SFCs deployed on a single CPU 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. Configuration (b) has two SFCs deployed across two CPU cores and each CPU 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 CPU cores. Finally, configuration (c) has one SFC deployed across multiple CPU cores and here we see that UN*i*S's achieved throughput deviates less than 1.3% from that of the intrusive approach.

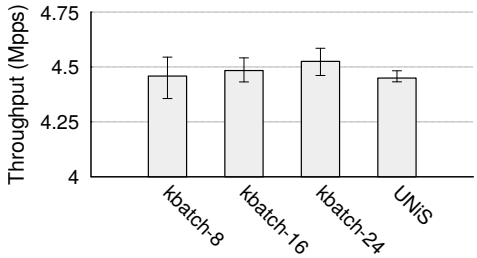
5.6 Investigation into UN*i*S's Throughput Gap

Recall from Section 5.2 that we observed up to 10% throughput gap between UN*i*S and the intrusive approach for an SFC solely composed of light VNFs sharing a CPU core. 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 consumed 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 length 5.

In our compared intrusive approach from Section 5.1.3, the VNF voluntarily yields CPU after processing every k batches of packets, which was set to 8 in 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 them try to fill up 75% capacity of the interfaces connecting adjacent VNFs.



(a) SFC with Light VNFs (length 5)



(b) SFC with Medium VNFs (length 5)

Fig. 9. Throughput for different *kbatch-n* scenarios and UN*i*S

In Fig. 9, we present throughput measurements (using smallest sized packets) for different *kbatch-n* scenarios and for UN*i*S. In Fig. 9(a), we see that both *kbatch-16* and *kbatch-24* scenarios have lower throughput than *kbatch-8*, *i.e.*, our default intrusive scheduling scenario. Whereas in Fig. 9(b),

we observe an opposite trend, *i.e.*, increasing the value of k resulting in a higher throughput. This suggests that there are different factors influencing the performance of SFC composed from light VNFs and from medium VNFs. In both cases UN*i*S has lower throughput than *kbatch-8* scenario and the throughput gap is more evident for SFC composed of light VNFs compared to the one with medium VNFs.

We first look at the number of context switches experienced by each VNF along the chain. We measure context swtiches using the *perf* [53] tool and report results averaged over 25 runs in Fig. 10. As we can see, UN*i*S causes less than half context switches compared to *kbatch-8*, the default intrusive scheduling scenario. This finding is rather counter-intuitive since we expect better throughput with lesser number of context switches, therefore, this result alone is insufficient to explain the throughput gap.

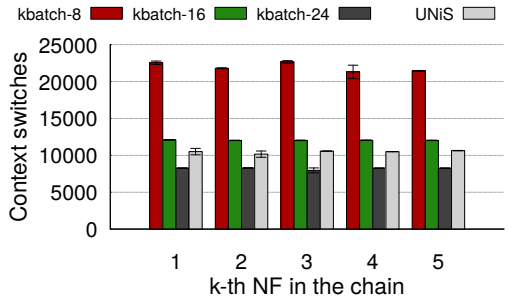


Fig. 10. Context switches in 1 second (uniform and light VNFs)

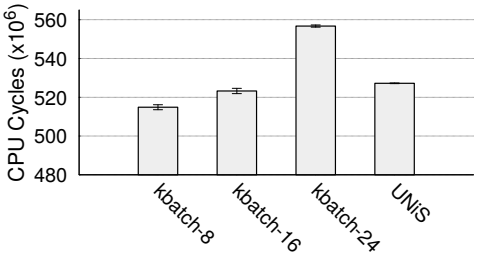


Fig. 11. CPU cycle consumption by the slowest VNF in 1 second (uniform and light VNFs)

To investigate further, we measure the CPU cycles consumed by each of the VNFs within a 1-second time window using perf and identify the slowest VNF, *i.e.*, the VNF getting the least fraction of CPU cyles in the SFC. We report the CPU cycles consumed by the slowest VNF averaged over 25 runs in Fig. 11. We focus on the slowest VNF in the chain since this VNF limits the throughput of the SFC. As we can see from Fig. 11, despite the lesser number of context switches, within a 1-second time window the slowest VNF in UN*i*S, *kbatch-16*, and *kbatch-24* get marginally more CPU cycles than *kbatch-8*, *i.e.*, the default intrusive scheduling scenario. 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 VNFs and present the results in Fig. 12 in terms of percentage of cache misses over total cache references for each of the VNF along the chain (measured using perf and averaged over 25 runs). We observe that UN*i*S 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 the cache. A closer look into the last VNF in the chain reveals that UN*i*S's cache-miss percentage (9.8%) is almost double than that of the default intrusive scheduling scenario (4.9%). This behavior is attributed to UN*i*S processing more number of batches of packets during each scheduling round than the default intrusive scenario. Processing more batches also increases the chances of evicting previously cached packets from the CPU cache hierarchy. This increase in cache misses combined with not so significant savings in CPU cycles from context switches contribute to reducing packet processing throughput of UN*i*S when scheduling light VNFs.

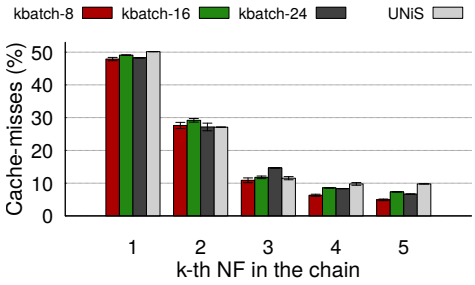
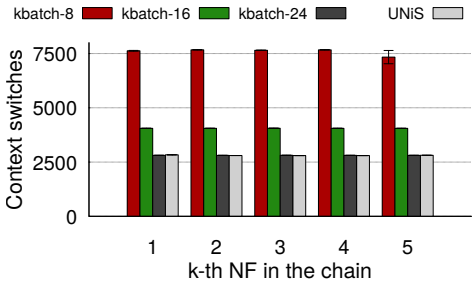
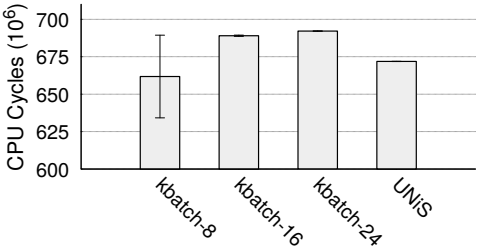


Fig. 12. Cache miss ratio for light VNFs



(a) Context switches in 1 second



(b) CPU cycles consumed by the Slowest NF in 1 second

Fig. 13. Context switches and CPU cycle consumption (uniform and medium VNFs)

In terms of the number of context switches and CPU cyles consumed, we observe similar behavior for an SFC composed of medium VNFs (Fig. 13). However, the processing cost of medium VNF is significantly high to dominate over cache miss penalty, therefore, increased CPU cycles

translate into increased packet processing throughput as can be seen by comparing Fig. 9(b) and Fig. 13(b). In summary, processing more batches of packets during a scheduling interval can reduce context switches at the expense of increasing cache miss percentage. CPU bound VNFs are not significantly affected by an increased cache miss rate. However, cache miss penalty can outweigh the gains from reduced context switches for VNFs that are not CPU bound.

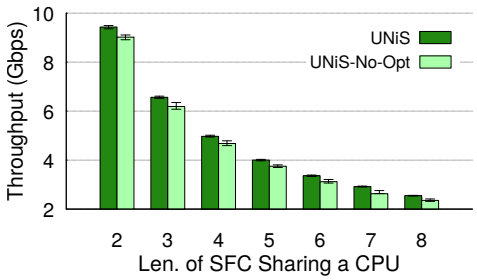
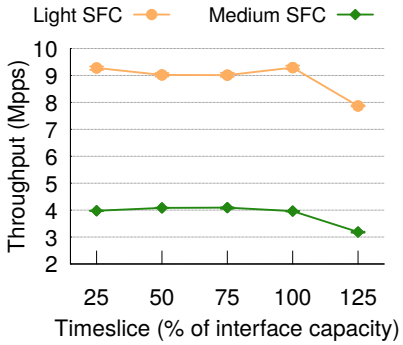
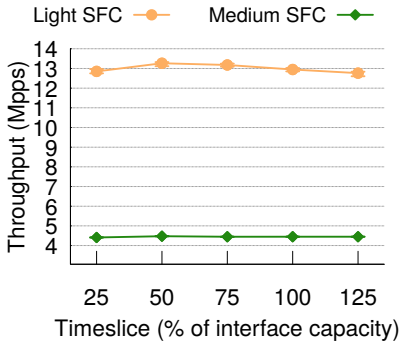


Fig. 14. Impact of interface occupancy based optimization in SFC with variable cost VNFs



(a) UN*i*S without occupancy based optimization



(b) UN*i*S with occupancy based optimization

Fig. 15. Impact of interface occupancy based optimization in SFC with fixed cost VNFs

5.7 Benefits of Occupancy based Optimization

We perform experiments with and without the interface occupancy based optimization (described in Section 4.4) and show its effectiveness in offsetting the impact of any inaccuracy in processing cost estimation of VNFs. We evaluate this impact in the following two scenarios.

The first scenario is similar to the one in Section 5.4. We deploy an SFC consisting of variable cost VNFs and play real traffic traces from [50]. We measure packet processing throughput with and without the interface occupancy based optimization in UN*i*Sand present the results in Fig. 14. 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). As we can see from Fig. 14, 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.

In the second scenario, we deploy an SFC of length 5, all of its constituent VNFs sharing a single CPU core. We vary the allocated `time_slice` to the VNFs from a value large enough to fill only 25% of the interface to a 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 the optimization in Fig. 15(a) shows that allocating a small `time_slice` does not have a significant impact on SFC throughput. However, when the allocated `time_slice` becomes larger than the time it takes to fill the whole interface (125% interface capacity), throughput of both the light and medium SFC experience a sharp drop of 12% and 22%, respectively. In the case of UN*i*S with the interface occupancy based optimization (Fig. 15(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. This demonstrates the effectiveness of the interface occupancy based optimization in offsetting the impact of inaccurate `time_slice` allocation.

5.8 Discussion: Cost vs. Benefit

Results from our testbed experiments suggest that even with a non-intrusive approach, UN*i*S 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 [21] 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 the other ones, thus alleviating the need for continuously monitoring it. Another example is, for a method similar to [21], 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 UN*i*S for high-precision time keeping and monitoring. This is the cost for achieving a generic scheduler capable of working with a wider range of VNFs.

6 Related Works

Scheduling has been extensively studied in various areas of systems and networking such as cluster scheduling [54],

[55], [56], packet scheduling [57], [58], flow scheduling [59], [60] 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 more predictable and usually depend on lesser number of variables (*e.g.*, flow completion time depends on the volume of data 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. We also briefly discuss the research literature on a related area, namely, VNF processing cost estimation.

6.1 Analytical Models for NFV Scheduling

There has been substantial developments in addressing VNF scheduling from a theoretical point of view using different methodologies [39], [40], [41], [42], [43], [61], [62], [63], [64]. Riera *et al.*, presents one of the early integer program formulation for scheduling VNFs on a set of servers [39], which is limited in scalability. Mijumbi *et al.*, presents an optimization model to jointly map and schedule VNFs on physical machines [40]. 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 [42]. The authors proposed to use a mixed integer linear program to optimally solve the problem and then use column generation [42] 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 [41]) and have been solved using methods such as game theory [43], [44]. In contrast to the aforementioned works, which assumed VNF placement to be given, Zheng *et al.*, considered jointly optimizing VNF placement and scheduling network flows to be processed at the VNFs [63]. However, these optimization models are suitable for devising an offline schedule of VNFs for processing network flows. They do not meet the micro-second scale scheduling decision making requirement of VNF scheduling systems such as UN*i*S or NFVNice [21] that replace existing OS schedulers.

In contrast, UN*i*S's focus is to serve as a viable alternative to local OS schedulers for VNF scheduling capable of taking scheduling decision at very short time scale. A related but different problem is addressed in [61]. It proposes an analytical model for scheduling network flows to be processed inside VNFs while ensuring fairness. A theoretical model focusing on processor sharing among VNFs in a single server is presented in [62]. Their objective is to reduce the time an outgoing NIC remains idle. In contrast, our objective is to pack as many VNFs as possible on the CPU cores and achieve comparable throughput to an intrusive scheduling mechanism.

6.2 Systems for NFV Scheduling

Flurries [27] and NFVNice [21] 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 [21] 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. A complimentary work presented in [65] showed that the cost of dropping a packet can be different depending on the stage of SFC processing the packet was dropped. However, NFVNice did not take such differential packet drop cost into account. Both Flurries and NFVNice take an intrusive approach towards scheduling, *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 [22]). 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 UN*i*S to work with a wider range of VNFs.

6.3 VNF processing cost estimation

An orthogonal but related area of research is on estimating the processing cost of VNFs. Indeed, the more accurately we can estimate VNF processing costs, the better schedulers will be able to right size CPU time allocation to competing VNFs. In the state-of-the-art literature, most approaches for cost estimation are offline [37], [38], [66], [67], *i.e.*, the VNF is tested in a controlled environment with different workloads before being deployed in production. VNF processing cost depends on a multitude of factors such as packet size distribution, packet inter-arrival time, packet content, among others [36], [37], [38]. Therefore, the actual VNF processing cost during live operation can deviate from the cost estimated in a controlled environment. Furthermore, estimating processing cost of poll-mode VNFs through existing benchmark techniques is difficult because they always consume 100% CPU. More recently, Gupta *et al.*, have proposed to use hardware counters in modern CPUs to estimate processing cost of poll-mode VNFs [19]. Their preliminary results on a limited set of VNFs are promising and opens a promising research direction. Another orthogonal issue is to identify the appropriate cost metric for VNFs. In the current research literature a wide range of metrics have been used such as CPU overhead [21], cache miss overhead [38], and context switching overhead [68], among others.

In this work, we consider the CPU overhead translated into packet processing latency as our cost metric. Similar to most of the benchmark approaches, we also take an offline approach to cost estimation. However, We introduced the interface occupancy based optimization as a means to mititgate the impact of any inaccuracies in cost estimation.

7 Conclusion and Future Work

In this paper, we presented the design and implementation 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. UN*i*S advances the state-of-the-art in VNF scheduling by being non-intrusive, *i.e.*, not requiring any changes to the VNFs for them to properly work with the scheduler. We compare our implementation of UN*i*S with an intrusive co-operative scheduler on a testbed. Experimental results are promising and demonstrate that even with a black box approach UN*i*S is able to achieve a comparable performance to that of intrusive schedulers that have better insights into the VNFs.

Building on these promising results, our next goal is to extend UN*i*S for considering SFCs with arbitrary graph structure deployed across multiple machines in a data center. Another research direction is to focus on dynamically adjusting CPU time allocation during live VNF operation while considering factors such as packet size distribution and packet inter-arrival time. In this way, we can more accurately allocate CPU time to VNFs and further reduce wasted CPU cycles and fragmentation. In its current form, UN*i*S considers only one quality of service (QoS) parameter, *i.e.*, packet processing throughput. Extending UN*i*S to consider other QoS parameters such as end-to-end latency of SFCs for supporting ultra-low latency network slicing is another promising research direction.

Acknowledgement

This work was supported in part by the INRIA International Chair in Network Softwarization Program and in part by the NSERC CREATE for Network Softwarization program. Anthony and Shihabur were interns with the RESIST team at INRIA Nancy and were supported in part by MITACS Globalink Research Award.

References

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

[2] 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 Rev.*, vol. 42, no. 4, pp. 13–24, 2012.

[3] 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.

[4] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat, "Taking the edge off with espresso: Scale, reliability and programmability for global internet peering," in *Proceedings of ACM SIGCOMM Conference*, 2017, pp. 432–445.

[5] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng, "Engineering egress with edge fabric: Steering oceans of content to the world," in *Proceedings of ACM SIGCOMM Conference*, 2017, pp. 418–431.

[6] L. Peterson, A. Al-Shabibi, T. Anshutz, S. Baker, A. Bavier, S. Das, J. Hart, G. Palukar, and W. Snow, "Central office re-architected as a data center," *IEEE Communications Magazine*, vol. 54, no. 10, pp. 96–101, October 2016.

[7] R. S. Montero, E. Rojas, A. A. Carrillo, and I. M. Llorente, "Extending the cloud to the network edge." *IEEE Computer*, vol. 50, no. 4, pp. 91–95, 2017.

[8] 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*, 2014, pp. 459–473.

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

[10] 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.

[11] 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*, 2016, pp. 203–216.

[12] T. Tsutomu, S. Shuichi, T. Junpei, S. Yuta, S. Hiroshi, and I. Tomohito, "Development of user plane control for vepc," *NEC Technical Journal – Special Issue on Telecom Carrier Solutions for New Value Creation*, vol. 10, pp. 23–27, 2016.

[13] 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*, 2016, pp. 26–31.

[14] 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*, 2017, pp. 348–361.

[15] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr, "Metron: Nfv service chains at the true speed of the underlying hardware," in *Proceedings of USENIX NSDI*, 2018, pp. 171–186.

[16] "Ciena d-nfvi software," Data sheet, Ciena Networks, 2018. [Online]. Available: https://media.ciena.com/documents/Ciena_D-NFVI_Software_DS.pdf

[17] S. R. Chowdhury, Anthony, H. Bian, T. Bai, and R. Boutaba, "μNF: A disaggregated packet processing architecture," in *Proceedings of IEEE NetSoft*, June 2019, pp. 342–350.

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

[19] H. Gupta, A. Sharma, A. Zelezniak, M. Jang, and U. Ramachandran, "A black-box approach for estimating utilization of polled io network functions," in *Proceedings of USENIX HotCloud*, 2019.

[20] P. Quinn and T. Nadeau, "Problem statement for service function chaining," RFC 7498, April 2015. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7498.txt

[21] 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*, 2017, pp. 71–84.

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

[23] 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 *Proceedings of ACM/IEEE/IFIP CNSM*, 2018.

[24] J. C. Mogul and K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997.

[25] C. Dovrolis, B. Thayer, and P. Ramanathan, "Hip: hybrid interrupt-polling for the network interface," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 4, pp. 50–60, 2001.

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

[27] 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*, 2016, pp. 3–17.

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

[29] "Intel flow director." [Online]. Available: https://software.intel.com/en-us/articles/setting-up-intel-ethernet-flow-director

[30] S. Kumar, M. Tufail, S. Majee, C. Captari, and S. Homma, "Service function chaining use cases in data centers," IETF, Internet-Draft draft-ietf-sfc-dc-use-cases-06, August 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-sfc-dc-use-cases-06

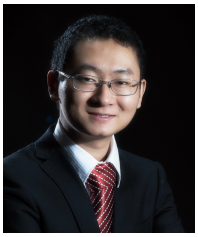
[31] J. Napper, M. Stiemerling, D. Lopez, and J. Uttaro, "Service function chaining use cases in mobile networks," IETF, Internet-Draft draft-ietf-sfc-use-case-mobility-08, May 2018. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-sfc-use-case-mobility-08.txt
[32] "Network Functions Virtualisation (NFV) Release 2; Management and Orchestration; Network Service Templates Specification," Aug 2017, white paper. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/014/02.03.01_60/gs_NFV-IFA014v020301p.pdf
[33] 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.
[34] "Open source MANO (OSM)." [Online]. Available: https://osm.etsi.org/
[35] "Open platform for nfv (OPNFV)." [Online]. Available: https://www.opnfv.org/
[36] S. Lange, A. Nguyen-Ngoc, S. Gebert, T. Zinner, M. Jarschel, A. Köpsel, M. Sune, D. Raumer, S. Gallenmüller *et al.*, "Performance benchmarking of a software-based LTE SGW," in *Proceedings of IEEE/ACM/IFIP CNSM*, 2015, pp. 378–383.
[37] 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*, 2015, pp. 93–99.
[38] 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, 2018, pp. 283–297.
[39] 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*, 2014, pp. 1–5.
[40] 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.
[41] 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.
[42] 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, 2017.
[43] 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.
[44] H. Alameddine, M. H. K. Tushar, and C. Assi, "Scheduling of low latency services in softwarized networks," *IEEE Transactions on Cloud Computing*, 2019.
[45] "hugetlbfs kernel documentation." [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt
[46] "Dpdk rte_timer." [Online]. Available: https://doc.dpdk.org/guides/prog_guide/timer_lib.html
[47] "libevent – an event notification library." [Online]. Available: https://libevent.org/
[48] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of ACM IMC*, 2015, pp. 275–287.
[49] "Data set for imc 2010 data center measurement." [Online]. Available: http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html
[50] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of ACM IMC*, 2010, pp. 267–280.
[51] "Blue coat® systems proxysg™," Tech. Report, Blue Coat® Systems, Tech. Report.
[52] "Barracuda web application firewall." [Online]. Available: https://www.barracuda.com/products/webapplicationfirewall
[53] "Linux perf tool." [Online]. Available: https://perf.wiki.kernel.org/index.php/Tutorial
[54] 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 ACM EuroSys*, 2010, pp. 265–278.
[55] 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.
[56] 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*, 2015.
[57] 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*, 2016, pp. 44–57.
[58] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *Proceedings of USENIX NSDI*, 2016, pp. 501–521.
[59] 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*, 2010.
[60] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 443–454.
[61] X. Li and C. Qian, "Low-complexity multi-resource packet scheduling for network function virtualization," in *Proceedings of IEEE INFOCOM*, 2015, pp. 1400–1408.
[62] 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.
[63] Q. Zhang, Y. Xiao, F. Liu, J. C. Lui, J. Guo, and T. Wang, "Joint optimization of chain placement and request scheduling for network function virtualization," in *Proceedings of IEEE ICDCS*, 2017, pp. 731–741.
[64] 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.
[65] C. Li and L. Cui, "A novel nfv schedule optimization approach with sensitivity to packets dropping positions," in *Proceedings of the TOPIC Workshop (In conjunction with ACM PODC)*, 2018, pp. 23–28.
[66] R. V. Rosa, C. Bertoldo, and C. E. Rothenberg, "Take your vnf to the gym: A testing framework for automated nfv performance benchmarking," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 110–117, 2017.
[67] D. Cotroneo, L. De Simone, and R. Natella, "Nfv-bench: A dependability benchmark for network function virtualization systems," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 934–948, 2017.
[68] M. Savi, M. Tornatore, and G. Verticale, "Impact of processing-resource sharing on the placement of chained virtual network functions," *IEEE Transactions on Cloud Computing*, 2019.



Anthony received MMath degree in Computer Science from the University of Waterloo, Canada and Bachelors degree in Computer Science from NCTU Taiwan. Anthony is a recipient of Mitacs Globalink Research Award in 2018. His research interest includes network softwarization and cloud computing.



Shihabur Rahman Chowdhury (S'13) is a PhD candidate at the David R. Cheriton School of Computer Science, University of Waterloo. He received his B.Sc. degree in computer science and engineering from BUET in 2009. His research interests include virtualization and softwarization of computer networks. He is co-recipient of the Best Paper Award at the IEEE/ACM/IFIP CNSM 2019, IEEE NetSoft 2019, and IEEE/ACM/IFIP CNSM 2017 conference.



Tim Bai received the MMath and BMath degrees from the David R. Cheriton School of Computer Science, University of Waterloo in 2019 and 2017, respectively. His current research interests include machine learning, cybersecurity, and network softwarization.



Raouf Boutaba (F'12) received the M.Sc. and Ph.D. degrees in computer science from the University Pierre & Marie Curie, Paris, in 1990 and 1994, respectively. He is currently a professor of computer science and university research chair at the University of Waterloo. He is the founding editor in chief of the IEEE Transactions on Network and Service Management (2007–2010). He received several best paper awards and recognitions including the Premiers Research Excellence Award, the IEEE ComSoc Hal Sobol, Fred W. Ellersick, Joe LociCero, Dan Stokesbury, Salah Aidarous Awards, and the IEEE Canada McNaughton Gold Medal. He is a fellow of the Royal Society of Canada, the Institute of Electrical and Electronics Engineers (IEEE), the Engineering Institute of Canada, and the Canadian Academy of Engineering. His research interests include resource and service management in networks and distributed systems.



Jérôme François received the M.S. degree in computer science and the Ph.D. degree in robustness and identification of communicating applications from the University of Lorraine, France, in December 2009. Then, he was a Research Associate with the University of Luxembourg in the SEDAN Team of Prof. T. Engel. This was an opportunity to extend his research scope toward other areas like vehicular networks or DNS but mostly with a focus on security. In March 2014, he started as a Research Scientist with Inria in the RESIST Team (formerly, MADYNES) and supports the Team Leader, I. Chrisment, as the Deputy Leader. He is in charge of different international collaborations of the research team with the University of Luxembourg and the University of Waterloo, Canada. His main research areas are focused on the use of data analytics techniques for security as well as the definition of software-based network monitoring probes both at the data and control planes. He was a recipient of the IEEE Young Professional Award in Network and Service Management in 2019. In addition to publications, he started as the Associate Editor-in-Chief of the International Journal of Network Management (Wiley) and as the Co-Chair of Network Management Research Group with Internet Research Task Force in 2019.