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

Anthony*, Shihabur Rahman Chowdhury*, Tim Bai*, Raouf Boutaba*, Jérôme François†
*David R. Cheriton School of Computer Science, University of Waterloo
{a3anthon | sr2chowdhury | tim.bai | rboutaba}@uwaterloo.ca
†INRIA - Nancy Grand Est, France
jerome.francois@inria.fr

***Abstract*—Network Function Virtualization (NFV) has gained a significant research interest in both academia and industry since its inception in the late 2012. One of the key research issues in NFV is the development of systems for building *Virtual Network Functions (VNFs)* capable of meeting the performance requirements of enterprise and telecommunication networks. New packet processing models leveraging kernel bypass I/O and poll-mode processing have gained popularity for building high performance VNFs because of their simple programming model and very low I/O overhead. However, a major drawback of such poll-mode processing is the inefficient use of CPU resources. Existing CPU schedulers are ill-suited for VNFs due to their inability to capture the actual processing cost of a poll-mode VNF, hence, cannot rightsize the CPU allocation. This is further exacerbated by their inability to consider VNF processing order when VNFs are chained to form Service Function Chains (SFCs). The state-of-the-art solutions proposed for VNF scheduling are *intrusive*, *i.e.*, requiring the VNFs to be built with scheduler specific libraries or having carefully selected scheduling checkpoints. This highly restricts the VNFs that can properly work with such schedulers. In this paper, we present UN*i*S: a User-space Non-intrusive work-flow aware VNF Scheduler. Unlike existing approaches, UN*i*S does not require VNF modifications and treats the poll-mode VNFs as a black box, hence, is *non-intrusive*. UN*i*S is also *workflow-aware*, *i.e.*, maintains SFC processing order while scheduling the VNFs. Testbed experiments show that UN*i*S is able to achieve a throughput within 90% (for synthetic traffic load) and 98% (for real data center traffic trace) of the achievable throughput using an intrusive co-operative scheduler.**

I. Introduction

Network operators ubiquitously deploy proprietary, purpose-built, and expensive hardware *middleboxes* (*e.g.*, firewalls, proxies, WAN optimizers, *etc*.) to realize different network services [1]. These middleboxes are a significant source of capital and operational expenditures (CAPEX and OPEX, respectively) because of their proprietary, vertically integrated and inflexible nature. This motivated the *Network Function Virtualization (NFV)* movement, which proposed to decouple Network Functions (NFs) from purpose-built hardware and run them as *Virtual Network Functions (VNFs)* on commodity servers [2]. Through such disaggregation, NFV promises to lower CAPEX by consolidating multiple NFs on the same commodity hardware, and reduce OPEX by enabling on-demand service provisioning.

Moving NFs from specialized hardware to VNFs running on commodity servers comes with several challenges [3]. One key challenge among many others is to achieve the same level of packet processing performance as that of the specialized hardware. A significant body of research has been dedicated to designing and developing VNFs capable of line rate processing at tens of Gbps even for the smallest size packets [4]–[9]. A fundamental building block for these research works is the recently emerged fast packet processing libraries such as netmap [10], Intel Data Path Development Kit (DPDK) [11] and FD.io [12]. These libraries facilitate a rapid development of user-space programs that can read/write packets directly from/to the Network Interface Card (NIC) bypassing the OS kernel, thus incurring very low I/O overhead.

The most popular programming model for developing VNFs leveraging these packet processing libraries is *poll-mode*, *i.e.*, VNFs continuously poll the NIC for incoming packets. Poll-mode VNF development has gained popularity in the last few years because it is simple to implement and incurs lower I/O overhead compared to a traditional interrupt driven model [6]–[9]. However, one caveat of this model is that the VNFs always utilize 100% CPU due to the continuous polling, even when there are no packets to process. This makes it hard to relate CPU utilization of poll-mode VNFs to their packet processing cost. This also renders the traditional kernel schedulers less effective since they heavily rely on CPU usages for taking scheduling decisions. Another drawback of existing kernel schedulers is that there is no interface to specify the desired processing order of VNFs. This is particularly important for scheduling VNFs sharing a CPU, since packets are required to traverse through an ordered sequence of chained VNFs, known as a Service Function Chain (SFC). Due to these reasons, it is very common to see that most research leveraging poll-mode VNFs suggest to pin the VNFs to dedicated CPU cores. This limits the number of VNFs that can be deployed on a machine. In this paper, we address the problem of scheduling poll-mode VNFs on shared CPU cores in a way such that we maximize the number of VNFs on a shared CPU core, while maintaining high packet processing performance.

Recently, NFVNice [13] addressed poll-mode VNF scheduling by proposing a mechanism to assign packet processing cost proportional CPU shares to VNFs. It also proposes to re-adjust assigned CPU shares to VNFs in an SFC when packets start dropping along the chain. However, NFVNice requires VNFs to be built using scheduler provided libraries to be able to monitor packet drops. Another VNF scheduling approach

978-3-903176-14-0

is to build the VNFs that can co-operate with other VNFs sharing a CPU by voluntarily yielding CPU at some carefully placed scheduling checkpoints in the code [14]. However, these solutions are *intrusive*, *i.e.*, require modifications to the VNFs to make them compatible with the scheduler, thus limiting the type of VNFs that can work properly with the scheduler. In this paper, we propose UN*i*S: a **U**ser-space **N**on-**i**ntrusive Workflow-aware VNF **S**cheduler that is: (i) *user-space*: works in the user-space and does not require any kernel modification; (ii) *non-intrusive*: does not require VNFs to be built with any UN*i*S specific library or to implement any specific scheduling logic; and (iii) *workflow-aware*: maintains SFC processing order while scheduling VNFs. We compare UN*i*S with an intrusive co-operative VNF scheduler similar to [14] using both synthetic and real traffic load on a testbed. Our key finding is that UN*i*S, in spite of its black box scheduling approach, is able to achieve a throughput within 90% (synthetic traffic) and 98% (real traffic) of that achieved using the intrusive scheduler.

The rest of the paper is organized as follows. Section II provides a brief overview of DPDK based packet processing and scheduling in Linux kernel. Section III describes a motivating experiment on how the existing OS schedulers fall short for NFV workloads. Section IV details the design and implementation of UN*i*Sand Section V presents our experimental results and analysis. Then, Section VI contrasts UN*i*S with the state-of-the-art approaches. Finally, Section VII concludes this paper by summarizing UN*i*S's main contributions and outlining some future research directions.

II. Background

A. Packet Processing with DPDK

Intel Data Path Development Kit (DPDK) is a set of libraries to facilitate fast packet processing. 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). Packet processors built using DPDK read packets from the NIC by continuously polling the NIC buffers for incoming packets. One advantage of poll-mode I/O over interrupt driven I/O is its lesser overhead that leads to high throughput. However, the major drawback of this model is that packet processors result in 100% CPU utilization for polling the NIC, even if there are no incoming packets.

B. 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. In CFS, the time allocated to a process depends on some configurable kernel parameters [15], namely: (i) `sched_min_granularity_ns`: minimum amount of time a process is allowed to be run on a CPU core before being preempted, (ii) `sched_latency_ns`: minimum period after which CFS takes a scheduling decision. The scheduling period (`sched_period`), *i.e.*, the period after which CFS takes scheduling decisions is set to `sched_latency_ns` if the number of competing processes for a CPU (`n_tasks`) is less than (`sched_latency_ns`/`sched_min_granularity_ns`), otherwise, to (`n_tasks`*`sched_min_granularity_ns`). Each competing process then gets (`sched_period` / `n_tasks`) amount of CPU time within a scheduling period.

CFS performs 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 is not suitable for VNF workload. RT scheduler with a round-robin policy has a number of tunable kernel parameters [15]. 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.

III. 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 [13] by considering a VNF chain as opposed to individual VNFs sharing a core. We developed a lightweight DPDK-based VNF for this study, which performs bare-minimal packet processing (swaps the source and destination MAC addresses) to ensure that its processing overhead is not a performance bottleneck. The VNFs are chained by using a shared-memory based zero-copy packet exchange mechanism built using DPDK `rte_ring` library. 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 3.3Ghz Intel Xeon E3-1230v3 CPU and a 10Gbps NIC, connected directly with a traffic generator. We generate traffic with varying packet sizes using pktgen-dpdk [16]. We use

both CFS and RT scheduler for this study. We express the throughput of the SFC as the percentage of throughput of the same SFC with each VNF pinned to a different CPU (which was found to be 10Gbps line rate for the smallest packet size).

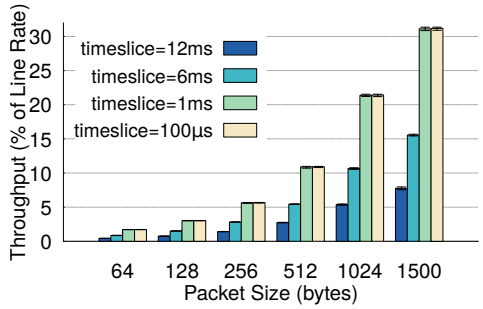
The results of this experiment are presented in Fig. 1. The first bar in each packet size represents the result obtained with the default scheduler parameters. For both CFS and RT scheduler, throughput is significantly low. For 64B packets, the throughput is ≈1% of line rate and with MTU size packets, it does not exceed ≈30% of line rate. Such poor performance can be explained as follows. In the case of CFS, the default configuration results in a `time_slice` of 12ms allocated to each VNF during a scheduling period, which we found to be too long. During this allocated time, a VNF fills up its outgoing interface very quickly. Since the outgoing interface becomes full, all the packets processed afterwards by the VNF are dropped, wasting the work already done 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 latency incurred by the packets.

We also tune the `time_slice` allocated to VNFs by changing CFS parameters described in Section II-B. 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.

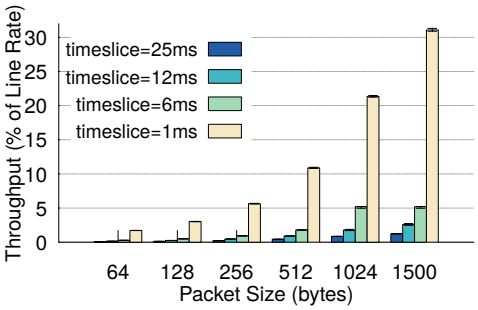
This experimental study motivates a further examination of scheduling in NFV context. The state-of-the-art in NFV scheduling proposes to build VNFs by linking scheduler provided libraries [13] or writing the VNF code in a way that allows VNFs to cooperate together [14]. The main idea here is to provide the scheduler with a better insight into and more control over VNFs. However, at the same time this limits the generality of the solution. To alleviate this limitation, we address VNF scheduling using a non-intrusive black box approach and design the UN*i*S scheduling to be work-flow aware, *i.e.*, preserve VNF execution order in an SFC for better CPU usage.

IV. 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 briefly describing the assumptions we make about the underlying NFV platform (Section IV-A). Then, we give an overview of the system architecture and individual components (Section IV-B), present our scheduling algorithm (Section IV-C) and describe the implementation of each UN*i*S's system component (Section IV-D).



(a) CFS



(b) RT Scheduler

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

A. 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. In this reference platform, the VNFs are chained using an abstract entity called *interface*. An interface is an abstraction over a finite storage with methods for pushing packets to and pulling packets from it in batches. A specific implementation of the interface can be based on virtual Ethernet (veth) pairs, shared-memory, *etc*. Our only assumption about the interface is that it can export the number of outstanding packets/bytes and the actual capacity of the underlying storage. This is a reasonable assumption since many existing system tools export similar information (*e.g.*, veth interfaces shaped by `tc` subsystem export queue occupancy information). Another abstract component, *flow classifier*, redirects incoming packets to the appropriate SFC. Flow classifier can be implemented in many ways such as in software or using specific NIC features [17]. The NFV platform considered here does not assume any specific implementation for the abstract components, hence, does not tie UN*i*S to any specific implementation.

As a first step to achieve non-intrusive workflow-aware VNF scheduling, we consider linear SFCs only and leave the case for general forwarding graphs for future extension. Also, we assume VNF to CPU mapping for an SFC to be externally computed using one of many available algorithms [18].

UN*i*S is intended to be used as a local scheduler for VNFs deployed on a machine and does not consider a cluster-wide scenario. Indeed, a cluster-wide view will result in better scheduling decisions. However, being first to address VNF

scheduling in a non-intrusive way, UN*i*S currently focuses on local scheduling (*i.e.*, an alternative to existing OS kernel and Intrusive schedulers) and leaves the cluster-wide case for future extensions.

B. System Architecture

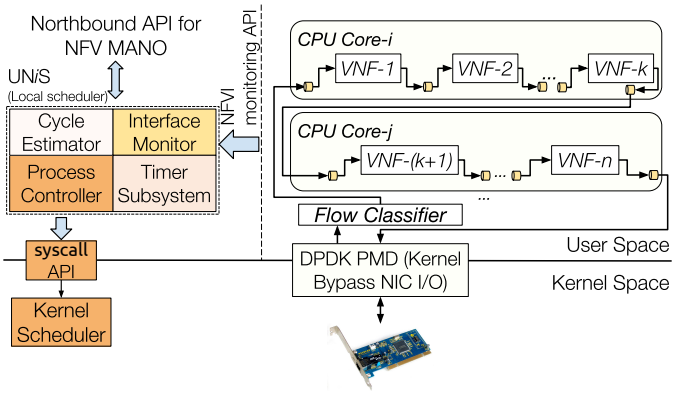


Fig. 2. System Architecture

We design UN*i*S as a user-space VNF scheduler. This design choice has several benefits such as 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 software 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) system 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*. These information are 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 to monitor the interfaces connecting VNFs. This follows the ETSI NFV reference architecture [2]. Finally, UN*i*S uses OS provided system call API to interact with scheduling and process control subsystem in the kernel to control VNF execution states (*e.g.*, change from running to waiting state). Apart from the different APIs for interaction, UN*i*S has the following key components:

1) Cycle Estimator: The cycle estimator is responsible for profiling the VNFs and estimate their processing cost in terms of packet processing latency. Cost of a VNF depends on a number of factors such as packet size, VNF configuration, packet content *etc*. [19]–[21] 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 used as an input to the scheduling algorithm for determining the `time_slice` allocated to that VNF.

2) Interface Monitor: UN*i*S considers the VNFs as black box and relies on externally monitoring the interfaces connecting the VNFs. 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 outstanding packets an interface can hold. These information are generic and are commonly exported by existing Linux system tools.

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) 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). Process controller hides the underlying OS specific details from UN*i*S. Therefore, porting UN*i*S to a different OS only requires updating the process controller with the corresponding system calls for the target system.

C. Scheduling Algorithm

At the core of UN*i*S, a scheduling algorithm makes a scheduling decision for each CPU core. The scheduling algorithm leverages the components of UN*i*S to monitor the system, determines which VNF to run next and the `time_slice` allocation, and acts upon the VNFs to start/stop them. Some research has been dedicated to address VNF scheduling from a theoretical perspective [22]–[27]. However, these proposals are more suitable for devising an offline execution schedule and not for taking online scheduling decisions at the micro-second 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 interface occupancy between the VNFs in an SFC.

UN*i*S maintains a per CPU core wait queue of VNFs. When an external orchestrator invokes UN*i*S's northbound API with VNF to CPU mapping for a new 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.

The pseudo-code of UN*i*S's main scheduling loop is presented in Alg. 1. Before entering the main loop (line 3), it deploys the first VNF in each CPU's wait queue and creates corresponding per core timer by leveraging the `timer_subsystem`. `time_slice` allocated to a VNF v is computed as: $complexity(v) * \gamma * interface\_capacity(v.egress)$, where $complexity(.)$ gives us the estimated per packet processing time (profiled by *Cycle Estimator*) required by v, and *interface_capacity(.)* gives us an interface's capacity to hold outstanding packets. This equation ensures that a VNF is given sufficient time to fill up its egress interface as close as possible to its full capacity, thereby maximizing throughput. $0 \leq \gamma \leq 1$ is a parameter used for leaving some head-room in the interface to account for deviation of actual packet processing cost from the estimation. Then, UN*i*S monitors the system and takes a scheduling decision every $\mathcal{T}\mu$s.

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

D. 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` [28] to create shared memory between VNFs facilitating zero-copy packet exchange. In the following, we describe the implementation of UN*i*S system components.

1) Cycle Estimator: We currently implement the Cycle Estimator to statically profile the VNFs by pushing a batch of 64B packets into the ingress interface of a VNF and quickly polling the egress interface to capture the batch back. This estimated cost is not the ideal representation of actual processing cost since the actual cost depends on many factors such as packet size, VNF configuration, content of the packets, *etc*. We work around this issue by adding the interface occupancy-based optimization in Alg. 1 to fine tune the effective `time_slice` and leave dynamic adaptation of processing cost as a future work.

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.

3) Timer Subsystem: We leverage DPDK's `rte_timer` library for high precision time keeping in the user-space. 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.

4) Process Controller: A key challenge in implementing process controller in the user-space is to ensure a low overhead in switching processes. After experimentally evaluating a few options, we resorted to the following mechanism. 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 moment 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. 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, there is no side-effect in using RT scheduler.

Algorithm 1: UN*i*S Scheduling Loop

Input: $cores$ = Set of CPU cores; $\mathcal{T}$ = monitoring interval; *timer_subsystem, process_controller, monitor* = Handler to UN*i*S system components

```
1  function ScheduleVNFs()
2      timer_subsystem.monitoring_timer.start(T)
       /* The system is initialized by running
          the first VNF in every core's wait
          queue and creating corresponding per
          core timers.                          */
3      while true do
           /* Take scheduling decision after
              every T μs                        */
4          if timer_subsystem.monitoring_timer.is_expired() ==
            false then continue
           /* Iterate over each core and check if
              a new VNF can be scheduled        */
5          foreach core ∈ cores do
6              C ← core.cur_vnf
7              if core.timer.is_expired() or
                monitor.num_pkts(C.ingress) ≤ θ_min or
                monitor.num_pkts(C.egress) ≥ θ_max then
                   /* Iterate over the wait queue
                      (WQ) and find a VNF that
                      has sufficient work to do  */
8                  core.WQ.push(C)
9                  N ← core.WQ.pop()
10                 while (C ≠ 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()
13                 end
14             end
               /* If a candidate VNF is found,
                  allocate it a time_slice      */
15             if C ≠ N then
16                 core.timer.stop()
17                 time_slice ← cost_estimator.get_cost(N) * γ
                    * monitor.pkt_cap(N.egress)
18                 process_controller.deactivate(C)
19                 process_controller.activate(N)
20                 core.cur_vnf ← N
21                 core.timer.reset(time_slice)
22             end
23         end
24     end
25     timer_subsystem.monitoring_timer.reset(T)
26 end
```

V. Performance Evaluation

We evaluate the performance of UN*i*S through testbed experiments. In the following, we first describe our experiment setup in Section V-A. 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 V-B), (ii) SFC with fixed but non-uniform cost VNFs (Section V-C), and (iii) SFC with variable cost (traffic dependent) VNFs (Section V-D), and (iv) one or more SFCs deployed across multiple CPU cores (Section V-E). We conclude this section with a discussion on cost vs. benefit of using intrusive and non-intrusive approach.

A. Experiment Setup

1) Testbed: Our testbed consists of two physical machines with identical configuration connected back to back with each other. 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 10Gbps NIC, 3.3Ghz 4-core Intel Xeon E3-1230v3 CPU, and 16GB of memory. When running UN*i*S, we isolate all the CPU cores except core 0 from the kernel scheduler and use them for VNF deployment.

2) VNFs and Workload: We use two types of VNF in our experiments: (i) a fixed cost VNF whose packet processing cost is fixed and does not depend on packet size, (*e.g.*, similar to a layer 2-4 firewall), and (ii) a variable cost VNF whose packet processing cost is a function of packet size (*e.g.*, a WAN optimizer performing payload compression). For fixed cost VNFs we use the same lightweight VNF used in Section III 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 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 64B to MTU size and consider the average packet processing latency over all sizes as their cost.

We use pktgen-dpdk [16] and Moongen [29] 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 [30] from [31], exhibiting a bi-modal 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.

3) Compared Approaches: We compare UN*i*S with an intrusive co-operative scheduling approach similar to [14]. In the intrusive approach, the VNF is designed to voluntarily yield CPU for other competing VNFs after processing a certain number of batches (we experimentally found 8 to be a good choice for this parameter). Due to the voluntary yields, the `time_slice` allocated to VNFs by RT scheduler does not have any impact on VNF performance.

4) Evaluation Metrics:

a) Throughput and Latency: We measure the throughput and packet processing latency for 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.

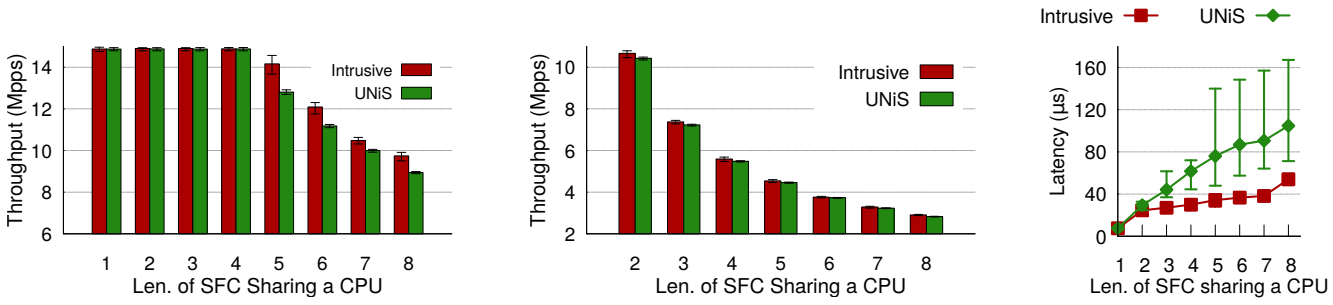
b) VNF density: VNF density of a scheduling approach is measured by fixing a target throughput and determining the maximum length of an SFC chain (*i.e.*, number of VNFs) that can be deployed on a single CPU to sustain that throughput.

B. SFC with fixed and uniform cost VNFs

Our first set of experiments measure how much does the non-intrusive scheduling approach deviates from the intrusive approach in terms of throughput. We deploy SFCs of different lengths composed of identical VNFs with fixed packet processing cost (all light VNFs) on a single CPU core and present throughput results for the smallest (*i.e.*, 64B) packet size in Fig. 3(a). Up to an SFC of length 4, both the intrusive approach and UN*i*S are able to sustain line rate. From length 5 and beyond, packet processing throughput drops below line rate and UN*i*S is not able to match that of the Intrusive approach. However, the deviation from the Intrusive approach was no more than 10% over all chain lengths. Note that the lighter the VNF the more the impact of accurate `time_slice` allocation. Therefore, this scenario with light VNFs measures the worst case performance deviation. In reality, with increasing VNF processing cost we expect the gap to be smaller. We confirm this hypothesis through another set of experimental results presented in Fig. 3(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 perform a set of experiments to measure the extent of latency incurred by the packets when the VNFs have some processing load (Fig. 3(c)). For the longest SFC, packets experience $\approx$2.5$\times$ more latency on average in UN*i*S compared to 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 to UN*i*S.

We also present VNF density by varying the target throughput in Fig. 4. We conduct the experiment by using all three flavors of VNFs, *i.e.*, light, medium, and heavy. UN*i*S has identical VNF density in most cases compared to the intrusive approach. Even when UN*i*S packed less number of VNFs than the intrusive approach the difference is less than 10%.



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

Fig. 3. SFC with fixed but non-uniform cost VNFs

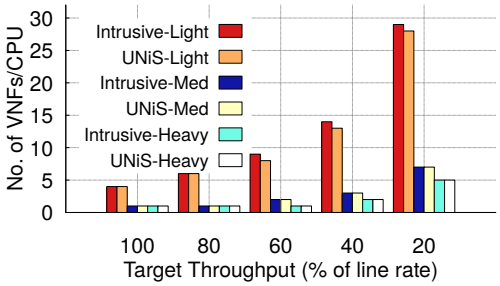


Fig. 4. VNF density on a Single Core with fixed cost VNFs in an SFC

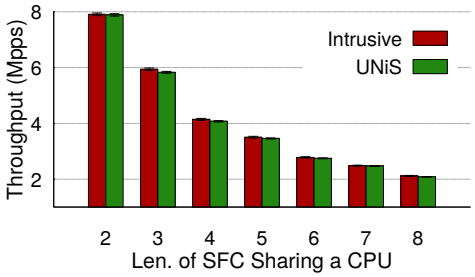


Fig. 5. SFC Composed of VNFs with fixed but non-uniform processing cost

C. SFC with fixed but non-uniform cost VNFs

In our next scenario, we deploy SFCs of different lengths with an alternating sequence of medium and heavy VNFs, *i.e.*, the VNFs at odd positions are the medium ones and 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. 5. As a result, 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.

D. SFC with variable cost (traffic dependent) VNFs

Previous experiments have not considered variable processing cost of a VNF based on traffic. However, many VNFs that operate on payloads can exhibit different processing costs depending on the packet size. To demonstrate the effectiveness of UN*i*S for such cases, we deploy SFCs composed of chains of variable cost VNFs described in Section V-A2. We play a real traffic trace containing packets of different sizes [31] and report the throughput in Fig. 6. UN*i*S performs very close to the intrusive approach with less than 2% deviation. The buffer occupancy based optimization in Alg. 1 helps UN*i*S in

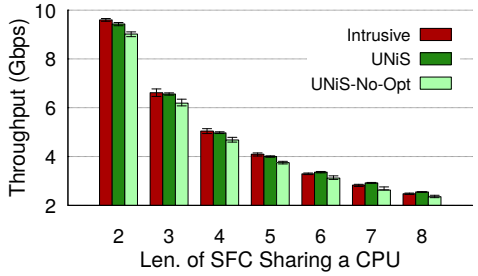


Fig. 6. SFC composed of VNFs with variable processing cost (function of packet size) under real traffic load from [31]

this scenario to fine tune the `time_slice` allocated to the VNFs, which was computed using a static VNF profile in the first place. We perform another set of experiments where we turn off the occupancy based optimization and show its impact (UN*i*S-No-Opt in Fig. 6). The added optimization results in as much as ≈10% performance improvement, which can be significant in absolute terms when packets are being processed at a rate of several Gbps.

E. 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 I. In Scenario (a), there are multiple SFCs deployed on a single core. With the Intrusive approach, both SFCs achieve equal throughput of 5.31Mpps for 64B packets. We also observe a near equal throughput distribution across S1 and S2 for UN*i*S, indicating no SFC is starving for CPU. Scenario (b) has two SFCs deployed across two cores and each core hosts VNFs from two SFCs. Similar to (a), the intrusive approach shows equal throughput for both SFCs. We also observe similar behavior in this case for UN*i*S, validating the fact that no starvation is occurring when CPU cores are hosting VNFs from multiple SFCs and SFCs are deployed across multiple cores. Finally, scenario (c) deploys one SFC across multiple cores and here we see UN*i*S achieving a throughput within 1.3% of the intrusive approach.

F. Discussion: Cost vs. Benefit

Our experimental results suggest that even with a non-intrusive approach, UN*i*S is able to schedule VNFs in an

TABLE I
RESULTS FOR MULTIPLE SFCS ACROSS MULTIPLE CPUS

# VNFs in SFC	# VNFs on Core-1	# VNFs on Core-2	Int. Thput. (Mpps)	UN*i*S Thput. (Mpps)
(a) S1 = 3 S2 = 1	S1 = 3 S2 = 1	–	S1 = 5.31 S2 = 5.31	S1 = 5.30 S1 = 5.21
(b) S1 = 4 S2 = 4	S1 = 3 S2 = 1	S1 = 1 S2 = 3	S1 = 5.24 S2 = 5.24	S1 = 5.10 S2 = 5.14
(c) S1 = 8	S1 = 4	S1 = 4	S1 =5.41	S1 = 5.34

SFC to achieve a comparable performance to that of an intrusive approach. Intrusive approaches such as co-operative scheduling and the one described in [13] 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 [13], 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.

VI. RELATED WORKS

Scheduling has been extensively studied in various areas of systems and networking such as cluster scheduling [32]–[34], packet scheduling [35], [36], flow scheduling [37], [38] among others. What makes NFV scheduling different from other areas is that VNF processing cost depends on a multitude of factors including packet size, packet arrival rate, VNF configuration, and packet contents to name a few. In contrast, in other areas that are close to NFV scheduling (*e.g.*, packet/flow scheduling, joint compute-network scheduling) processing costs are much more predictable and is usually dependent on lesser number of variables (*e.g.*, flow completion time depends on the amount of data to transfer and available bandwidth). In this section, we discuss recent developments in scheduling with a particular focus on NFV and contrast UN*i*S with the state-of-the-art.

A. Analytical Models for NFV Scheduling

There has been substantial developments in addressing VNF scheduling from a theoretical point of view using different methodologies [22]–[27]. Riera *et al.*, presents one of the early integer program formulation for scheduling VNFs on a set of servers [22], which is limited in scalability. Mijumbi *et al.*, presents an optimization model to jointly map and schedule VNFs on physical machines [23]. 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 [25] and [26]. Both proposals use a mixed integer linear program to optimally solve the problem and then use a tabu search meta-heuristic [25] and column generation [26] 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 [24]) and have been solved using methods such as game theory [27]. The optimization models for different variations of VNF scheduling is focused on scheduling SFCs across multiple machines, considering the network topology, available compute and network resources *etc*. In contrast, UN*i*S's focus is to serve as a viable alternative to local OS schedulers for VNF scheduling. Moreover, the methods used in this line of research are more suitable for devising an offline execution plan rather than for online scheduling decision making at micro-second time, which is a key requirement for UN*i*S. An analytical model focusing on processor sharing among VNFs in a single server is presented in [39]. The objective of this work is to reduce the time an outgoing NIC remains idle. In contrast, our objective is to pack as many VNFs as possible on the CPUs and achieve comparable throughput to an intrusive scheduling mechanism.

B. Systems for NFV Scheduling

Flurries [40] and NFVNice [13] 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 [13] proposes a back-pressure based mechanism to slow down an SFC by setting Explicit Congestion Notification (ECN) bit inside packets when VNFs experience packet drops. However, both of these approaches are intrusive, *i.e.*, they require the VNFs to be built with scheduler provided library to get a better insight into the VNFs or assume usage of certain mechanisms by the VNFs (*e.g.*, set ECN bit in packet). Another approach is to write VNFs from scratch to co-operate with other VNFs for better scheduling (similar to [14]). 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.

VII. CONCLUSION AND FUTURE WORK

In this paper, we presented the design, implementation and evaluation of 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. 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 perform very close to the intrusive scheduling method.

Building on these promising results, our next goal is to extend UN*i*S to consider SFCs deployed across multiple machines in a cluster. Another research direction is to focus on dynamically adjusting `time_slice` allocation in UN*i*S and better adapt to factors such as packet size, packet inter-arrival time, VNF configuration, *etc*.

REFERENCES

[1] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

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

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

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

[6] 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.

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

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

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

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

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

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

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

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

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

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

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

[18] 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.

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

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

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

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

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

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

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

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

[33] 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.

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

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

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

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

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

[39] 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.

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