NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains

Sameer G Kulkarni[*★], Wei Zhang[‡], Jinho Hwang[§], Shriram Rajagopalan[§], K.K. Ramakrishnan[†], Timothy Wood[‡], Mayutan Arumaithurai[*] and Xiaoming Fu[*]

[*]University of Göttingen, Germany, [‡]George Washington University, [§]IBM T J Watson Research Center, [†]University of California, Riverside.

ABSTRACT

Managing Network Function (NF) service chains requires careful system resource management. We propose *NFVnice*, a user space NF scheduling and service chain management framework to provide fair, efficient and dynamic resource scheduling capabilities on Network Function Virtualization (NFV) platforms. The NFVnice framework monitors load on a service chain at high frequency (1000Hz) and employs backpressure to shed load early in the service chain, thereby preventing wasted work. Borrowing concepts such as rate proportional scheduling from hardware packet schedulers, CPU shares are computed by accounting for heterogeneous packet processing costs of NFs, I/O, and traffic arrival characteristics. By leveraging cgroups, a user space process scheduling abstraction exposed by the operating system, NFVnice is capable of controlling when network functions should be scheduled. NFVnice improves NF performance by complementing the capabilities of the OS scheduler but without requiring changes to the OS's scheduling mechanisms. Our controlled experiments show that NFVnice provides the appropriate rate-cost proportional fair share of CPU to NFs and significantly improves NF performance (throughput and loss) by reducing wasted work across an NF chain, compared to using the default OS scheduler. NFVnice achieves this even for heterogeneous NFs with vastly different computational costs and for heterogeneous workloads.

CCS CONCEPTS

• **Networks → Network resources allocation**; **Network management**; *Middle boxes / network appliances*; *Packet scheduling*;

KEYWORDS

Network Functions (NF), Backpressure, NF-Scheduling, Cgroups.

ACM Reference format:
S.G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K.K. Ramakrishnan, T. Wood, M. Arumaithurai, X. Fu. 2017. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098828

★Work done while Sameer was at University of California, Riverside, as part of secondment in the EU FP7 Marie Curie CleanSky ITN Research Project.

1 INTRODUCTION

Network Function Virtualization (NFV) seeks to implement network functions and middlebox services such as firewalls, NAT, proxies, deep packet inspection, WAN optimization, etc., in software instead of purpose-built hardware appliances. These software based network functions can be run on top of commercial-off-the-shelf (COTS) hardware, with virtualized network functions (NFs). Network functions, however, often are chained together [20], where a packet is processed by a sequence of NFs before being forwarded to the destination.

The advent of container technologies like Docker [34] enables network operators to densely pack a single NFV appliance (VM/bare metal) with large numbers of network functions at runtime. Even though NFV platforms are typically capable of processing packets at line rate, without efficient management of system resources in such densely packed environments, service chains can result in serious performance degradation because bottleneck NFs may drop packets that have already been processed by upstream NFs, resulting in wasted work in the service chain.

NF processing has to address a combination of requirements. Just as hardware switches and routers provide rate-proportional scheduling for packet flows, an NFV platform has to provide a fair processing of packet flows. Secondly, the tasks running on the NFV platform may have heterogeneous processing requirements that OS schedulers (unlike hardware switches) address using their typical fair scheduling mechanisms. OS schedulers, however, do not treat packet flows fairly in proportion to their arrival rate. Thus, NF processing requires a re-thinking of the system resource management framework to address both these requirements. Moreover, standard OS schedulers: a) do not have the right metrics and primitives to ensure fairness between NFs that deal with the same or different packet flows; and b) do not make scheduling decisions that account for chain level information. If the scheduler allocates more processing to an upstream NF and the downstream NF becomes overloaded, packets are dropped by the downstream NF. This results in inefficient processing and wasting the work done by the upstream NF. OS schedulers also need to be adapted to work with user space data plane frameworks such as Intel's DPDK [1]. They have to be cognizant of NUMA (Non-uniform Memory Access) concerns of NF processingand the dependencies among NFs in a service chain. Determining how to dynamically schedule NFs is key to achieving high performance and scalability for diverse service chains, especially in a scenario where multiple NFs are contending for a CPU core[1]

[1]While CPU core counts are increasing in modern hardware, they are likely to remain a bottleneck resource, especially when service chains are densely packed into a single machine (as is often the case with several proposed approaches [23, 52]).

Hardware routers and switches that employ sophisticated scheduling algorithms such as rate proportional scheduling [40, 50] have predictable performance per-packet, because processing resources are allocated fairly to meet QoS requirements and bottlenecks are avoided by design. However, NFV platforms are necessarily different because: a) the OS scheduler does not know a priori, the capacity or processing requirements for each NF; b) an NF may have variable per-packet costs (*e.g.,* some packets may trigger DNS lookup, which are expensive to process, and others may just be an inexpensive header match). With NFV service chains, there is a need to be aware of the computational demands for packet processing. There can also be sporadic blocking of NFs due to I/O (read/write) stalls.

A further consideration is that routers and switches 'simply' drop packets when congested. However, an NF in a service chain that drops packets can result in considerable wasted processing at NFs earlier in the chain. These wasted resources could be gainfully utilized by other NFs being scheduled on the same CPU core to process other packet flows.

We posit that a scheduling framework for NFV service chains has to simultaneously account for both task level scheduling on processing cores and packet level scheduling. This combined problem is what poses a challenge: *When you get a packet, you have to decide which task has to run, and also which packets to process, and for how long.*

To solve these problems we propose NFVnice, an NFV management framework that provides fair and efficient resource allocations to NF service chains. NFVnice focuses on the scheduling and control problems of NFs running on shared CPU cores, and considers a variety of realistic issues such as bottlenecked NFs in a chain, and the impact of NFs that perform disk I/O accesses, which naturally complicate scheduling decisions. NFVnice makes the following contributions:

- Automatically tuning CPU scheduling parameters in order to provide a fair allocation that weighs NFs based on both their packet arrival rate and the required computation cost.
- Determining when NFs are eligible to get a CPU share and when they need to yield the CPU, entirely from user space, improving throughput and fairness regardless of the kernel scheduler being used.
- Leveraging the scheduling flexibility to achieve backpressure for service chain-level congestion control, that avoids unnecessary packet processing early in a chain if the packet might be dropped later on.
- Extending backpressure to apply not only to adjacent NFs in a service chain, but for full service chains and managing congestion across hosts using ECN.
- Presenting a scheduler-agnostic framework that does not require any operating system or kernel modifications.

We have implemented NFVniceon top of OpenNetVM [54], a DPDK-based NFV platform that runs NFs in separate processes or containers to facilitate deployment. Our evaluation shows that NFVnice can support different kernel schedulers, while substantially improving throughput and providing fair CPU allocation based on processing requirements. In controlled experiments using the vanilla CFS BATCH [37] scheduler, NFVnice reduces packet drops from 3Mpps (million packets per second) to just 0.01Mpps during overload conditions. NFVnice provides performance isolation for TCP flows when there are competing UDP flows, improving throughput of TCP flows from 30Mbps to 4Gbps, without penalizing UDP flows, by avoiding wasted work. While this is scenario dependent, we believe the performance benefits of NFVnice are compelling. Further, our evaluations demonstrate that NFVnice, because of the dynamic backpressure, is resilient to the variability in packet-processing cost of the NFs, yielding considerable improvement in throughput even for the large service chains (including chains that span multiple cores).

2 BACKGROUND AND MOTIVATION

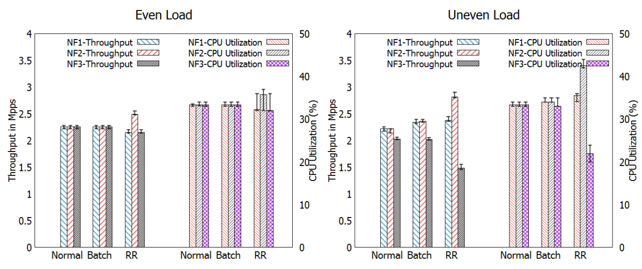
2.1 Diversity, Fairness, and Chain Efficiency

The middleboxes that are being deployed in industry are diverse in their applications as well as in their complexity and processing requirements. ETSI standards [13] show that NFs have dramatically different processing and performance requirements. Measurements of existing NFs show the variation in CPU demand and per packet latency: some NFs have per-core throughput in the order of million packets per second (Mpps), *e.g.,* switches; others have throughputs as low as a few kilo pps, *e.g.,* encryption engines.

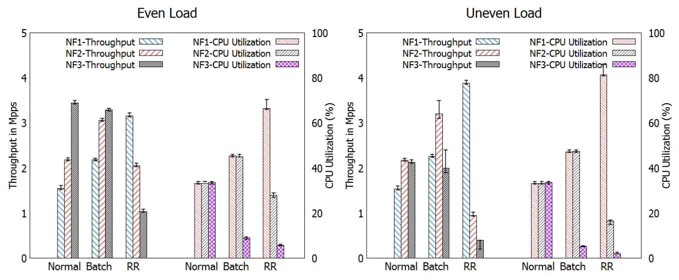
Fair Scheduling: Determining how to allocate CPU time to network functions in order to provide fair and efficient chain performance despite NF diversity is the focus of our work. Defining "fairness" when NFs may have completely different requirements or behavior can be difficult. A measure of fairness that we leverage is the work on Rate Proportional Servers [40, 50], that apportion resources (CPU cycles) to NFs based on the combination of an NF's arrival rate and its processing cost. Intuitively, if either one of these factors is fixed, then we expect its CPU allocation to be proportional to the other metric. For example, if two NFs have the same computation cost but one has twice the arrival rate, then we want it to have twice the output rate relative to the second NF. Alternatively, if the NFs have the same arrival rate, but one requires twice the processing cost, then we expect the heavy NF to get about twice as much CPU time, resulting in both NFs having the same output rate. This definition of fairness can of course be supplemented with a prioritization factor, allowing an understandable and consistent way to provide differentiated service for NFs.

Unfortunately, standard CPU schedulers do not have sufficient information to allocate resources in a way that provides rate-cost proportional fairness. CPU schedulers typically try to provide fair allocation of processing time, but if computation costs vary between NFs this cannot provide rate-cost fairness. Therefore, NFVnice must enhance the scheduler with more information so that it can appropriately allocate CPU time to provide correctly weighted allocations.

We adopt the notion of rate-cost proportional fairness for two fundamental reasons: i) it not only seeks to maximize the throughput for a given load across NFs, but even in the worst case scenarios (highly uneven and high overload across competing NFs), it ensures that all competing NFs get a minimal CPU share necessary to progress the NFs; and ii) the rate-cost proportional fairness is general and flexible, so that it can be tuned to meet the QoS policies desired by the operator. Further, the approach ensures that when contending NFs include malicious NFs (those that fail to yield), or



(a) Homogeneous NFs



(b) Heterogeneous NFs

Figure 1: The scheduler alone is unable to provide fair resource allocations that account for processing cost and load.

misbehaving NFs (get stuck in a loop making no progress), such NFs do not consume the CPU excessively, impeding the progress of other NFs. While the Linux default scheduler addresses this through the notion of a virtual run-time for each running task, we fine-tune that capability to provide the correct share of the CPU for an NF, rather than just allocating an equal share of the CPU for each contending NF.

Efficient Chaining: Beyond simply allocating CPU time fairly to NFs on a single core, the combination of NFs into service chains demands careful resource management across the chain to minimize the impact of bottlenecks. Processing a packet only to have it dropped from a subsequent bottleneck's queue is wasteful, and a recipe for *receive livelock* [27, 36].

When an NF (whether a single NF or one in a service chain) is overloaded, packet drops become inevitable, and processing resources already consumed by those packets are wasted. For responsive flows, such as a TCP flow, congestion control and avoidance using packet drop methods such as RED, REM, SFQ, CSFQ [14, 15, 28, 51] and feedback with Explicit Congestion Notification (ECN) [42] can cause the flows to adapt their rates to the available capacity on an end-to-end basis. However, for non-responsive flows (e.g., UDP), a local, rapidly adapting, method is backpressure, which can propagate information regarding a congested resource upstream (i.e., to previous NFs in the chain). NFVnice allows the upstream node to determine either to propagate the backpressure information further upstream or drop packets, thus minimizing wasted work. It is important however to ensure that effects such as head-of-the-line blocking or unfairness do not creep in as a result.

2.2 Existing OS schedulers are ill-suited for NFV deployment

Linux provides several different process schedulers, with the Completely Fair Scheduler (CFS) [37] being the default since kernel 2.6.23. In this work we focus on three schedulers: i) CFS Normal, ii) CFS Batch, and Round Robin.

The CFS class of schedulers use a nanosecond resolution timer to provide fine granularity scheduling decisions. Each task in CFS maintains a monotonically increasing virtual run-time which determines the order and quantum of CPU assignment to these tasks. The time-slice is not fixed, but is determined relative to the run-time of the contending tasks in a time-ordered red-black tree [9, 19]. The task with the smallest run-time (the left most node in the ordered red-black tree) is scheduled to run until either the task voluntarily yields, or consumes the allotted time-slice. If it consumes the allocated time-slice, it is re-inserted into the red-black tree based on its cumulative run-time consumed so far. The CFS scheduler is analogous to weighted fair queueing (WFQ) scheduling [10, 53]. Thus, CFS ensures a fair proportion of CPU allocation to all the tasks. The CFS Batch variant has fewer timer interrupts than normal CFS, leading to a longer time quantum and fewer context switches, while still offering fairness. The Round Robin scheduler simply cycles through processes with a 100 msec time quantum, but does not attempt to offer any concept of fairness.

To explore the impact of these schedulers on NFV applications we consider a simple deployment with three NF processes sharing a CPU core. The NFs run atop a DPDK-based NFV platform that efficiently delivers packets to the NFs. We look at two workloads: 1) equal offered load to all NFs of 5 Mpps; 2) unequal offered load, with NF1 and NF2 getting 6 Mpps, and NF3 getting 3 Mpps. We also consider the case where NFs have different computation costs. As described above, the desirable behavior is for NFs to be allocated resources in proportion to both their arrival rate and processing requirements.

Table 1: Context Switches for Homogeneous NFs

	Even Load						Uneven Load					
	SCHED_ NORMAL		SCHED_ BATCH		SCHED_ RR		SCHED_ NORMAL		SCHED_ BATCH		SCHED_ RR	
NF	csw-ch/s	nvc swch /s	csw-ch/s	nv cswch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s
NF1	0	339	0	333	266	3	0	3544	0	527	247	5
NF2	0	334	0	333	265	4	0	6205	0	479	246	5
NF3	0	333	0	334	266	3	9753	9	1007	0	248	3

Table 2: Context Switches for Heterogeneous NFs

	Even Load						Uneven Load					
	SCHED_ NORMAL		SCHED_ BATCH		SCHED_ RR		SCHED_ NORMAL		SCHED_ BATCH		SCHED_ RR	
NF	csw-ch/s	nvc swch /s	csw-ch/s	nv cswch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s
NF1	0	33785	0	504	198	7	0	38585	0	503	85	10
NF2	0	32214	1	505	204	2	0	41089	4	496	92	1
NF3	65796	107	1010	8	206	0	79479	85	1004	4	93	0

In our first test, illustrated in Figure 1a, all 3 NFs have equal computation cost (roughly 250 CPU cycles per packet). With an even load sent to all NFs, we find that the three schedulers perform about the same, with an equal division of CPU time leading to equal throughputs for each NF. However, reducing the traffic to NF3 by half shows the different behaviour of the schedulers:

while the CFS-based schedulers continue to evenly divide the CPU (CFS's definition of fairness), the RR scheduler allocates CPU time in proportion to the arrival rate, which better matches our notion of rate proportional fairness. This happens because RR uses a time quantum that is substantially longer than an NF ever needs, so NFs which yield the CPU earlier (*i.e.,* because they have fewer packets to process) receive less CPU time and thus have lower throughput. Note the context switches (shown in Table 1) in RR case are predominantly voluntary context switches, while the CFS based schedulers incur non-voluntary context switches.

We next consider heterogeneous NFs (computation costs: NF1= 500, NF2=250 and NF3=50 CPU cycles) with even or uneven load. Figure 1b shows that when arrival rates are the same, none of the schedulers are able to provide our fairness goal—an equal output rate for all three NFs. CFS Normal always apportions CPU equally, regardless of offered load and NF processing cost, so the lighter weight NF3 gets the highest throughput. The RR scheduler is the opposite since it gives each NF an equal chance to run, but does not limit the time the NF runs for. The CFS Batch scheduler is in between these extremes since it seeks to provide fairness, but over longer time periods. Notably, the Batch scheduler provides NF3 almost the same throughput as Normal CFS, despite allocating it substantially less CPU. The reason for this is that Normal CFS can incur a very large number of context switches due to its goal of providing very fine-grained fairness. Since Batch mode reduces scheduler preemption, it has substantially fewer non-voluntary context switches—reducing from 65K to 1K per second—as illustrated in the Table 2. While RR also has low context switch overhead, it allows heavy weight NFs to greedily consume the CPU, nearly starving NF3.

These results show that just having the Linux scheduler handle scheduling NFs has undesirable results as by itself it is unable to adapt to both varying per-packet processing requirements of NFs and packet arrival rates. Moreover, it is important to avoid the overheads of excessive context switches. All of these scheduling requirements must be met on a per-core basis, while accounting for the behaviour of chains spanning multiple cores or servers.

3 DESIGN AND IMPLEMENTATION

In an NFV platform, at the top of the stack are one or more network functions that must be scheduled in such a way that idle work (i.e., while waiting for packets) is minimized and load on the service chain is shed *as early as possible* so as to avoid wasted work. However, the operating system's process scheduler that lies at the bottom of the software stack remains completely application agnostic, with its goal of providing a fair share of system resources to all processes. As shown in the prior section, the kernel scheduler's metrics for scheduling are along orthogonal dimensions to those desired by the network functions. NFVnice bridges the gap by translating the scheduling requirements at the NFV application layer to a format consumable by the operating system.

The design of NFVnice centers around the concept of assisted preemptive scheduling, where network functions provide hints to the underlying OS with regard to their utilization. In addition to monitoring the average computation time of a network function per packet, NFVnice needs to know when NFs in a chain are overloaded, or blocked on packet/disk I/O. The queues between NFs in a service chain serve as a good indicator of pending work at each NF. To facilitate the process of providing these metrics from the NF implementation to the underlying operating system, NFVnice provides network function implementations with an abstraction library called *libnf*. In addition to the usual tasks such as efficient reading/writing packets from/to the network at line rate and overlapping processing with non-blocking asynchronous I/O, *libnf* co-ordinates with the NFVnice platform to schedule/deschedule a network function as necessary.

Modifying the OS scheduler to be aware of various queues in the NFV platform is an onerous task that might lead to unnecessary maintenance overhead and potential system instability. One approach is to change the priority of the NF based on the queue length of packet at that NF. This will have the effect of increasing the number of CPU cycles provided to that NF. This will require the change to occur frequently as the queue length varies. The change requires a system call, which consumes CPU cycles and adds latency. In addition, with service chains, as the queue at an upstream NF builds, its priority has to be raised to process packets and deliver to a queue at the downstream NF. Then, the downstream NF's priority will have to be raised. We believe that this can lead to instability because of frequent changes and the delay involved in effecting the change. This only gets worse with complex service chains, where an NF is both an upstream NF for one service chain and a downstream NF for another service chain. Instead, NFVnice leverages cgroups [5, 33], a standard user space primitive provided by the operating system to manipulate process scheduling. NFVnice monitors queue sizes, computation times and I/O activities in user space with the help of *libnf* and manipulates scheduling weights accordingly.

3.1 System Components

Figure 2 illustrates the key components of the NFVnice platform. We leverage DPDK for fast user space networking [1]. Our NFV platform is implemented as a system of queues that hold packet descriptors pointing to shared memory regions. The NF Manager runs on a dedicated set of cores and is responsible for ferrying packet references between the NIC queues and NF queues in an efficient manner. When packets arrive to the NIC, Rx threads in the NF Manager take advantage of DPDK's poll mode driver to deliver the packets into a shared memory region accessible to all the NFs. The Rx thread does a lookup in the Flow Table to direct the packet to the appropriate NF. Once a flow is matched to an NF, packet descriptors are copied into the NF's receive ring buffer and the Wakeup subsystem brings the NF process into the runnable state. After being processed by an NF, the NF Manager's Tx Threads move packets through the remainder of the chain. This provides zero-copy packet movement.

Service chains can be configured during system startup using simple configuration files or from an external orchestrator such as an SDN controller. When an NF finishes with a packet, it enqueues it in its Tx queue, where it is read by the manager and redirected to the Rx queue of the next NF in the chain. The NF Manager also picks up packets from the Tx queue of the last NF in the chain, and sends it out over the network.

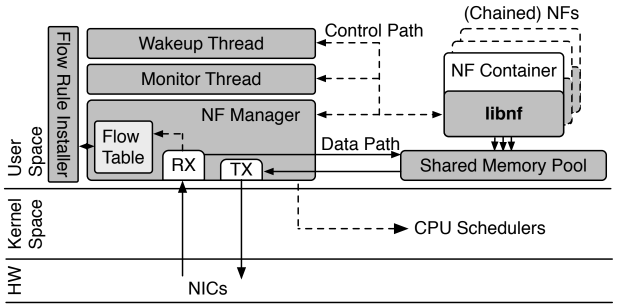


Figure 2: NFVnice Building Blocks

We have designed NFVnice to provide high performance processing of NF service chains. The NF Manager's scheduling subsystem determines when an NF should be active and how much CPU time it should be allocated relative to other NFs. The backpressure subsystem provides chain-aware management, preventing NFs from spending time processing packets that are likely to be dropped downstream. Finally, the I/O interface facilitates efficient asynchronous storage access for NFs.

System Management and NF deployment: The NF Manager 's (Rx, Tx and Monitor) threads are pinned to separate dedicated cores. The number of Rx, Tx and monitor threads are configurable (`C-Macros`), to meet system needs, and available CPU resources. Similarly, the maximum number of NFs and maximum chain length can be configured. NFVnice allows NFs and NF service chains to be deployed as independent processes or Docker containers which are linked with *libnf* library. *libnf* exports a simple, minimal interface (9 functions, 2 callbacks and 4 structures), and both the NF Manager and *libnf* leverage the DPDK libraries (ring buffers, timers, memory management). We believe developing or porting NFs or existing docker containers can be reasonably straightforward. For example, a simple bridge NF or a basic monitor NF is less than 100 lines of `C` code.

3.2 Scheduling NFs

Each network function in NFVnice is implemented inside its own process (potentially running in a container). Thus the OS scheduler is responsible for picking which NF to run at any point in time. We believe that rather than design an entirely new scheduler for NFV, it is important to leverage Linux's existing scheduling framework, and use our management framework in user space to tune any of the stock OS schedulers to provide the properties desired for NFV support. In particular, we exploit the CFS Batch scheduler, but NFVnice provides substantially similar benefits to each of the other Linux kernel schedulers. Figure 3 shows the NFVnice scheduling that makes the OS scheduler be governed by NF Manager via cgroups, and ultimately assigns running NFs to shared CPU cores. The detailed description of the figure is in the Sections 3.2 and 3.3.

Activating NFs: NFs that busy wait for packets perform very poorly in a shared CPU environment. Thus it is critical to design the NF framework so that NFs are only activated when there are packets available for them to process, as is done in NFV platforms such as netmap [43] and ClickOS [32]. However, these systems provide only a relatively simple policy for activating an NF: once one or more

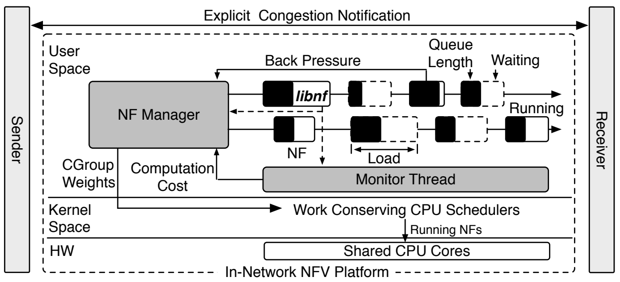


Figure 3: NF Scheduling and Backpressure

packets are available, a signal is sent to the NF so that it will be scheduled to run by the OS scheduler in netmap, or the hypervisor scheduler in ClickOS. While this provides an efficient mechanism for waking NFs, neither system allows for more complex resource management policies, which can lead to unfair CPU allocations across NFs, or inefficient scheduling across chains.

In NFVnice, NFs sleep by blocking on a semaphore shared with the NF Manager, granting the management plane great flexibility in deciding which NFs to activate at a given time. The policy we provide for activating an NF considers the number of packets pending in its queue, its priority relative to other NFs, and knowledge of the queue lengths of downstream NFs in the same chain. This allows the management framework to indirectly affect the CPU scheduling of NFs to be fairness and service-chain aware, without requiring that information be synchronized with the kernel's scheduler.

Relinquishing the CPU: NFs process batches of packets, deciding whether to keep processing or relinquish the CPU between each batch. This decision and all interactions with the management layer, e.g., to receive a batch of packets, are mediated by *libnf*, which in turn exposes a simple interface to developers to write their network function. After a batch of at most 32 packets is processed, *libnf* will check a shared memory flag set by the NF Manager that indicates if it should relinquish the CPU early (e.g., as a result of backpressure, as described below). If the flag is not set, the NF will attempt to process another batch; if the flag has been set or there are no packets available, the NF will block on the semaphore until notified by the Manager. This provides a flexible way for the manager to indicate that an NF should give up the CPU without requiring the kernel's CPU scheduler to be NF-aware.

CPU Scheduler: Since multiple NF processes are likely to be in the runnable state at the same time, it is the operating system's CPU scheduler that must determine which to run and for how long. In the early stages of our work we sought to design a custom CPU scheduler that would incorporate NF information such as queue lengths into its scheduling decisions. However, we found that synchronizing queue length information with the kernel, at the frequency necessary for NF scheduling, incurred overheads that outweighed any benefits.

Linux's CFS Batch scheduler is typically used for long running computationally intensive tasks because it incurs fewer context switches than standard CFS. Since NFVnice carefully controls when individual NF processes are runnable and when they yield the CPU (as described above), the batch scheduler's longer time quantum and less frequent preemption are desirable. In most cases, NFVnice

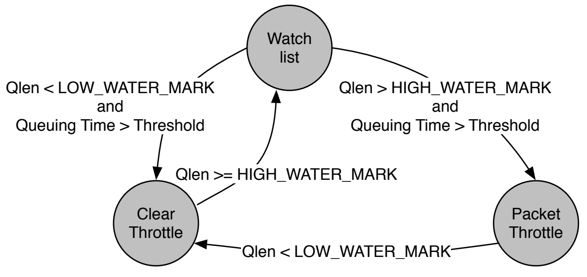

Watch list
Qlen < LOW_WATER_MARK and Queuing Time > Threshold
Qlen > HIGH_WATER_MARK and Queuing Time > Threshold
Qlen >= HIGH_WATER_MARK
Clear Throttle
Packet Throttle
Qlen < LOW_WATER_MARK

Figure 4: Backpressure State Diagram

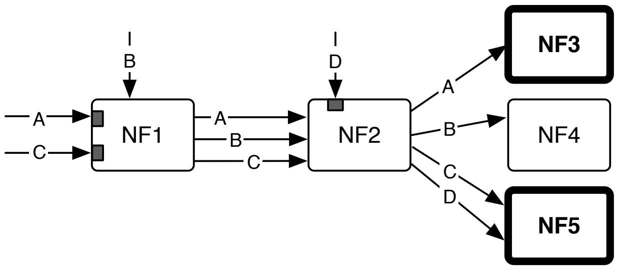


Figure 5: Overloaded NFs (in bold) cause back pressure at the entry points for service chains A, C, and D.

NFs relinquish the CPU due to policies controlled by the manager, rather than through an involuntary context switch. This reduces overhead and helps NFVnice prioritize the most important NF for processing without requiring information sharing between user and kernel space.

Assigning CPU Weights: NFVnice provides mechanisms to monitor a network function to estimate its CPU requirements, and to adjust its scheduling weight. Policies in the NF Manager can then dynamically tune the scheduling weights assigned to each process in order to meet operator specified priority requirements.

The packet arrival rate for a given NF can be easily estimated by either the NF or the NF Manager. We measure the service time to process a packet inside each NF using *libnf*. To avoid outliers from skewing these measurements (e.g., if a context switch occurs in the middle of processing a packet), we maintain a histogram of timings, allowing NFVnice to efficiently estimate the service time at different percentiles.

For each NF i on a shared core, we calculate $load(i) = \lambda_i * s_i$, the product of arrival rate, λ, and service time, s. We then find the total load on each core, such as core m, $TotalLoad(m) = \sum_{i=1}^{n} load(i)$, and assign cpu shares for NF_i on $core_m$ following the formula:

$$Shares_i = Priority_i * \frac{load(i)}{TotalLoad(m)}$$

This provides an allocation of CPU weights that provides rate proportional fairness to each NF. The $Priority_i$ parameter can be tuned if desired to provide differential service to NFs. Tuning priority in this way provides a more intuitive level of control than directly working with the CPU priorities exposed by the scheduler since it is normalized by the NF's load.

3.3 Backpressure

A key goal of NFVnice is to avoid wasting work, *i.e.,* preventing an upstream NF from processing packets if they are just going to be dropped at a downstream NF later in the chain that has become overloaded. We achieve this through backpressure, which ensures bottlenecks are quickly detected while minimizing the effects of head of line blocking.

Cross-Chain Pressure: The NF Manager is in an ideal position to observe behavior across NFs since it assists in moving packets between them. When one of the NF Manager's TX threads detects that the receive queue for an NF is above a high watermark (HIGH_WATER_MARK) and queuing time is above threshold, then it examines all packets in the NF's queue to determine what service chain they are a part of. NFVnice then enables *service chain-specific* packet dropping at the upstream NFs. NF Manager maintains states of each NF, and in this case, it moves the NF's state from *backpressure watch list* to *packet throttle* as shown in Figure 4. When the queue length becomes less than a low watermark (LOW_WATER_MARK), the state moves to *clear throttle*, then again moves to the watch list if the queue length goes beyond the high mark.

The backpressure operation is illustrated in Figure 5, where four service chains (A-D) pass through several different NFs. The bold NFs (3 and 5) are currently overloaded. The NF Manager detects this and applies back pressure to flows A, C, and D. This is performed upstream where those flows first enter the system, minimizing wasted work. Note that backpressure is selective based on service chain, so packets for service chain B are not affected at all. Service chains can be defined at fine granularity (*e.g.,* at the flow-level) in order to minimize head of line blocking.

This form of system-wide backpressure offers a simple mechanism that can provide substantial performance benefits. The backpressure subsystem employs hysteresis control to prevent NFs rapidly switching between modes. Backpressure is enabled when the queue length exceeds a high watermark and is only disabled once it falls below the low watermark.

Local Optimization and ECN: NFVnice also supports simple, local backpressure, *i.e.,* an NF will block if its output TX queue becomes full. This can happen either because downstream NFs are slow, or because the NF Manager TX Thread responsible for the queue is overloaded. Local backpressure is entirely NF-driven, and requires no coordination with the manager, so we use it to handle short bursts and cases where the manager is overloaded.

We also consider the fact that an NFVnice middlebox server might only be one in a chain spread across several hosts. To facilitate congestion control across machines, the NF Manager will also mark the ECN bits in TCP flows in order to facilitate end-to-end management. Since ECN works at longer timescales, we monitor queue lengths with an exponentially weighted moving average and use that to trigger marking of flows following [42].

3.4 Facilitating I/O

A network function could block when its receive ring buffer is empty or when it is waiting to complete I/O requests to the underlying storage. In both cases, NF implementations running on the NFVnice platform are expected to yield the CPU, returning any unused CPU cycles back to the scheduling pool. In case of

```
// Read the next packet from the receive ring buffer
packet_descriptor* libnf_read_pkt();

// Output the processed packet to specified destination
int libnf_write_pkt(packet_descriptor*);

// Enqueue request to read from storage. Flow specific data
    can be stored in context
int libnf_read_data(int fd, void *buf,
     size_t size, size_t offset,
     void (*callback_fn)(void *), void *context);

// Enqueue request to write to storage. Flow specific data
    can be stored in context
int libnf_write_data(int fd, void *buf,
     size_t size, size_t offset,
     void (*callback_fn)(void *), void *context);
```

Figure 6: *libnf* API exposed to network function implementations.

I/O, NF implementations should use asynchronous I/O to overlap packet processing with background I/O to maintain throughput. NFVnice provides a simple library called *libnf* that abstracts such complexities from the NF implementation.

The *libnf* library exposes a simple set of APIs that allow the application code to read/write packets from the network, and read-/write data from storage. The APIs are shown in Listing 6. If the receive ring buffer is empty while calling the `libnf_read_pkt` API, *libnf* notifies the NF manager and blocks the NF until further packets are available in the buffer.

In case of I/O, an NF implementation uses the `libnf_read_data` and `libnf_write_data` APIs. I/O requests can be queued along with a callback function that runs in a separate thread context. Using batched asynchronous I/O with double buffering, *libnf* enables the NF implementation to put the processing of one or more packets on hold, while continuing processing of other packets unhindered.

Batching reads and writes allows an NF to continue execution without waiting for I/O completion. The size of the batches and the flush interval is tunable by the NF implementation. Double buffering enables *libnf* to service one set of I/O requests asynchronously while the other buffer is filled up by the NF. When both buffers are full, *libnf* suspends the execution of the NF and yields the CPU.

3.5 Optimizations

Separating overload detection and control. Since the NFV platform [23] must process millions of packets per second to meet line rates, we separate out overload detection from the control mechanisms required to respond to it. The NF Manager's Tx threads are well situated to detect when an NF is becoming backlogged as it is their responsibility to enqueue new packets to each NF's Tx queue. Using a single DPDK's enqueue interface, the Tx thread enqueues a packet to a NF's Rx queue if the queue is below the high watermark, while getting feedback about the queue's state in the return value. When overload is detected, an overload flag is set in the meta data structure related to the NF.

The control decision to apply backpressure is delegated to th NF Manager's Wakeup thread. The Wakeup thread scans through the list of NFs classifying them into two categories: ones where backpressure should be applied and ones that need to be woken up. This separation simplifies the critical path in the Tx threads and also provides some hysteresis control, since a short burst of packets causing an NF to exceeds its threshold may have already been processed by the time the Wakeup thread considers it for backpressure.

Separating load estimation and CPU allocation. The load on an NF is a product of its packet arrival rate and the per-packet processing time. The scheduler weight is calculated based on the load and the cgroup's weights for the NF are updated. Since changing a weight requires writing to the Linux sysfs, it is critical that this be done outside of the packet processing data path. *libnf* merely collects samples of packet processing times, while the NF Manager computes the load and assigns the CPU shares using cgroup virtual file system.

The data plane (*libnf*) samples the packet processing time in a lightweight fashion every millisecond by observing the CPU cycle counter before and after the NF's packet handler function is called. We chose sampling because measuring overhead for each packet using the CPU cycle counters results in a CPU pipeline flush [3], resulting in additional overhead. The samples are stored in a histogram, in memory shared between *libnf* and the NF Manager.

The processing time samples produced by each NF are stored in shared memory and aggregated by the NF Manager. Not all packets incur the same processing time, as some might be higher due to I/O activity. Hence, NFVnice uses the median over a 100ms moving window as the estimated packet processing time of the NF. Every millisecond, the NF Manager calculates the load on each NF using its packet arrival rate and the estimated processing time. Every 10ms, it updates the weights used by the kernel scheduler.

4 EVALUATION

4.1 Testbed and Approach

Our experimental testbed has a small number of Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz servers, 157GB memory, running Ubuntu SMP Linux kernel 3.19.0-39-lowlatency. Each CPU has dual-sockets with a total of 56 cores. For these experiments, nodes were connected back-to-back with dual-port 10Gbps DPDK compatible NICs to avoid any switch overheads.

We make use of DPDK based high speed traffic generators, Moongen [12] and Pktgen [38] as well as Iperf3 [11], to generate line rate traffic consisting of UDP and TCP packets with varying numbers of flows. Moongen and Pktgen are configured to generate 64 byte packets at line rate (10Gbps), and vary the number of flows as needed for each experiment.

We demonstrate NFVnice's effectiveness as a user-space solution that influences the NF scheduling decisions of the native Linux kernel scheduling policies, *i.e.,* Round Robin (RR) for the Real-time scheduling class, SCHED_NORMAL (termed NORMAL henceforth) and SCHED_BATCH (termed BATCH) policies in the CFS class. Different NF configurations (compute, I/O) and service chains with varying workloads (traffic characteristics) are used. For all the bar plots, we provide the average, the minimum and maximum values

Table 3: Packet drop rate per second

	NORMAL		BATCH		RR(1ms)		RR(100ms)	
	Default	NFVnice	Default	NFVnice	Default	NFVnice	Default	NFVnice
NF1	3.58M	11.2K	2M	0	0.86M	0	0.53M	0
NF2	2.02M	12.3K	0.9M	11.5K	2.92M	12K	0.03M	12K

Table 4: Scheduling Latency and Runtime of NFs

	NORMAL		BATCH		RR(1ms)		RR(100ms)	
measured in ms	Default	NFVnice	Default	NFVnice	Default	NFVnice	Default	NFVnice
NF1-Avg. Delay	0.002	0.112	0.003	1.613	1.022	0.730	0.924	0.809
NF1-Runtime	657.825	128.723	312.703	143.754	-	-	-	-
NF2-Avg. Delay	0.065	0.008	1.144	0.255	0.570	0.612	0.537	0.473
NF2-Runtime	602.285	848.922	836.940	803.185	-	-	-	-
NF3-Avg. Delay	0.045	0.025	0.149	0.009	0.885	0.479	0.703	0.646
NF3-Runtime	623.797	1014.218	826.203	1047.968	-	-	-	-

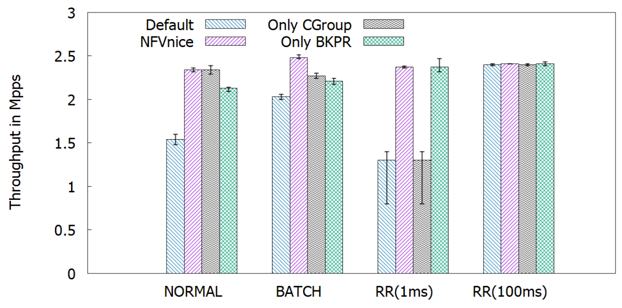


Figure 7: Performance of NFVnice in a service chain of 3 NFs with different computation costs

observed across the samples collected every second during the experiment. In all cases, the NFs are interrupt driven, woken up by NF manager when the packets arrive while NFs voluntarily yield based on NFVnice's policies. Also, when the transmit ring out of an NF is full, that NF suspends processing packets until room is created on the transmit ring.

4.2 Overall NFVnice Performance

We first demonstrate NFVnice's overall performance, both in throughput and in resource (CPU) utilization for each scheduler type. We compare the default schedulers to our complete NFVnice system, or when only including the CPU weight allocation tool (which we term `cgroups`) or the `backpressure` to avoid wasted work at upstream NFs in the service chain.

4.2.1 NF Service Chain on a Single Core: Here, we first consider a service chain of three NFs; with computation cost Low (NF1, 120 cycles), Medium (NF2, 270 cycles), and High (NF3, 550 cycles). All NFs run on a single shared core.

Figure 7 shows that NFVnice achieves an improvement of as much as a factor of two times in throughput (especially over the RR scheduler). We separately show the contribution of the `cgroups` and `backpressure` features. By combining these, NFVnice improves the overall throughput across all three kernel scheduling disciplines. Table 3 shows the number of packets dropped at either of the upstream NFs, NF1 or NF2, after processing (an indication of truly wasted work). Without NFVnice, the default schedulers drop millions of packets per second. But with NFVnice, the packet drop rate is dramatically lower (near zero), an indication of effective avoidance of wasted work and proper CPU allocation.

Table 5: Throughput, CPU utilization and wasted work in chain of 3 NFs on different cores

	Default			NFVnice		
	Svc. rate	Drop rate	CPU Util	Svc. rate	Drop rate	CPU Util
NF1 (~550cycles)	5.95Mpps	-	100%	0.82Mpps	-	11% ±3%
NF2 (~2200cycles)	1.18Mpps	4.76Mpps	100%	0.72Mpps	150Kpps	64% ±1%
NF3 (~4500cycles)	0.6Mpps	0.58Mpps	100%	0.6Mpps	70Kpps	100%
Aggregate	0.6Mpps	-	300%	0.6Mpps	-	175% ±3%

We also gather perf-scheduler statistics for the average scheduling delay and runtime of each of the NFs. From Table 4, we can see that i) with NFVnice the run-time for each NF is apportioned in a cost-proportional manner (NF1 being least and NF3 being most), unlike the NORMAL scheduler that seeks to provide equal allocations independent of the packet processing costs. ii) the average scheduling delay with NFVnice for the NFs (that is the time taken to begin execution once the NF is ready) is lower for the NFs with higher processing time (which is exactly what is desired, to avoid making a complex NF wait to process packets, and thus avoiding unnecessary packet loss). Again this is better than the behaviour of the default NORMAL or RR schedulers [2] .

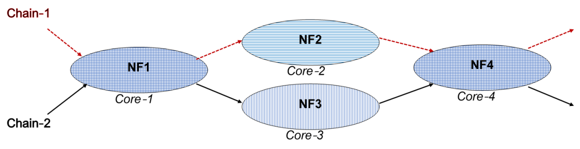


Figure 8: Different NF chains (Chain-1 and Chain-2, of length three), using shared instances for NF1 and NF4.

4.2.2 Multi-core Scalability: We next demonstrate the benefit of NFVnice with the NFs in a chain across cores, with an NF being pinned to a separate, dedicated core for that NF. We use these experiments to demonstrate the benefits of NFVnice, namely: a) avoiding wasted work through backpressure; and b) judicious resource (CPU cycles) utilization through scheduling. When NFs are pinned to separate cores, there is no specific role/contribution for

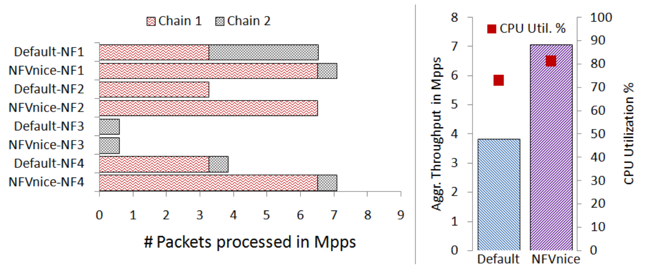


Figure 9: Multi-core chains: Performance of NFVnice for two different service chains of 3 NFs (each NF pinned to a different core), as shown in Fig. 8.

[2] Even though, for this experiment, RR(100ms) performs as well as NFVnice, it performs very poorly with variable per-packet processing costs, as seen in 4.3.1 and for chains with heterogeneous computation costs, as in 4.3.2 scenarios.

Table 6: Throughput, CPU utilization and wasted work in a chain of 3 NFs (each NF pinned to a different core) with different NF computation costs

		Default			NFVNice		
		Svc.Rate (pps)	Drop Rate (pps)	CPU Util.%	Svc.Rate (pps)	Drop Rate (pps)	CPU Util.%
NF1 (~270cycles)	Chain1	3.26M	2.86M	78.6% ±0.4	6.498M	0	82.1% ±0.5
	Chain2	3.26M			0.583M		
	Aggregate	6.522M			7.08M		
NF2 (~120cycles)	Chain1	3.26M	~0	52.8% ±1.2	6.498M	~0	58% ±0.7
	Chain2	-			-		
	Aggregate	3.26M			6.498M		
NF3 (~4500cycles)	Chain1	-	2.68M	100% ±0	-	<100	100% ±0
	Chain2	0.582M			0.582M		
	Aggregate	0.582M			0.582M		
NF4 (~300cycles)	Chain1	3.26M	0	60% ±0.7	6.498M	0	84% ±0.7
	Chain2	0.582M			0.582M		
	Aggregate	3.842M			7.08M		

the vanilla OS schedulers, and for such an experiment we use the default scheduler (NORMAL).

First, we consider the chain of 3 NFs, NF1 (Low, 550 cycles), NF2 (Medium, 2200 cycles) and NF3 (High, 4500 CPU cycles). Compared to the default scheduler (NORMAL), NFVnice plays a key role in avoiding the wasted work and efficiently utilizing CPU cycles. Table 5 shows that NFVnice's CPU utilization by NF1 and NF2 on their cores is dramatically reduced, going down from 100% to $\tilde{1}1$% and 64% respectively, while maintaining the aggregate throughput (0.6 Mpps). This is primarily because of backpressure ensuring that the upstream NFs only process the correct amount of packets that the downstream NFs can consume. Excess packets coming into the chain are dropped at the beginning of the chain. When we use only the default NORMAL scheduler by itself, NF1 and NF2 use 100% of the CPU to process a huge number of packets (the 'service rate' in the Table 5), only to be discarded at the downstream NF3.

We now consider two different service chains using 4 cores in the system. Chain-1 has three NFs: NF1 (270 cycles), NF2 (120 cycles) and NF4 (300 cycles) running on 3 different cores. Chain-2 comprises NF1, NF3(4500 cycles) and NF4. The same instances of NF1 and NF4 are part of both chain-1 and chain-2 as shown in Figure 8. Moongen generates 64-byte packets at line rate, equally splitting them between two flows that are assigned to chain-1 and chain-2. Table 6 shows that in the Default case (NORMAL scheduler), NF1 processes almost an equal number of packets for chain-1 and chain-2. However, for chain-2, the downstream NF3 discards a majority of the packets processed by NF1. This results not only in wasted work, but it also adversely impacts the throughput of chain-1. On the other hand, with NFVnice, backpressure has the upstream NF1 process only the appropriate number of packets of chain-2 (which has its bottleneck at the downstream NF, NF3). This frees up the upstream NF1 to use the remaining processing cycles to process packets from chain-1. NFVnice improves the throughput of chain-1 by factor of 2. At the same time, it maintains the throughput of chain-2 at its bottleneck (NF3) rate of 0.6Mpps. Overall, NFVnice not only avoids wasted work, but judiciously allocates CPU resources (at upstream NFs) proportionate to the chain's bottleneck resource capacity as shown in the Figure 9.

4.3 Salient Features of NFVnice

4.3.1 Variable NF packet processing cost. We now evaluate the resilience of NFVnice to not only heterogeneity across NFs, but also variable packet processing costs within an NF. We use the same three-NF service chain used in 4.2.1, but modify their processing costs. Packets of the same flow have varying processing costs of 120, 270 or 550 cycles at each of the NFs. Packets are classified as having one of these 3 processing costs at each of the NFs, thus yielding 9 different variants for the total processing cost of a packet across the 3 -NF service chain. Figure 10 shows the throughput for different schedulers. With the Default scheduler, the throughput achieved differs considerably compared to the case with fixed per-packet processing costs as seen in Figure 7. For the Default scheduler, the throughput degrades considerably for the vanilla coarse time-slice schedulers (BATCH and RR(100ms)), while the NORMAL and RR(1ms) schedulers achieve relatively higher throughputs. When examining the throughput with only the CPU weight assignment, `CGroup`, we see improvement with the BATCH scheduler, but not as much with the NORMAL scheduler. This is because the variation in per-packet processing cost of NFs result in an inaccurate estimate of the NF's packet-processing cost and thus an inappropriate weight assignment and CPU share allocation. This inaccuracy also causes NFVnice (which combines `CGroup` and backpressure) to experience a marginal degradation in throughput for the different schedulers. Backpressure alone (the Only BKPR case), which does not adjust the CPU shares based on this inaccurate estimate is more resilient to the packet-processing cost variation and achieves the best (and almost the same) throughput across all the schedulers. NFVnice gains this benefit of backpressure, and therefore, in all cases NFVnice's throughput is superior to the vanilla schedulers. We could mitigate the impact of variable packet processing costs by profiling NFs more precisely and frequently, and averaging the processing over a larger window of packets. However, we realize that this can be expensive, consuming considerable CPU cycles itself. This is where NFVnice's use of backpressure helps overcome the penalty from the variability, getting better throughput and reduced packet loss compared to the default schedulers.

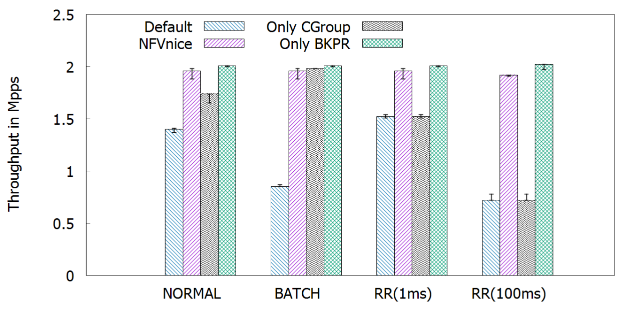


Figure 10: Performance of NFVnice in a service chain of 3 NFs with different computation costs and varying per packet processing costs.

4.3.2 Service Chain Heterogeneity. We next consider a three NF chain, but vary the chain configuration—(Low, Medium, High);(High, Medium, Low); and so on for a total 6 cases—so that the location of the bottleneck NF in the chain changes in each case. Results in Figure 11 show significant variance in the behaviour of the vanilla kernel schedulers. NORMAL and BATCH perform similar to each other in most cases, except for the small differences for the reasons

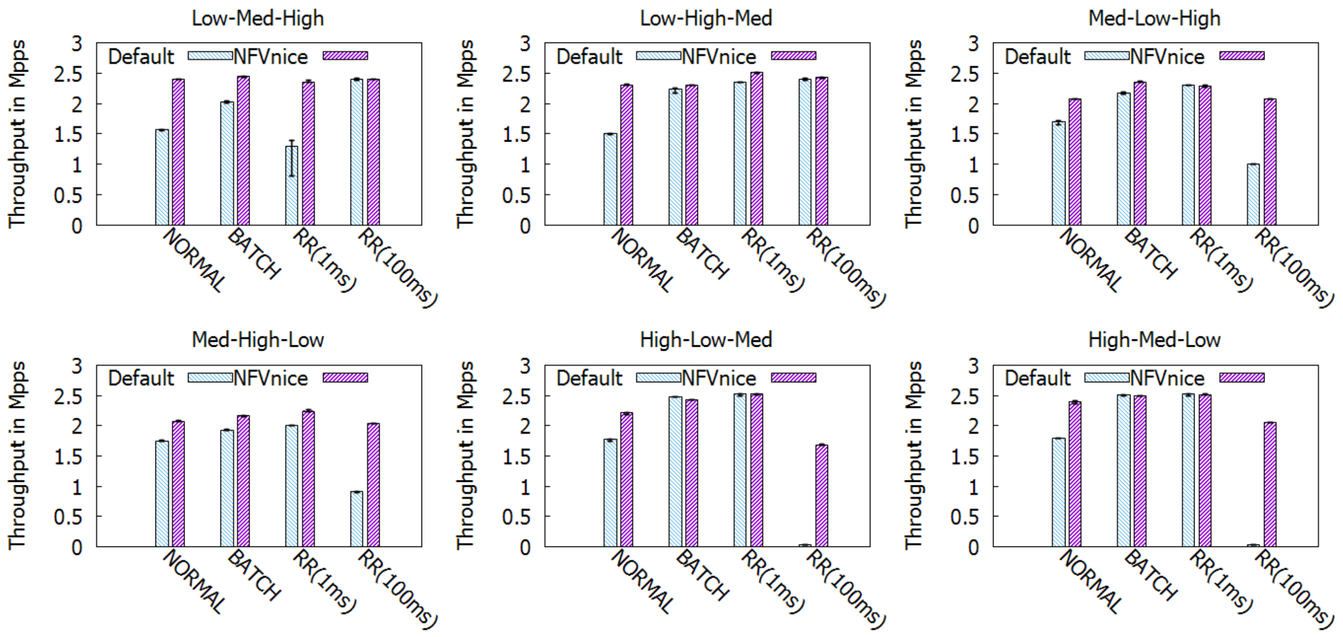


Figure 11: Throughput for varying combinations of 3 NF service chain with Heterogeneous computation costs

described earlier in Section 2. We also looked at RR with time slices of 1ms and 100ms, and their performance is vastly different. For the small time-slice, performance is better when the bottleneck NF is upstream, while RR with a larger time-slice performs better when the bottleneck NF is downstream. This is primarily due to wasted work and inefficient CPU allotment to the contending NFs. However, with NFVnice, in almost every case, we can see considerable improvements in throughput, for all the schedulers. NFVnice minimizes the wasted cycles independent of the OS scheduler's operational time-slice.

Impact of RR's Time Slices with NFV: Consider the chain configurations "High-Med-Low" and "Med-High-Low" in Figure 11. RR(100 ms time slice) performs very poorly, with very low throughput $< 40Kpps$. This is due to the 'Fast-producer, slow-consumer' situation [44], making the NF with "High" computes hog the CPU resource. Now, in the default RR scheduler, the packets processed by this NF would be dequeued by the Tx threads but will be subsequently dropped, as the next NF in the chain does not get an adequate share of the CPU to process these packets. The upstream NF that is hogging the CPU has to finish its time slice and the OS scheduler then causes a involuntary context switch for this "High" NF. However, with NFVnice, the queue buildup results in generating a backpressure signal across the chain, forcing the upstream NF to be evicted (i.e., triggering a voluntary context switch) from the CPU as soon as the downstream NFs buffer levels exceed the high watermark threshold. The upstream NF will not execute till the downstream NF gets to consume and process its receive buffers. Thus, NFVnice is able to enforce judicious access to the CPU among the competing NFs of a service chain. We see in every case in Figure 11, NFVnice's throughput is superior to the vanilla scheduler, emphasizing the point we make in this paper: NFVnice's design

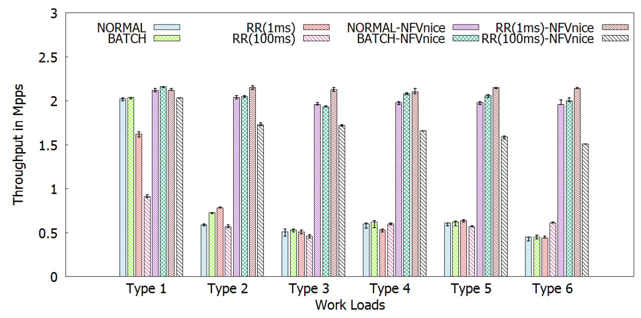


Figure 12: Throughput (Mpps) with varying workload mix, random initial NF for each flow in a 3 NF service chain (homogeneous computation costs)

can support a number of different kernel schedulers, effectively support heterogeneous service chains and still provide superior performance (throughput, packet loss).

4.3.3 *Workload Heterogeneity.*

We next use 3 homogeneous NF's with the same compute cost, but vary the nature of the incoming packet flows so that the three NFs are traversed in a different order for each flow. We increase the number of flows (each with equal rate) arriving from 1 to 6, as we go from Type 1 to Type 6, with each flow going through all 3 NFs in a random order. Thus, the bottleneck for each flow is different. Figure 12, shows that the native schedulers (first four bars) perform poorly, with degraded throughput as soon as we go to two or more flows, because of the different bottleneck NFs. However, NFVnice performs uniformly better in every case, and is almost independent of where the bottlenecks are for the multiple flows. Moreover, NFVnice provides

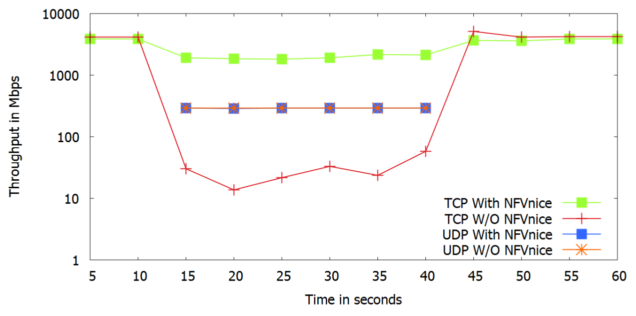


Figure 13: Benefit of Backpressure with mix of responsive and non-responsive flows, 3 NF chain, heterogeneous computation costs

a substantial improvement and robustness to varying loads and bottlenecks even across all the schedulers (NORMAL, BATCH, RR with 1ms or 100 ms slice.)

4.3.4 Performance isolation. It is common to observe that when there are responsive (TCP) flows that share resources with non-responsive (UDP) flows, there can be a substantial degradation of TCP performance, as the congestion avoidance algorithms are triggered causing it to back-off. This impact is exacerbated in a software-based environment because resources are wasted by the non-responsive UDP flows that see a downstream bottleneck, resulting in packets being dropped at that downstream NF. These wasted resources result in less capacity being available for TCP. Because of the per-flow backpressure in NFVnice, we are able to substantially correct this undesirable situation and protect TCP's throughput even in the presence of non-responsive UDP.

In this experiment, we generate TCP and UDP flows with Iperf3. One TCP flow goes through only NF1 (Low cost) and NF2 (Medium cost) on a shared core. 10 UDP flows share NF1 and NF2 with the TCP flow, but also go through an additional NF3 (High cost, on a separate core) which is the bottleneck for the UDP flows - limiting their total rate to 280 Mbps.

We first start the 1 TCP flow. After 15 seconds, 10 UDP flows start, but stop at 40 seconds. As soon as the UDP flows interfere with the TCP flow, there is substantial packet loss without NFVnice, because NF1 and NF2 see contention from a large amount of UDP packets arriving into the system, getting processed and being thrown away at the queue for NF3. The throughput for the TCP flow craters from nearly 4 Gbps to just around 10-30 Mbps (note log scale), while the total UDP rate essentially keeps at the bottleneck NF3's capacity of 280 Mbps. With NFVnice, benefiting from per-flow backpressure, the TCP flow sees much less impact (dropping from 4 Gbps to about 3.3 Gbps), adjusting to utilize the remaining capacity at NF1 and NF2. This is primarily due to NFVnice's ability to perform selective early discard of the UDP packets because of the backpressure. Otherwise we would have wasted CPU cycles at NF1 and NF2, depriving the TCP flow of the CPU. Note that the UDP flows' rate is maintained at the bottleneck rate of 280 Mbps as shown in Figure 13 (UDP lines are one on top of the other). Thus, NFVnice ensures that non-responsive flows (UDP) do not unnecessarily steal the CPU resources from other responsive (TCP) flows in an NFV environment.

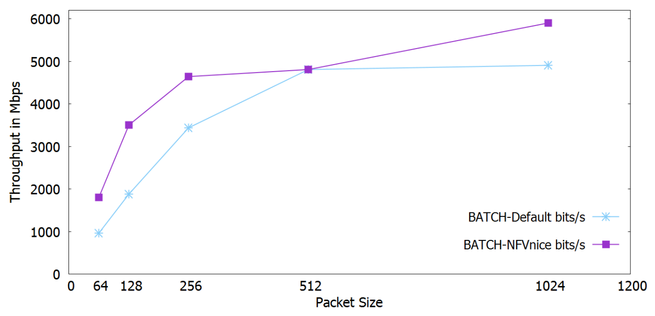


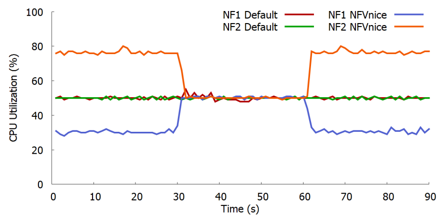
Figure 14: Improvement in Throughput with NFs performing Asynchronous I/O writes withNFVnice

4.3.5 Efficient I/O handling by NFVnice. It is important for NFs to be able to perform I/O required by the packet of a flow, while efficiently continuing to process other flows (e.g., packet monitors, proxies, etc.). Using Moongen we send 2 flows at line rate. Both the flows share the same upstream NFs, but only one of the flows performs I/O *i.e.,* logs the packets to the disk using NFVnice's I/O library. Figure 14 compares the aggregate throughput achieved with and without NFVnice, using the BATCH scheduler in the kernel. We vary the packet size. NFVnice maintains a higher throughput consistently, even for small packet sizes. Moreover, NFVnice maintains progress on the second flow while I/O is being performed for packets of the first flow, thus providing better isolation.

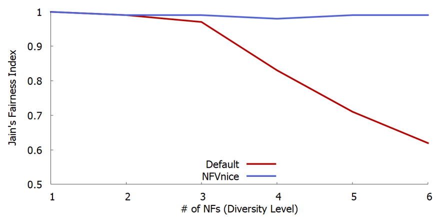
4.3.6 Dynamic CPU Tuning and fairness. Dynamic CPU tuning: NFVnice dynamically adjusts the CPU allocations based on the packet processing cost and arrival rate for each NF. Two NFs initially with different computation costs (ratio 1:3) run on the same core, with MoonGen transmitting a flow each to the two NFs at the same rate. To demonstrate adaptation, we have the computation cost of NF1 temporarily increase 3 times(to the same level as NF2) during the 31 sec. to 60 sec. interval.

Figure 15a has the default NORMAL scheduler evenly allocating the CPU between NF1 and NF2 regardless of their computation cost throughout. On the other hand, NFVnice allocates NF2 three times the CPU as NF1 initially. At t=30s, NFVnice allocates each NF half of the CPU. And at t=60s, we go back to the original allocation. We observed that the throughput for the two flows (not shown) is equal throughout, indicating the capability of NFVnice to dynamically provide a fair allocation of resources factoring in the heterogeneity of the NF CPU compute cost.

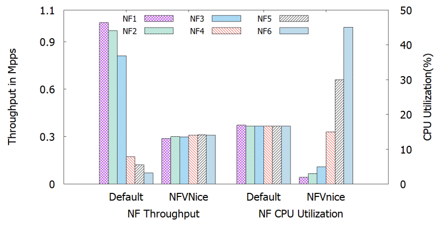
Fairness measure: We evaluate the fairness in throughput as we increase the diversity of computation for each of the NFs for default CFS scheduler and NFVnice. We vary the number of NFs sharing the core. Each NF has the same packet arrival rate, but different computation cost. At diversity level 1, we start with a single flow (uses NF1, compute cost 1). With a diversity level of two, we have 2 flows, flow 1 uses NF1 (compute cost 1), flow 2 uses NF2 (compute cost 2). At a diversity level of 6, there are 6 NFs, with the ratio of computation costs of 1:2:5:20:40:60, and one flow each going to the corresponding NF. At diversity level 6, the NORMAL scheduler allocates 16.6% of the CPU to each of the NFs, being unaware of the computation cost of each NF. Thus, the throughput for flow 1 is 1.02 Mpps, while flow 6 is only 0.07 Mpps. With NFVnice, the CPU



(a) Effect of Dynamic CPU Weight Updates



(b) Measure of Fairness



(c) Effect of rate-cost proportional fairness on CPU Utilization and Throughput

Figure 15: Adaptation to Dynamic Load and Fairness measure of NFVnice compared with the NORMAL scheduler

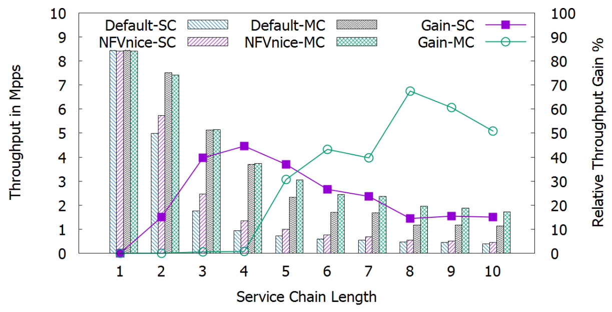


Figure 16: Performance of NFVnice for different NF service chain lengths.

allocated to the lightweight NF is 1%, while the heavyweight NF gets 46%, and all the flows achieve nearly equal throughput 15c). Using Jain's fairness index [24], we show that the vanilla scheduler is dramatically unfair (going down to 0.62) while NFVnice consistently achieves fair throughout (Jain's fairness index of 1.0) as shown in figure 15b).

4.3.7 *Supporting longer NF chains.* We now see how well NFVnice can support longer NF service chains. We choose three different NFs, as in 4.2, and increase the chain length from 1 NF up to a chain of 10 NFs, including one of the 3 NFs each time. We examine two cases: (i) all the NFs of the chain are on a single core (denoted by SC); and (ii) three cores are used, and as the chain length is increased, the additional NF is placed on the next core in round-robin fashion (represented by MC). Results are shown in Figure 16. For the single core, NFVnice achieves higher throughput than the Default scheduler for longer chains, with the greater improvements achieved for chain lengths of 3-6. As the chains get longer (>7 NFs sharing the same core), the improvement with NFVnice is not as high. For the multiple core case, NFVnice improves throughput substantially, especially as more NFs are multiplexed on a care (e.g., chain lengths > 4), compared to the Default scheduler. Of course, the improvement with NFVnice will depend on the type of NFs and their computation costs, for individual use-cases.

4.3.8 *Tuning and Overhead Considerations. Tuning NFVnice:* To tune the key parameters of NFVnice, viz., the HIGH_WATER_MARK and LOW_WATER_MARK, the thresholds for the queue occupancy in the Rx ring, we measure the throughput, wasted work, context-switch overheads and achieved Instructions per Cycle (IPC) count for different configurations. We use a 3 NF, "Low-Med-High" service chain, and use Pktgen to generate line rate minimum packet size traffic. We begin with a fixed 'margin' (difference between the High and Low thresholds). With the margin at 30, we vary the high threshold. Below 70%, the throughput starts to drop (under-utilization), while above 80% the number of packet drops at the upstream NFs increases (insufficient buffering). We then varied the NF service chain length (from 2 to 6), and computation costs (per packet processing cost from 100 cycles to 10000 cycles) to see the impact of setting the water marks. Across all these cases, we observed that a choice of 80% for the HIGH_WATER_MARK worked 'well'. With the high water mark fixed at 80%, we varied the LOW_WATER_MARK, by varying the margin. With a very small margin (1 to 5), packet drops increased, while a margin above 30 degraded throughput. We chose a margin of 20 because it provided the best performance across these experiments. We acknowledge that these watermark levels and thresholds are sensitive to overall path-delay, chain length and processing costs of the NFs in the chain, and that these parameters are necessarily an engineering compromise.

Periodic profiling and CPU weight assignment granularity: We based our frequency of CPU profiling based on the overheads of rdtsc (observed to be roughly 50 clock cycles) and average time to write to the cgroup virtual file system (5 μ seconds). We discard the first 10 samples to effectively account for warming the cache and to eliminate outliers.

5 RELATED WORK

NF Management and Scheduling: In recent years, several NFV platforms have been developed to accelerate packet processing on commodity servers [4, 21, 23, 32, 43]. There is a growing interest in managing and scheduling network functions. Many works address the placement of middleboxes and NFs for performance target or efficient resource usage [16, 25, 30, 39, 41, 46]. For example, E2 [39] builds a scalable scheduling framework on top of BESS [21]. They abstract NF placement as a DAG, dynamically scale and migrate NFs while keeping flow affinity. NFV-RT [30] defines deadlines for requests, and places or migrates NFs to provide timing guarantees. These projects focus on NF management and scheduling across cluster scale. Our work focuses on a different scale: how to schedule NFs on shared cores to achieve fairness when flows have load pressure. Different from traditional packet scheduling for fairness on hardware platforms [18, 47, 49, 50], software-based NFs or middleboxes are more complex, resulting in diversity of packet processing costs. Furthermore, different kinds of flow arrival rates exacerbate the difficulty of fair scheduling.

PSPAT [45] is a recent host-only software packet scheduler. PSPAT aims to provide a scalable scheduler framework by decoupling the packet scheduler algorithm from dispatching packets to the NIC for high performance. NFVnice considers the orthogonal problem of packet processing cost and flow arrival rate to fairly allocate CPU resources across the NFs. PIFO [48] presents the packet-in-first-out philosophy distinct from the typical first-in-first-out packet processing models. We use the insight from this work to decide whether to accept a packet and queue it for processing at the intended NF or discard at the time of packet arrival. Then, the enqueued packets are always processed in order. This approach of selective early discard yields two benefits: i) it avoids dropping partially processed (through the chain) packets, thus not wasting CPU cycles; ii) it avoid CPU stealing and allows CPU cycles to be judiciously allocated to other contending NFs.

User space scheduling and related frameworks: Works, such as [2, 6], consider cooperative user-space scheduling, providing very low cost context switching, that is orders of magnitude faster than regular Pthreads. However, the drawbacks with such a framework are two-fold: a) they invariably require the threads to cooperate, i.e., each thread must voluntarily yield to ensure that the other threads get a chance to share the CPU, without which progress of the threads cannot be guaranteed. This means that the programs that implement L-threads must include frequent rescheduling points for each L-thread [2] incurring additional complexity in developing the NFs. b) As there is no specific scheduling policy (it is just FIFO based), all the L-threads share the same priority, and are backed by the same kernel thread (typically pinned to a single core), and thus lack the ability to perform selective prioritization and the ability to provide QoS differentiation across cooperating threads. Nonetheless, NFVnice's backpressure mechanism can still be effectively employed for such cooperating threads to voluntarily yield the CPU as necessary. Another approach used by systems such as E2 [39] and VPP [4] is to host multiple NFs within a shared address space, allowing them to be executed as function calls in a run to completion manner by one thread. This incurs very low NUMA and cross-core packet chaining overheads, but being monolithic, it is inflexible and impedes the deployment of NFs from third party vendors.

Congestion Control and Backpressure: Congestion control and backpressure have been extensively studied in the past [7, 8, 22, 26, 29, 35]. DCTCP [7] leverages ECN to provide multi-bit feedback to the end hosts. MQ-ECN [8] enables ECN for tradeoff of both high throughput and low latency in multi-service multi-queue production DCNs (Data Center Network). All of these focus on congestion control in DCNs. However, in an NFV environment, flows are typically steered through a service chain. The later congestion is found, the more resources are wasted. If the end hosts do not enable ECN support or there are UDP flows, it is especially important for the NFV platform to gracefully handle high load scenarios in an efficient and fair way. Using multiple mechanisms (ECN and backpressure), NFVnice ensures that overload at bottlenecks are quickly detected in order to avoid congestion and wasted work. *Fair Queueing:* Orthogonal work such as [17, 31], propose to ensure fair sharing of network resources among multiple tenants by spreading requests to multiple processing entities. That is, they distribute flows with different costs to different processing threads. In contrast, NFVnice seeks to achieve fairness by scheduling the NFs that process the packets of different flows appropriately, Thus, a fair share of the CPU is allocated to each competing NF.

6 CONCLUSION

As the use of highly efficient user-space network I/O frameworks such as DPDK becomes more prevalent, there is be a growing need to mediate application-level performance requirements across the user-kernel boundary. OS-based schedulers lack the information needed to provide higher level goals for packet processing, such as rate proportional fairness that needs to account for both NF processing cost and arrival rate. By carefully tuning scheduler weights and applying backpressure to efficiently shed load early in the the NFV service chain, NFVnice provides substantial improvements in throughput and drop rate and dramatically reduces wasted work. This allows the NFV platform to gracefully handle overload scenarios while maintaining efficiency and fairness.

Our implementation of NFVnice demonstrates how an NFV framework can efficiently tune the OS scheduler and harmoniously integrate backpressure to meet its performance goals. Our results show that selective backpressure leads to more efficient allocation of resources for NF service chains within or across cores, and scheduler weights can be used to provide rate proportional fairness, regardless of the scheduler being used.

7 ACKNOWLEDGEMENT

This work was supported by EU FP7 Marie Curie Actions CleanSky ITN project Grant No. 607584, and US NSF grants CNS-1522546 and CNS-1422362. The work was also partially supported by the Department of the Army, US Army Research, Development and Engineering Command grant W911NF-15-1-0508 and Huawei Technologies Co. Ltd.'s HIRP Grant. We also thank our shepherd, Justine Sherry, and the anonymous reviewers for their thoughtful feedback; Victor G. Hill for helping with the setup of our evaluation testbed; Dr. Jiachen Chen and Shriya Kulkarni for their help and ideas on the representation of the graphs and plots.

REFERENCES

[1] 2014. Data plane development kit. http://dpdk.org/. (2014). [ONLINE].
[2] 2014. DPDK L-Thread subsystem. http://dpdk.org/doc/guides/sample_app_ug/performance_thread.html. (2014). [ONLINE].
[3] 2016. Performance measurements with RDTSC. https://www.strchr.com/performance_measurements_with_rdtsc. (June 2016). [ONLINE].
[4] 2016. VPP. https://fd.io/. (2016). [ONLINE].
[5] 2017. cgroups-Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html. (2017). [ONLINE].
[6] 2017. Fibers. https://msdn.microsoft.com/en-us/library/ms682661.aspx. (2017). [ONLINE].
[7] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *ACM SIGCOMM computer communication review*, Vol. 40. ACM, 63–74.
[8] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. 2016. Enabling ECN in Multi-Service Multi-Queue Data Centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 537–549.
[9] Rudolf Bayer. 1972. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica* 1, 4 (1972), 290–306.

[10] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review*, Vol. 19. ACM, 1–12.

[11] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. 2014. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. (2014).

[12] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: a scriptable high-speed packet generator. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*. ACM, 275–287.

[13] ETSI-GS-NFV-002. 2013. Network Functions Virtualization (NFV): Architectural Framework. http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf. (2013). [ONLINE].

[14] Wu-chang Feng, Dilip Kandlur, Debanjan Saha, and Kang Shin. 1999. BLUE: A new class of active queue management algorithms. *Ann Arbor* 1001 (1999), 48105.

[15] Sally Floyd and Van Jacobson. 1993. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.* 1, 4 (Aug. 1993), 397–413.

[16] Aaron Gember, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. 2013. *Stratos: A network-aware orchestration layer for middleboxes in the cloud.* Technical Report.

[17] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. 2012. Multi-resource Fair Queueing for Packet Processing. *SIGCOMM Comput. Commun. Rev.* 42, 4 (Aug. 2012), 1–12. https://doi.org/10.1145/2377677.2377679

[18] Pawan Goyal, Harrick M Vin, and Haichen Chen. 1996. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *ACM SIGCOMM Computer Communication Review*, Vol. 26. ACM, 157–168.

[19] Leo J Guibas and Robert Sedgewick. 1978. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on.* IEEE, 8–21.

[20] J Halpern and C Pignataro. 2015. RFC 7665: Service Function Chaining (SFC) Architecture. https://tools.ietf.org/html/rfc7665. (2015). [ONLINE].

[21] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. *SoftNIC: A Software NIC to Augment Hardware.* Technical Report UCB/EECS-2015-155. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[22] Keqiang He, Eric Rozner, Kanak Agarwal, Yu Jason Gu, Wes Felter, John Carter, and Aditya Akella. 2016. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference.* ACM, 244–257.

[23] J. Hwang, K. K. Ramakrishnan, and T. Wood. 2015. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. *IEEE Transactions on Network and Service Management* 12, 1 (March 2015), 34–47. https://doi.org/10.1109/TNSM.2015.2401568

[24] Raj Jain, Dah-Ming Chiu, and William R Hawe. 1984. *A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer System.* Vol. 38. Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA.

[25] Dilip A Joseph, Arsalan Tavakoli, and Ion Stoica. 2008. A policy-aware switching layer for data centers. In *ACM SIGCOMM Computer Communication Review*, Vol. 38. ACM, 51–62.

[26] Glenn Judd. 2015. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter.. In *NSDI*. 145–157.

[27] Tom Kelly, Sally Floyd, and Scott Shenker. 2003. Patterns of congestion collapse. *International Computer Science Institute, and University of Cambridge* (2003).

[28] D. Lapsley and S. Low. 1999. Random early marking: an optimisation approach to Internet congestion control. In *Networks, 1999. (ICON '99) Proceedings. IEEE International Conference on.* 67–74. https://doi.org/10.1109/ICON.1999.796161

[29] Changhyun Lee, Chunjong Park, Keon Jang, Sue B Moon, and Dongsu Han. 2015. Accurate Latency-based Congestion Feedback for Datacenters.. In *USENIX Annual Technical Conference*. 403–415.

[30] Yang Li, Linh Thi Xuan Phan, and Boon Thau Loo. 2016. Network functions virtualization with soft real-time guarantees. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on.* IEEE, 1–9.

[31] Jonathan Mace, Peter Bodik, Madanlal Musuvathi, Rodrigo Fonseca, and Krishnan Varadarajan. 2016. 2DFQ: Two-Dimensional Fair Queuing for Multi-Tenant Cloud Services. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16).* ACM, New York, NY, USA, 144–159. https://doi.org/10.1145/2934872.2934878

[32] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14).* USENIX Association, Seattle, WA, 459–473. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins

[33] Paul Menage. 2017. Linux Kernel Documentation: CGROUPS. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. (2017). [ONLINE].

[34] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014).

[35] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 537–550.

[36] Jeffrey C Mogul and KK Ramakrishnan. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems* 15, 3 (1997), 217–252.

[37] Ingo Molnar. 2017. Linux Kernel Documentation: CFS Scheduler Design. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt. (2017).

[38] Robert Olsson. 2005. Pktgen the linux packet generator. In *Proceedings of the Linux Symposium, Ottawa, Canada*, Vol. 2. 11–24.

[39] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15).* ACM, New York, NY, USA, 121–136. https://doi.org/10.1145/2815400.2815423

[40] Abhay K Parekh and Robert G Gallagher. 1994. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case. *IEEE/ACM Transactions on Networking (ToN)* 2, 2 (1994), 137–150.

[41] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13).* USENIX, 227–240.

[42] K.K. Ramakrishnan, S. Floyd, and D. Black. 2001. RFC 3168: The Addition of Explicit Congestion Notification (ECN) to IP. https://tools.ietf.org/html/rfc3168. (2001). [ONLINE].

[43] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference*. USENIX, Berkeley, CA, 101–112. https://www.usenix.org/conference/usenixfederatedconferencesweek/netmap-novel-framework-fast-packet-io

[44] Luigi Rizzo, Stefano Garzarella, Giuseppe Lettieri, and Vincenzo Maffione. 2016. A Study of Speed Mismatches Between Communicating Virtual Machines. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems (ANCS '16).* ACM, New York, NY, USA, 61–67. https://doi.org/10.1145/2881025.2881037

[45] Luigi Rizzo, Paolo Valente, Giuseppe Lettieri, and Vincenzo Maffione. 2016. PSPAT: software packet scheduling at hardware speed. http://info.iet.unipi.it/~luigi/papers/20160921-pspat.pdf. (2016). [ONLINE].

[46] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12).* USENIX Association, Berkeley, CA, USA, 24–24.

[47] Madhavapeddi Shreedhar and George Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking* 4, 3 (1996), 375–385.

[48] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference.* ACM, 44–57.

[49] Dimitrios Stiliadis and Anujan Varma. 1998. Efficient fair queueing algorithms for packet-switched networks. *IEEE/ACM Transactions on Networking (ToN)* 6, 2 (1998), 175–185.

[50] Dimitrios Stiliadis and Anujan Varma. 1998. Rate-proportional servers: a design methodology for fair queueing algorithms. *IEEE/ACM Transactions on networking* 6, 2 (1998), 164–174.

[51] Ion Stoica, Scott Shenker, and Hui Zhang. 2003. Core-stateless Fair Queueing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High-speed Networks. *IEEE/ACM Trans. Netw.* 11, 1 (Feb. 2003), 33–46. https://doi.org/10.1109/TNET.2002.808414

[52] Rahul Upadhyaya, CB Anantha Padmanabhan, Meenakshi Sundaram Lakshmanan, and Satya Routray. 2016. Optimising NFV Service Chains on OpenStack Using Docker. https://www.openstack.org/videos/video/optimising-nfv-service-chains-on-openstack-using-docker. (April 2016).

[53] Lixia Zhang. 1991. VirtualClock: a new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems (TOCS)* 9, 2 (1991), 101–124.

[54] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMIddlebox '16).* ACM, New York, NY, USA, 26–31. https://doi.org/2940147.2940155