NFV Performance Profiling on Multi-core Servers

Peng Zheng[‡*], Wendi Feng[§*], Arvind Narayanan[*], and Zhi-Li Zhang[*]
[*]University of Minnesota, USA [‡]Xi'an Jiaotong University, China [§]BUPT, China
Email: {pzheng, wfeng, naray111}@umn.edu, zhzhang@cs.umn.edu

***Abstract*—Network Function Virtualization (NFV) lays the foundation for future networking. In this paper we attempt to provide an in-depth analysis of NFV performance, namely, how many packets a single CPU core can process, and in particular, whether and when packet processing performance will *scale linearly* with the number of cores. Understanding these questions is important given the limited server capacities in a (mobile) edge cloud. Through careful and extensive performance measurement, we demonstrate server microprocessor architecture has an enormous impact on NFV performance, and interplay among the NF state, operations and workload characteristics in a service function chain (SFC) further compounds the problem. We develop a systematic profiling model for benchmarking and estimating NFV/SFC performance under diverse traffic demands.**

I. Introduction

By implementing and executing network functions (NFs) as software running on multi-core commodity servers, network function virtualization (NFV) makes networks more programmable, flexible, and scalable. Network functionality can also be continually evolved, new services quickly rolled out, and network capacity readily expanded by adding new servers. NFV is therefore touted as one of the primary pillars on which future networks will be built. A basic premise of NFV is to dynamically *scale out/in* by allocating more/fewer CPU cores to execute NF instances to meet traffic demands.

To effectively utilize the limited server capacity in a (mobile) network edge cloud (NEC) site while providing line-rate packet processing, a fundamental yet important problem for operators is to understand the NFV *scaling performance* on multi-core servers *under diverse workloads*. Namely, *how many packets can a single CPU core process for a given workload? When scale-out, how many packets can a k-core server process in total?* In particular, *under what conditions does the packet processing performance scale linearly with the number of cores?*

These questions are far more complex than they appear. Taking a network monitor (NM) function (see §II-B for more detail) as an example, the performance of a single core (on a 48-core server) is shown in Fig. 1 using a real packet trace from [1]. We can see the performance varies drastically: from as high as 5.5 Mpps (million packets per second) at some times to only 3.9 Mpps. What contributes to such wild fluctuations in NFV packet processing performance? We will see in Fig. 7 later that when scaling out NM by running multiple instances on multiple cores, the performance often *does not increase linearly* with the number of cores; worst, adding more cores sometimes even hurts the overall system throughput!

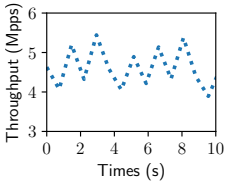


Figure 1: NM performance example

The goal of this paper is to provide deeper insights into these performance issues. We first present a conceptual framework to characterize and understand the behavior of individual NFs in terms of their operations, state and traffic workloads, and further extend it to multiple NFs executed in a sequence, namely, a *service function chain* (SFC). In particular, we examine the interplay between the NF state, operations and the workloads and define several *induced* traffic characteristics and metrics, e.g., *affinity*, *spatial diversity* and *temporal locality*. Guided by this framework, we develop a *systematic* profiling methodology to conduct extensive micro-benchmarking and dive deep into the multi-core microprocessor architecture to study their impact on NF/SFC performance on a single core as well as NF/SFC scaling performance on multiple cores. The main findings and contributions are summarized below.

- We show that multi-core server microprocessor architecture, especially cache memory hierarchy, has a huge impact on NF performance. Apart from the complexity of NF operations on packets, the NF state is the key factor determining the number of packets can be processed per core: As the state size grows beyond the L1/L2 caches, NF performance degrades quickly, as much as less than half of the peak performance. This explains the wild fluctuations under varying workload observed in Fig. 1.
- When scale-out on multiple cores, the *traffic dispatching* policy, namely, the strategy & rules used to split and balance, and distribute the workload among the instances is another key performance factor. Again the interplay between the NF state and the workload characteristics defined with respect to the induced (traffic) affinity plays a critical role. We find that NF/SFC performance scales *linearly* with the number of cores only when the workload can be (near-evenly) partitioned and balanced among the instances *and* the (active) state of each NF instance resides within its (*core-dedicated*) L1/L2 caches. Under "non-uniform" workload or when instances have *shared* state, scaling performance deteriorates significantly; using locks to protect shared state can negate the benefit of scale-out entirely. This is largely because the cores compete for the *shared* L3 cache and DRAM resources.
- SFC further compounds the situation, as multiple NFs in an SFC may have competing and conflicting state requirements

ISBN 978-3-903176-28-7

and induce different traffic affinities. Hence what is "uniform" workload to one NF is "non-uniform" to another, making the choice of the *right* traffic dispatching policy more difficult. Nonetheless, our conceptual framework enables to define useful (NF-specific) workload characteristics and reason about the competing resource requirements of NFs in an SFC.

• Last but not the least, using the systematic profiling methodology to benchmark NF/SFC performance using *select* synthetic workloads, we demonstrate how to estimate and bound NF/SFC performance on diverse workloads via a simple and yet effective method.

While there have been a flurry of research on developing various NFV platforms as well as modeling, analysis and verification of NF/SFC behavior, *e.g.*, via program analysis and symbolic execution, we believe, to the best of our knowledge, this is the first systematic look at the impact of server microprocessor architecture on NFV scaling performance via microbenchmarking and in-depth cache-level measurements.

II. Multi-core server and SFC execution

As a target NFV execution environment in NEC is multi-core servers, we provide a quick overview of a typical multi-core server architecture and its NUMA memory hierarchy. We then briefly present some background on NFV and SFC, with a simple example SFC used in this paper.

A. Multi-core NUMA Architecture

Fig. 2 schematically depicts a typical (Intel Skylake) multi-core server architecture with its NUMA memory hierarchy. This server has two CPU sockets, each with 24 cores. Each core has its own dedicated L1/L2 caches, and the cores within a socket have a shared L3 cache (also referred to as the last level cache, LLC) and other *uncore* resources such as integrated memory controller (iMC, to which the main memory bank DDR4 is attached) and DDIO. Intel DDIO allows direct transfer data between NICs and LLCs, minimizing main memory accesses. This is utilized by Intel DPDK [2] to enable high performance packet processing. While L1 and L2 cache access latencies are ˜1.2 ns and ˜4.1 ns respectively, we note that an NF running on a core takes 13-20 ns to access data stored in the shared local L3 cache in the same socket; this latency further increases to 29-44 ns when accessing (via UPI bus) data stored in the non-local L3 cache on the other socket. In contrast, the typical memory access latency to the local DRAM is 70 ns, and the non-local DRAM is 125 ns.

The figure also suggests that the multi-core server architecture – especially, memory access latencies – will have a significant impact on NFV performance. For example, to keep up with the line rate of a 10Gbps NIC, the average per-packet processing time is ˜67ns for minimal 64 bytes packet size, roughly the same as one local DRAM memory access latency; with a 40Gbps NIC, this reduces to ˜18ns, within the range of one L3 cache access. Keeping up with a 100Gbps line rate requires most data accesses confined to the core-dedicated L1/L2 caches. Unfortunately, the L1/L2 cache sizes are limited. As where data is stored in the memory hierarchy has a significant impact on the performance of an

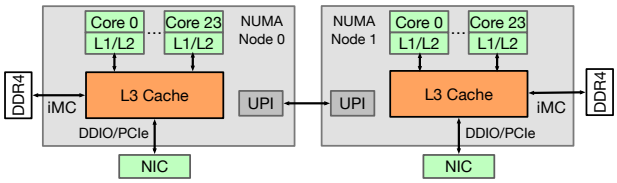


Figure 2: Multi-core NUMA Server Architecture

NF, profiling NF performance is a nontrivial task. In this paper, we provide an in-depth examination of NFV performance, with the goal to develop a systematic framework for profiling and benchmarking NFV/SFC performance.

B. NFV, SFC Execution and Scaling

As stated in the introduction, NFs are "reactive" programs that perform operations on packets as they arrive. The operations of an NF on packets are determined by both its functionality and (control) state. An NF is *stateless* if it simply reads its state to determine what to do with a packet, and the operation on a future packet is independent of those arriving before it. A simple example is a layer-3 forwarder (L3FW) NF, which reads its routing/forwarding table to determine where to route a packet. In contrast, a *stateful* NF must also *update* its state as a result of a decision/operation made on a packet, which will subsequently influence the decision/operation made on some future packets. A classical example is a layer-4 load balancer (L4LB) NF which, upon the arrival of the first packet of a TPC 5-tuple flow, selects a (backend) server to which it will be assigned for processing and rewrites the destination IP address of the packet, which is then forwarded to the selected server – this decision also results in a new flow entry inserted into a flow table (the state) that L4LB must maintain; the destination IP addresses of the subsequent packets belonging to the same flow must be rewritten with the same server address so that they can be forwarded to the correct server for processing. NFs are often strung together to form an SFC, e.g., L4LB followed by L3FW. Hence given a packet, often multiple operations from a sequence of NFs must be performed before it is shipped out of an NFV system.

For simplicity of exposition, we consider the common *run-to-completion* (RTC) model for SFC execution, namely, all NFs in an SFC instance are executed on a single core. It has been shown [3], [4], [5] that RTC in general yields superior performance over the alternative pipeline model [6], [7], [8] (albeit not always the case [5]). This is because RTC avoids the inter-core transfer penalty when packets are moved from one core to another core for processing, the latency of which is at least as much as that of one local L3/LLC access. RTC also simplifies the problem of SFC scaling – we can simply scale out SFC by running multiple SFC instances, one per core. Hence the basic NFV profiling problem reduces to: i) how much traffic a single SFC instance can process on a single core; and then ii) the scaling performance on multiple cores. In this paper we will *empirically* demonstrate that answering each of these two questions are *not* as straightforward as they may appear. There are many complex factors affecting the performance of NFs and SFCs, and several challenges must be addressed when scaling SFC.

To make the exposition both lucid and concrete, we will use a "toy" SFC with four simple NFs, namely, ACL → NM → L4LB → L3FW , as running examples throughout the paper. In this SFC, two NFs (ACL and L3FW) are stateless and the other two (NM and L4LB) are stateful. The operations and the state they maintain are briefly described below: (i) Access control (ACL) performs flow classification operation using a set of configured access control rules to decide whether to block a flow. (ii) Network monitor (NM) maintains a host counter array, one counter per host (src_ip), to keep track of the number of packets generated by each host (e.g., for accounting); for fast look up, we implement the host counter array using a hash table. (iii) Layer-4 load balancer (L4LB), as mentioned earlier, assigns a flow id to each flow using a hash function performed on the packet's 5-tuple header fields whose information is stored in the flow table, looks up a flow-to-server table using the flow id to map each flow to one of the several destination servers by rewriting the dst_ip and dst_port fields of the packet. (iv) Layer 3 forwarder (L3FW), also mentioned earlier, performs exact match by looking up a routing table using a hash of the destination IP address to find the output interface and rewrite the packet's dst_mac field. We have implemented these simple yet highly optimized NFs using Intel DPDK libraries 18.11. In particular, the access control function is implemented using the built-in DPDK ACL library [9]. All the look up tables are implemented with hash library [10], where the hash function is CRC-based.

III. Characterizing Network Function Behavior: Key Factors and Performance Metrics

We present a conceptual framework for characterizing NF behavior along three dimensions: NF *operations*, *state* and *workload* (i.e., network traffic). As part of this framework, we also identify the major factors that affect NF performance and put forth several performance metrics.

A. NF Operations

The program logic of an NF dictates what operations it must perform. These operations can be broadly separated into two kinds: i) operations on packets and ii) operations on the state; both kinds involve memory accesses. A typical NF will perform at least one read and one write operations on packets; at least one read to the state (to look up the state) and may be also a write operation to the state (for a stateful NF). For example, the simple stateless NF, L3FW, reads the packet header, uses it to look up its forwarding table, and then rewrites the dst_mac field for packet forwarding. In contrast, the stateful L4LB NF performs multiple read/write operations both on packets and the state: for each packet, it reads the 5-tuple header fields and maps it to a unique flow id, and uses the flow id to look up its flow table; if the lookup is successful, it uses the returned value to rewrite the dst_ip field of the packet; a failed lookup operation indicates the packet belonging to a new flow, which results in additional operations – L4LB looks up the server pool table to select a server for the new flow, rewrites the dst_ip field of the packet accordingly, and inserts a new entry in the flow table.

Table I: The State and Operations of 2 Stateful NFs

NFs	Scope of NF State (and induced TAG)	# of state operations	# of packet operations	# of (static) instructions
NM	per src_ip	1 read, 1 write	1 read	461
L4LB	per 5-tuple	2-3 read, 0-2 write	1 read, 1 write	676

One way to quantify the complexity of an NF is to calculate the number of instructions (e.g., using the tool developed in [11]) and measure the cycles needed to execute them. The number of instructions clearly affects the instruction cache (i-cache) requirement, nonetheless it is a less performance *critical* indicator. We also remark that modern servers have separate i-caches and d-caches (data caches), hence NF instructions do not compete for the cache resources as in the case of NF state and packets (see more discussion below). Instead, it is far more important to quantify the number of NF memory accesses, i.e., the number of operations on packets and state, due to the NUMA memory hierarchy and vastly differing cache/memory access latencies. In the last three columns of Table I, we list and compare the numbers of state and packet operations and the size of instructions for NM and L4LB. We see that L4LB is relatively more complex than NM – this is also reflected by the performance results in §IV-B.

B. NF State

Whether stateless or stateful, NF operations are determined by its *state*. The size of NF state is a key factor affecting the performance of an NF, as it determines where in the memory hierarchy NF state entries may be cached or stored. In particular, the state of a stateful NF has far significant impact on NF performance. Each NF state has a *scope* and induces *traffic affinity*: it binds a group of packets together to the same entry of the NF state – namely, the processing of these packets triggers access/update to the same state entry – referred to as the *traffic affinity group* (TAG). The scope of NF state determines the (potential) size of TAGs. By default, the scope of a stateless NF is per-packet, and thus the size of each TAG is 1. On the other hand, scope of the L4LB state is per (5-tuple) flow, and hence its TAGs correspond to packets belonging to individual flows. In contrast, the scope of the NM state is per host (per source IP address), and hence its TAG sizes vary depending on how many packets/flows are generated by each host. The scope of (*stateful*) NF state and its granularity have a significant influence on the size of memory footprints of an NF at various levels (L1/L2/L3) of the cache hierarchy. For example, as its state is more fine-grained than that of NM, L4LB may require much more cache resources to maintain its state (e.g., its flow table) than NM to process the same collection of packets. When the flow table size exceeds the L1/L2 cache sizes, part of the state must be stored in the L3 cache or even DRAM, which are not only slower than the L1/L2 caches, but also shared by the cores on the same CPU socket. This creates resource contention among them.

NF state, its scope and the induced TAGs not only affect the performance of an NF running on a single core, but also have important implications when scaling NF to multiple cores. A key issue here is to how to *distribute* (load-balance) and *steer* traffic among different instances of the same NF – this problem is simply referred to as *traffic dispatching*. For example, when

scaling NM to multiple instances, is it better to i) distribute traffic on a *per-flow* basis and steer the flows randomly to each NM instance as most existing NFV frameworks using RTC do [4], [6], or ii) distribute traffic on a *per-host* basis and steer all packets with the same source IP address to the same instance (i.e., observing the TAG boundary)? The former strategy allows the workload to be more evenly balanced among the NF instances, but multiple cores now need to access and update the same state entry (a host counter). In the other words, the NM state is *shared* among the cores and needs to be resident in the L3 cache, creating resource contention and slower state updates. In contrast, the latter strategy enables the cores to effectively "divide" the (host) counter array for exclusive state access. But the workload may not be evenly balanced among the instances (especially when there are some "elephant" hosts generating large amounts of traffic), and thus the cores are not equally utilized. As will be discussed later, this situation can become far more complex when considering scaling out an SFC with multiple NFs, which calls for a systematic methodology for SFC performance profiling.

C. Workload Characteristics

Clearly, the performance of an NF hinges on its workload, i.e., incoming traffic. As discussed above, the state scope of a stateful NF induces traffic affinity among packets. Given a collection of packets, we use the term *traffic diversity* to measure the number of state entries, one per TAG, that the NF must maintain. This yields the total state memory requirement for processing the given traffic demand. However, the actual cache requirements will generally be much smaller than the total state size, and will depend on the cache/memory hierarchy footprints of the NF as it processes packets dynamically as they arrive. Measuring the cache requirements/memory hierarchy footprints directly is challenging in practice. We estimate them indirectly by using two metrics: i) the *temporal locality* of the traffic workload characterizes how often packets belonging to the same TAG arrive within a time window (see §V for a formal metric to measure temporal locality); and ii) the *spatial locality* measures how many different TAGs packets arriving within a time window belong to. These metrics depend on the measurement time window size and vary over time, and can be quantified using the average, worst-case or percentile statistics or modeled as a probability distribution. Last but not the least, DPDK often transfers a burst of packets directly from NICs to the LLC/L3 cache. The packets under processing by an NF also compete with the NF state for the d-cache resources.

IV. Performance Profiling of Individual NFs

We conduct experiments to empirically benchmark NF performance. We start with a single core and then investigate the issues in scaling NF performance using multiple cores.

A. Testbed and Workload Generation

Before the profiling and experimental results, we first provide a brief description of the testbed and how we generate the workload by controlling and varying various characteristics.

Testbed. Our testbed consists of two 48-core servers, each equipped with a dual-port 100Gbps DPDK-capable NIC. The servers are directly connected via optical links of 100 Gbps line rate. Both servers have two sockets (24 cores/socket), with the same architecture as shown in Fig. 2. CPUs are Intel Xeon Platinum 8168 @2.7GHz clocked at 3.4GHz. One server is used to generate packets and send them to the other server for NF/SFC processing, which sends them back after being processed. The OS on the server running our DPDK-based NFV framework is Ubuntu 18.04.2. For both the servers, we modify the kernel parameters to isolate the CPU cores such that the Linux kernel does not use those cores for scheduling. We however leave just one core (core 0) for the OS. Hyper-threading is disabled. We allocate 256 hugepages of 1GB size for packet processing, thus each of the two CPU sockets get 128 GB of hugepage memory. Due to space limitation, we will not delve into the design of our NFV framework. To dispatch traffic to different cores, we use DPDK's `rte_flow` [12] library to offload the specified traffic dispatching policies (based on 5-tuple headers) into the hardware NIC.

Workload Generation. We use *TRex* [13] – one of the few software-based traffic generators capable of generating 100Gbps traffic (TRex can generate traffic at 100Gbps line rate consistently using 20 cores). For compatibility, the traffic generator is installed on the server with CentOS 7.6 distribution. TRex allows us to generate traffic with diverse characteristics with *controllable* rules ("traffic profiles"). For example, we can fix the total volume of traffic (e.g., #. of packets with a given packet size), and vary the number of (5-tuple) flows, or more generally, TAGs. Take the host-level TAGs (of NM) as an example. We can generate a collection of target TAGs by setting the source IP range to be `10.0.0.1 - 10.0.0.10`, and randomly produce a given number of 5-tuple flows (or packets) belonging to these TAGs. This would allow us to control the total number of state entries in an NF by generating a certain number of TAGs with maximal traffic diversity (and worst temporal/spatial locality properties). TRex provides APIs for us to control and vary the temporal characteristics of the workload, e.g., by changing the inter packet gap (thus the # of packets per second), the start time (in terms of flows or packets in the same workload), or the number of packets in each flow. TRex can also record the generated traffic in a pcap trace, and replay a pcap file (e.g., from previously generated or *real* traffic trace) for traffic generation. Given a workload, we can modify the traffic temporal locality characteristics by reordering flows/TAGs, e.g., by replaying flows $f_1 \rightarrow f_2 \rightarrow \ldots \rightarrow f_n$, one by one in a loop (to maximize temporal locality), or by varying the sets of flows in each time interval (to control spatial locality).

B. NF Performance Benchmarking on a Single Core

We now examine the impact of NF operations, state and workload characteristics on the performance of an NF running on a single core.

Impact of NF Operations. We first determine how many packets an NF can process on a single core under the *best*

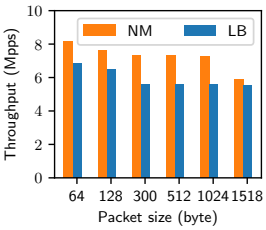


Figure 3: NFs performance with packet size

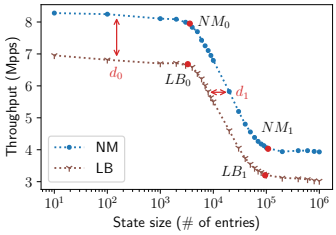
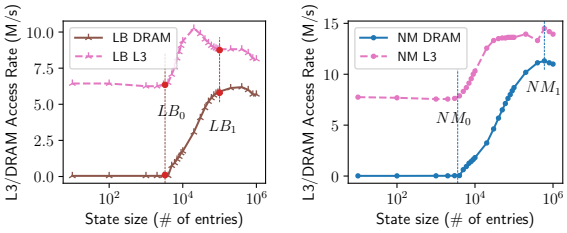


Figure 4: NFs performance with different state size



(a) L4LB (b) NM

Figure 5: NF performance vs. L3/DRAM memory access rates

scenario by increasing the number of packets while fixing the numbers of flows/TAGs to ensure that the NF state resides within the L1 cache. Fig. 3 shows the maximum number of packets that `L4LB` and `NM` running on a single core can process under different packet sizes. With 64 bytes packet size, `L4LB` running on a single core can handle a maximum of 6.9 Mpps, while `NM` running on a single core can handle a maximum of 8.2 Mpps. The performance degrades slightly with larger packet sizes, as more L3 cache space is needed for holding them. That `L4LB` can handle fewer Mpps than `NM` is not too surprising. (We remark that with a smaller packet size, an NFV system must be able to handle more packets so as to keep up with the 100Gbps line rate. In the following experiments, unless otherwise stated, the default packet size is 64 bytes). As shown in Table I, `L4LB` is slightly more complex, requiring more memory access operations per packet. The L1 i-cache is 16KB in our server. As the average instruction length is 64 bits, roughly 2000 instructions can be packed into the cache. Hence the i-cache typically will not be the bottleneck for NFs (and SFCs) that contain fewer than 2000 instructions. Instead, the number of *memory-bound* operations in an NF is a key performance critical factor. This will be more evident when we examine the impacts of NF state and workload characteristics below.

Impact of NF State. In this set of experiments, we investigate the impact of the NF state size by varying the number of flows (`L4LB`) or the number of (source) hosts (`NM`) while fixing the total traffic volume. The results are shown in Fig. 4. For either NF, as we gradually increase the state size starting with 10 entries, the performance stays nearly constant until the state size reaches a critical point (LB_0 and NM_0 resp. in the figure) – dubbed the *best performance point* (BPP) – and starts dropping rapidly once beyond this point. As the state size continues to increase beyond another critical point (LB_1 and NM_1 resp. in the figure) – dubbed the *worst performance point* (WPP), the performance stabilizes and stays largely constant again. These phenomena can be explained by examining the memory accesses of the NFs, as shown in Fig. 5: before BPP, the memory accesses are mostly confined to the L1/L2 caches, yielding the best performances; between BPP and WPP, we see that memory accesses are increasingly bound to the L3 cache and DRAM; beyond WPP, the rate of DRAM bound memory accuses largely stabilizes for both NFs. The performance gaps between the two NFs (depicted by d_0, d_1) are due to the fact that `L4LB` requires more memory per state entry and more memory operations per packet. We see that performance of NFs is primarily determined by memory access speeds. This is true not only for `L4LB` and `NM`, but for a variety of other NFs we have tested (e.g., stateful NAT, firewall, IDS). This is because NFs typically perform relatively simple computations on packets (and state) and then ship them out – this is in contrast to many cloud computing applications which are often compute-intensive.

Impact of Workload Characteristics. Clearly, *traffic diversity* (as measured by the number of TAGs in a given traffic demand) is intimately related to the NF state, in particular, its scope – more diverse traffic leads to larger state size. In the above experiments, packets/flows belonging to different TAGs are randomly generated; hence the workload thus generated likely has the worst-case temporal and spatial localities. We now explore the impact of the temporal locality characteristics in traffic workload on the performance of an NF by comparing the "best-case" vs. "worst-case" scenarios. For the "best-case" scenario, we fix the number of flows/TAGs and generate them one at a time in a loop (instead of randomly as before): namely, we generate packets belonging to flow 1/TAG 1 first (either with a fixed flow/TAG size, or the size randomly drawn from a distribution), we then move on to generate packets belonging to flow 2/TAG 2, and so forth; this process is repeated until a target traffic volume is reached. Fig. 6 compares the `L4LB` performance under the worst-case and best-case temporal localities as we vary the traffic diversity (i.e., the number of flows). We see that temporal localities in traffic workload help improve the NF performance, delaying the performance turning points (BPPs and WPPs). The same observation also holds for `NM`. By controlling the spatial localities in traffic workloads, we obtain similar results. Due to space limitation, we do not report them here. In §V we will explore these issues further – in particular, *the interplay of NF state and workload characteristics* – in the context of profiling SFC performance.

C. Scaling Out NF Performance

When the traffic demand exceeds what an NF running on a single core can process, scaling out the NF by running on multiple cores is necessary. Given the results obtained from NF performance benchmarking on a single core, would the overall throughput simply scale linearly with the number of cores, c? Unfortunately such *linear scaling* performance can be attained only for certain types of NFs, e.g., `L4LB`, with *per-flow* (5-tuple flow) TAG. As discussed in §III-B, when the TAG granularity of a *stateful* NF is coarser than a flow, a new question arises: *how traffic should be distributed and steered to multiple instances of an NF?* The answer to this question matters because whenever a TAG is split among multiple cores, these cores will need to access to and *update* the same state

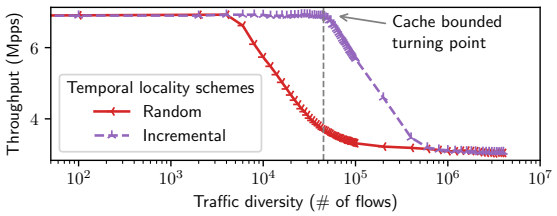


Figure 6: L4LB performance under the worst case and best case temporal localities

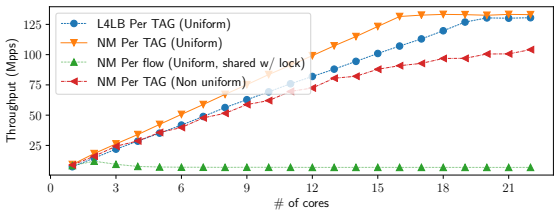


Figure 7: The scaling performance of NM vs L4LB

entry of the NF. In other words, the state must be *shared* among multiple cores, and therefore the state entries are at best residents in the L3 cache and worst in the DRAM. If strong consistency of the NF state is required, locking mechanism must be employed [14], [15], [16]. This further slows down the operations of an NF. Hence the strategy used for traffic dispatching to scale out an NF (and likewise, an SFC) has a huge impact on its performance.

We use NM as an example to empirically demonstrate the impact of traffic dispatching strategies and *shared* state on NFV performance scaling. We generate two traffic workloads with different characteristics, both with a total of 1000 (src_ip) TAGs. The first is the "uniform" workload with each TAG containing an equally number of (5-tuple) flows; the src_ip of these flows is randomly generated from one of the 1000 TAGs. The other workload is "non-uniform" with half of the TAGs ("mice hosts") containing equal number of flows, while the remaining TAGs ("elephant" hosts) containing 10 times as many flows as in the "mice" TAGs. Each (5-tuple) flow has the same number of packets with the frame size of 64 bytes. The number of cores used in the experiments ranges from 1 to 22, where in each experiment, we gradually increase the traffic volume until the system is saturated to the best scaling performance under each setting. In conducting these experiments, we employ Intel Cache Allocation Technology [17] to provision an equal or proportional fare of the L3 cache per core to avoid the so-called "noisy neighbor" problem and ensure performance isolation among the NF instances running on cores sharing the same uncore resources.

The evaluation results are shown in Fig. 7. The three plots with markers (•), (▽), and (△) correspond to the results obtained using the uniform workload; for (•) and (▽), traffic is dispatched to each core (an L4LB or NM instance) by observing the TAG boundary, *e.g.*, on a *per-host* basis for NM; whereas for (△), *per-flow* traffic dispatching is used. Hence in the latter case, the state is shared among multiple cores; for (△) explicit locking mechanism is used to ensure strong consistency (i.e., before a core can access/update a state entry, it must first acquire the lock). We see that with the uniform workload and per-host traffic dispatching, *linear scaling* performance can indeed be attained. That the performance flattens out when using more than 16 cores for NM (versus 20 cores for L4LB due to their different NF operations) is because the number of packets have reached the 100Gbps line rate. In other words, NM can keep up with the 100Gbps line rate with 16 cores (under the uniform workload and per-host traffic dispatching); devoting more cores beyond 16 leads to a waste of core resources. In contrast, with *per-flow* traffic dispatching for NM, devoting more cores does not significantly improve the overall system throughput; in the case of *shared state with locks* (△), NM is capable of processing fewer than 10 Mpps even with the uniform workload – using more than two cores in fact degrades the overall system throughput! Finally, the plot with the (◁) marker corresponds to the results obtained using the non-uniform workload and per-host dispatching. We see that non-uniform workload degrades the performance of NM: while the scaling is still approximately linear, NM cannot keep with the line rate even with 22 cores.

Takeaway. When profiling the performance of an NF, we must carefully analyze various performance critical factors, *i.e.*, NF state, operations and workloads, and conduct performance benchmarking under various representative settings. Importantly, our empirical results suggest that in general per-TAG traffic dispatching would yield the best performance by avoiding state sharing. However, we should also note this may not always be possible: besides the issue of non-uniform workloads, an TAG may be too big to be processed by a single core so that state sharing becomes unavoidable.

V. SFC Performance Profiling

In this section we explore the interplay of the performance critical factors identified earlier and expound on the new issues that arise when profiling SFC performance. We will use the four-NF chain presented in §II-B (see also in Fig. 11): ACL → NM → L4LB → L3FW, in our experiments.

Competing NF States, TAGs with Differing Granularities, and Their Interplay with Workload Characteristics. As for a single NF, we can characterize the complexity of an SFC by adding up the number of instructions per NF, and in particular, obtain the total number of packet and state operations that the SFC must perform on each packet. The total SFC state requirement can be measured by summing up the size of each NF state. The new challenge lies in that the *stateful* NFs in an SFC have their own scopes and induce TAGs with likely differing granularities. (Here we can exclude stateless NFs from consideration – their states only add to the total state size, but induce no traffic affinity among packets.) *How should we characterize the workload then?* The workload characteristics discussed in §III-C are all defined with respect to the state scope/TAG. To get around this issue, we consider simply the *finest* and *coarsest* TAG granularities induced by all (stateful) NFs in an SFC. For the simple four-NF chain, the finest is per-flow TAGs induced by L4LB, and coarsest is per-host TAG induced by NM. The finest TAGs lead to more state entries, and thus have highest state requirements. On the other hand, the coarsest TAGs (which typically produce largest volumes

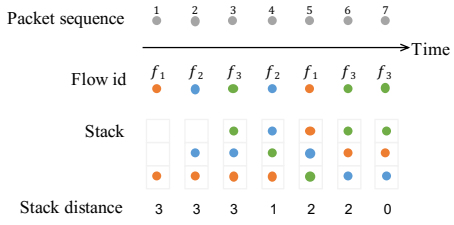


Figure 8: Metric for traffic temporal locality: an example of stack distance

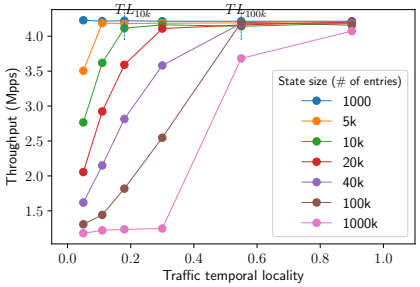


Figure 9: SFC performance and traffic temporal locality

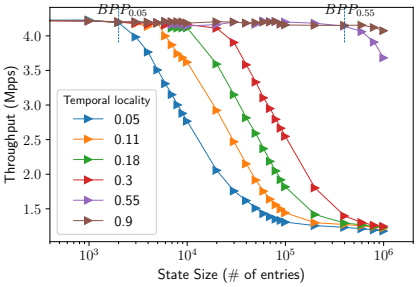
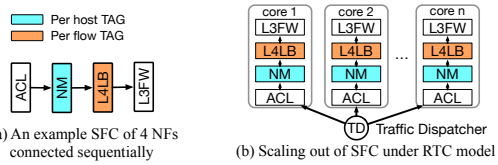


Figure 10: SFC performance and the state size in the chain



(a) An example SFC of 4 NFs connected sequentially

(b) Scaling out of SFC under RTC model

Figure 11: An example SFC and the scaling out

of packets per TAG) are the ones that likely create *shared* state entries, and thus serve as a primary performance-critical factor in deciding on traffic dispatching strategies. We note that in general there may be multiple finest TAG granularities as well as multiple coarsest TAG granularities, each defined by a different set of header fields. A fuller exploration of these issues and their implications in workload characterization is beyond the scope of this paper and is left as future work.

Interplay of NF States & Workload Characteristics, and Impact on SFC Performance. Continuing the discussion in §IV-B, we examine the interplay of NF states and workload characteristics on NF/SFC performance. Earlier we have shown that the performance of an individual NF decreases as the number of NF state increases (see Fig. 4). These experiments are done for randomly generated uniform workloads (with "worst-case" temporal/spatial localities). We show that the situations can be more complex in general. We first introduce a unified metric to measure the *temporal locality* based on *stack distance* used in programming languages [18], [19]. As illustrated in Fig. 8, a stack is used to emulate a LRU cache (of infinite size), whose depth is a pre-specified parameter, which we set as the total state size of NF/SFC. If a flow state is first referenced, the flow id (state entry) is pushed onto the stack and its stack distance is set to the stack depth (equals 3 in Fig. 8); if a packet of the same flow, *e.g.*, packet 5, arrives, the stack distance of the flow id entry is now 2. We compute the average stack distance of each state entry (corresponding to a TAG), and define the (normalized) *temporal locality* (TL) as $TL = \frac{1}{log_2(2+avg\_stack\_distance)}$. TL closer to 1 indicates better temporal locality; smaller TL is, worse the temporal locality. With TL, we are able to accommodate (and compare) varying temporal localities of TAGs with differing granularities induced by stateful NFs in an SFC in a single unified metric.

To systematically generate workloads with varying TL values, we start with a workload (say, with n flows) and re-order/re-mix the flows and their packets. As stated earlier, playing flows one by one yields the best TL value. We can interleave packets from f_1 and f_2, and then those of f_3 and f_4, etc., to reduce TL slightly. We can carry out this process with increasing larger groups of flows. When packets from $f_1, \ldots, f_n$ follow each other one by one, it yields the worst TL value. This process can be carried out in the same time with TAGs with coarsest, e.g., by ordering the flows based on `src_ip` while mixing the packet orders of the flows. Likewise, we can vary the state size by changing, e.g., # of flows or `src_ip`'s. The following experiments are conducted using the workloads thus generated. The number of flows (the finest TAG granularity) is used as a proxy for state size. Fig. 9 (best viewed in color) compares the performance of the four-NF SFC as a function of TL. We see that increasing temporal locality improves the SFC performance – there is a *temporal locality critical* point (marked as TL_x) for each given state size x beyond which the SFC can always attain the best performance. The same results are re-plotted in Fig. 10 as a function of state size. We see that the BPP identified in Fig. 4 is also determined by the TL in the workload: higher TL delays BPP, beyond which SFC performance degrades rapidly; this also indicates that with higher TL, fewer memory accesses are L3/DRAM-bound.

Scaling SFC: Traffic Dispatching for Minimizing Shared State vs. Load Balancing. In studying scaling individual NFs, we have shown that *shared* state incurs a significant performance penalty; it is best to use a traffic dispatching strategy to minimize the shared state. When scaling SFC, the situation becomes more complex. We illustrate using the four-NF SFC: if we dispatch traffic per-host using `src_ip`, coarsest TAG induced by `NM`, we avoid sharing any state. However, a per-host TAG may contain a large number of (5-tuple) flows – the finest TAG induced by `L4LB` – which causes the state size to exceed the BPP (*cf.* Fig. 4) and therefore an increasing rate of L3/DRAM bound memory accesses. This will slow down the performance of a core operating on such a per-host TAG. We first note that if the workload is highly uniform, i.e., the average number of flows in the per-host TAGs is (nearly) the same, then per-host traffic dispatching is still the best strategy. This is shown in Fig. 12 where the *blue dot* plot corresponds to the performance of the four-NF SFC with the uniform workload. Here the uniform workload contains 20 per-host TAGs, each with an equal number of flows; we increase the number of flows (per host-TAG) and devote more cores to scale out the SFC. We see that using

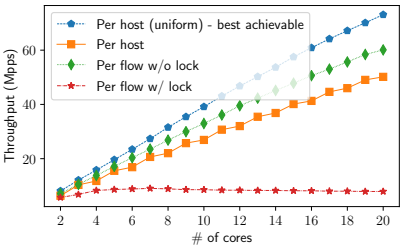


Figure 12: SFC scaling performance

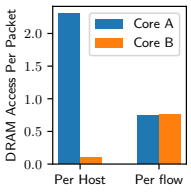


Figure 13: DRAM-bound access rates

the per-host traffic dispatching strategy, we are able to attain *linear performance scaling* in this case. Now we investigate what happens when the workload is non-uniform. We generate a non-uniform workload also containing 20 per-host TAGs, but half of them containing 10 times as many flows as the other half. We consider two dispatching strategies and the results are shown in Fig. 12: the per-host strategy (the *orange square* plot) and the per-flow (w/o lock) strategy (the *green diamond* plot). Here under the per-host strategy, all (host) TAGs assigned to some cores are "elephant" hosts. We see that with such a non-uniform workload and per-host traffic dispatching strategy, its scaling performance is, in fact, worse than the per-flow traffic dispatching strategy despite that the latter introduces shared states. This is because the loads of the cores under the per-host strategy are not balanced. This can be seen in Fig. 13, where the average number of DRAM-bound accesses per packet of a light-loaded core 'B' vs. a heavy-loaded core 'A' under the per-host strategy are shown; for comparison, the number of DRAM-bound memory accesses of two cores under the per-flow strategy are also shown, which is balanced with fewer DRAM-bound accesses per packet. Hence when devising traffic dispatching strategies for scaling out SFC, we must take into account the *trade-off between shared state vs. load balancing*. For reference, the results for the per-flow (*w/ lock*) strategy are also plotted in Fig. 12 (the *red stars*). If strong consistency is required, better state management mechanisms are needed, e.g., by using *lockless* data structures [20], [21].

Takeaway. Profiling SFC performance not only requires measuring the overall SFC operational complexity and state requirements, but also quantifying the interplay of stateful NF states in an SFC and workload characteristics. Generating representative workloads that can account for the competing and sometimes conflicting NF states, TAGs of differing granularities and their impact on SFC performance is important. When scaling out SFC, we must also carefully consider the trade-offs between shared state and unbalanced loads so as to devise better traffic dispatching strategies.

VI. Performance Bounds and Evaluation

We put forth a simple method for profiling, estimating and bounding NF/SFC performance on a multi-core server under diverse target workloads.

Basic Methodology. The first step is to understand the expected throughput achieved by running the SFC on a single core, given a target traffic demand W and an SFC instance to be executed on this target demand. Here we assume that we have some knowledge of the workload statistics and characteristics (e.g., obtained from past network data) that can be used to guide us in generating *synthetic* workloads with matching workload characteristics. Our methodology obtains an upper bound and a lower bound on the SFC performance on a single core by looking up a *performance profile* table.

This profile table is obtained by generating synthetic workloads with varying state sizes (as measured by the number of 5-tuple flows) and normalized temporal localities in discrete increments, and then benchmark the SFC performance on a single core using these workloads. The synthetic workload generation and SFC benchmarking processes are similar to what we have used in generating the results in Figs. 9 & 10. Now given a target traffic demand with the workload characteristics characterized by, say, the expected number of flows per second, wl, and TL value, tl, we use these two parameters to look up the performance profile table to find two closest points within which (wl, tl) falls: $(\hat{wl}_1, \hat{tl}_1)$ and $(\hat{wl}_2, \hat{tl}_2)$ such that $\hat{wl}_1 \leq wl \leq \hat{wl}_2$ and $\hat{tl}_1 \geq tl \geq \hat{tl}_2$. The corresponding performance measures, $P(\hat{wl}_1, \hat{tl}_1)$ and $P(\hat{wl}_2, \hat{tl}_2)$ produce an upper and a lower bound on $P(wl, tl)$, the SFC performance on the target workload using a single core. Here we assume that the target workload falls within the throughputs achievable by the SFC on a single core.

The final step of our methodology aims to provide the scaling performance bound for SFC instances running on n cores. As shown in §IV&V, if cores share the state with lock, the overall performance is bounded by the shared state and almost won't scale up with the increase of the cores. Otherwise without shared state, the key factor affecting the scaling performance becomes the traffic dispatching strategy. We present how to obtain the lower and upper scaling performance bound under the following two situations. One situation assumes a traffic dispatching strategy is given, then we model the performance bound for the given strategy and workload. With n cores, the target demand is divided into n sub-workloads, $\{\bar{w}_i, i = 1, \ldots, n\}$. As our performance profile table has provided the bound for core i the steered workload w_i, where we mark the lower bound as $p(\underline{w}_i)$ and the upper bound as $p(\bar{w}_i)$. The scaling performance for n cores P is bounded by $\sum_{i=1}^{n} p(\underline{w}_i) \leq P \leq \sum_{i=1}^{n} p(\bar{w}_i)$. Another situation where we are allowed to select a best traffic dispatching strategy from a set of policies pre-specified, we can repeat the process of the first situation for each dispatching policy and select the best estimates as the final bounds.

Case Study using Real Traffic. We evaluate the estimates produced by the simple profiling model using a real network packet trace of (~1Tb) from [1]. We divide the dataset into multiple segments of varying sizes and workload characteristics. The statistics (the number of flows and normalized temporal locality) for 8 sample workload segments are shown along the x-axis of Fig. 14. To obtain the real performance results, we run the four-NF SFC on these workloads. The results are shown in Fig. 14 as marked by the green $\times$ markers. The corresponding estimated upper and lower bounds obtained by looking up the performance profile table are shown in Fig. 14

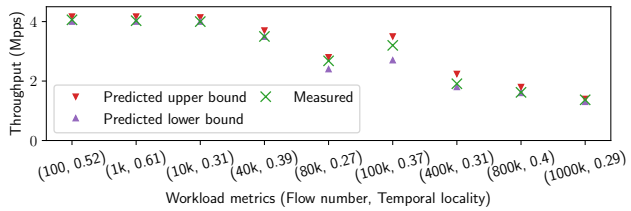


Figure 14: Evaluation of the profiling model using real traffic

with the $\triangle$ and $\triangledown$ markers. We see that for these 8 workloads, the performance obtained from experiments on the real traffic falls within the upper and lower bound estimates. These results show both the validation of our methodology and the accurate of the performance profiling model.

VII. Related Works

NFV performance profiling. Based on static program analysis, Bolt [11] employs source code analysis of NFs to generate performance contracts and [22] generates the adversarial workload instead of typical benchmarking workload to quantify worst-case performance for NFs. Both of these works ignore the scaling issues. Works [23], [24], build runtime framework to diagnose NFV performance but focus on anomaly detection. In particular, [25] investigated the performance under various attack traffic but only for single NF Deep Packet Inspection (DPI), while our framework focus on normal workloads and is general to different NFs and SFCs. Other relevant to our work are those in [26], [27] which examine the impact of multi-core NUMA architecture on NFV performance. None of those works above take both the competing NF states, different granularities of TAGs and their interplay with workload characteristics into consideration while profiling, which are part of key insights provided by this paper.

NFV scalability. One key challenge in scaling SFC is NF state management, which have been investigated in several studies [15], [16], [14]. Different mechanisms, such as in-memory data store [15], [14] and distributed shared space [16] are proposed to improve the scaling performance of NF. While these studies provide valuable insights to our work, they do not consider the impact of fine-grained state resources requirements on NF performance. Our work takes both the SFC specific constraints and micro-architecture of multi-core server into consideration, towards the first step to systematically identify the critical factors for scaling performance.

VIII. Conclusion

We have presented a conceptual framework to characterize NF/SFC behavior with an accompanying set of key concepts and metrics. To the best of our knowledge, our study is the first that illuminates various key factors affecting NFV performance in a multi-core server environment and identifies the key challenges in profiling and scaling SFC performance. Based on the observations and results obtained, we put together a systematic methodology to guide network operators in profiling the performance of individual NFs and SFCs. Our results shade lights on future NFV research directions, *e.g.*, developing automated NFV performance profiling and adaptive runtime systems that dynamically provision system resources for NEC to meet changing traffic demands.

Acknowledgment

This research was supported in part by US NSF grants CNS-1618339, CNS-1617729, CNS-1814322, and CNS-1836772. Peng Zheng and Wendi Feng gratefully acknowledge the financial support from China Scholarship Council.

References

[1] "CAIDA Traffic Dataset," http://www.caida.org, 2019.
[2] "DPDK," http://dpdk.org/, 2018, accessed: 2019-04-11.
[3] A. Panda, S. Han, K. Jang, M. Walls *et al.*, "Netbricks: Taking the v out of NFV," in *Proc. of OSDI'16*, 2016, pp. 203–216.
[4] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr., "Metron: NFV service chains at the true speed of the underlying hardware," in *Proc. of NSDI'18*, 2018, pp. 171–186.
[5] P. Zheng, A. Narayanan, and Z.-L. Zhang, "A closer look at NFV execution models," in *Proc. of APNet'19*, 2019.
[6] S. Palkar, C. Lan *et al.*, "E2: A framework for NFV applications," in *Proc. of SOSP'15*, 2015, pp. 121–136.
[7] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *Proc. of NSDI'14*, 2014, pp. 459–473.
[8] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *Proc. of NSDI'14*, 2014, pp. 445–458.
[9] Intel, "DPDK Packet Classification and Access Control library," 2019, https://doc.dpdk.org/guides-18.11/prog_guide/packet_classif_access_ctrl.html. Accessed: 2019-04-9.
[10] "DPDK hash library," 2019, https://doc.dpdk.org/guides/prog_guide/hash_lib.html. Accessed: 2019-04-9.
[11] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea, "Performance contracts for software network functions," in *Proc. of NSDI'19*, 2019, pp. 517–530.
[12] Intel, "DPDK rteflow API," 2019, https://doc.dpdk.org/guides/prog_guide/rte_flow.html. Accessed: 2019-04-11.
[13] "Cisco T-Rex: Realistic traffic generator," 2019. [Online]. Available: https://trex-tgn.cisco.com
[14] J. Khalid and A. Akella, "Correctness and performance for stateful chained network functions," in *Proc. of NSDI'19*, Boston, MA, 2019.
[15] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. of NSDI'17*, 2017, pp. 97–112.
[16] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. of NSDI'18*, 2018, pp. 299–312.
[17] Intel, "Increasing Platform Determinism with Platform Quality of Service for the Data Plane Development Kit - White Paper," 2016.
[18] S. JR, "Program locality and dynamic memory management," Ph.D. dissertation, Dept. of Elec. Eng., Princeton Univ., 1973, ph.D Dissertation.
[19] C. Fox, D. Lojpur, and A. A. Wang, "Quantifying temporal and spatial localities in storage workloads and transformations by data path components," in *Proc. of MASCOTS'08*, 2008, pp. 1–10.
[20] "An Introduction to Lock-Free Programming," https://preshing.com/20120612/an-introduction-to-lock-free-programming, 2019.
[21] N. Shavit, "Data structures in the multicore age," *Communications of the ACM*, vol. 54, no. 3, pp. 76–84, 2011.
[22] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki, "Automated synthesis of adversarial workloads for network functions," in *Proc. of SIGCOMM'18*, 2018, p. 372–385.
[23] J. Nam, J. Seo, and S. Shin, "Probius: Automated approach for vnf and service chain analysis in software-defined nfv," in *Proc. of SOSR'18*, 2018, pp. 14:1–14:13.
[24] P. Naik, D. K. Shaw, and M. Vutukuru, "NFVPerf: Online performance monitoring and bottleneck detection for NFV," in *Proc. of IEEE NFV-SDN'16*, 2016, pp. 154–160.
[25] Y. Afek, A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, "Making DPI engines resilient to algorithmic complexity attacks," *IEEE/ACM Transactions on Networking*, pp. 3262–3275, 2016.
[26] M. Dobrescu, N. Egi *et al.*, "Routebricks: Exploiting parallelism to scale software routers," in *Proc. of SOSP'09*, 2009, pp. 15–28.
[27] A. Tootoonchian, A. Panda *et al.*, "Resq: Enabling slos in network function virtualization," in *Proc. of NSDI'18*, 2018, pp. 283–297.