# Metron: NFV Service Chains at the True Speed of the Underlying Hardware

Georgios P. Katsikas[1,3], Tom Barbette[2], Dejan Kostić[3], Rebecca Steinert[1], Gerald Q. Maguire Jr.[3]

[1]RISE SICS, [2]University of Liege, [3]KTH Royal Institute of Technology

## Abstract

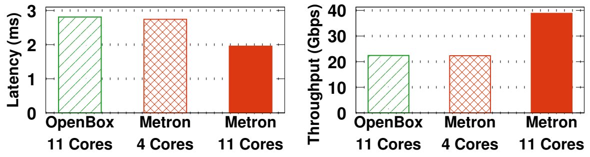
In this paper we present Metron, a Network Functions Virtualization (NFV) platform that achieves high resource utilization by jointly exploiting the underlying network and commodity servers' resources. This synergy allows Metron to: (*i*) offload part of the packet processing logic to the network, (*ii*) use smart tagging to setup and exploit the affinity of traffic classes, and (*iii*) use tag-based hardware dispatching to carry out the remaining packet processing at the speed of the servers' fastest cache(s), with zero inter-core communication. Metron also introduces a novel resource allocation scheme that minimizes the resource allocation overhead for large-scale NFV deployments. With commodity hardware assistance, Metron deeply inspects traffic at 40 Gbps and realizes stateful network functions at the speed of a 100 GbE network card on a single server. Metron has 2.75-6.5x better efficiency than OpenBox, a state of the art NFV system, while ensuring key requirements such as elasticity, fine-grained load balancing, and flexible traffic steering.

## 1 Introduction

Following the success of Software-Defined Networking (SDN), Network Functions Virtualization (NFV) is poised to dramatically change the way network services are deployed. NFV advocates running chains of network functions (NFs) implemented as software on top of commodity hardware. This is in contrast with chaining expensive, physical middleboxes, and brings numerous benefits: (*i*) decreased capital expenditure and operating costs for network service providers and (*ii*) facilitates the deployment of exciting new services.

Achieving high performance (high throughput and low latency with low variance) using commodity hardware is a hard problem. As 100 Gbps switches and network interface cards (NICs) are starting to be standardized and deployed, maintaining high performance at the ever-increasing data rates is vital for the success of NFV.

In an NFV service chain, packets move from one physical or virtual server (hereafter simply called server) to another to realize a programmable data plane. The servers themselves are predominantly multi-core machines. Different ways of structuring the NFs exist, e.g., one per physical core or using multiple threads to leverage multiple cores within each NF. Network functions range from simple stateless ones to complex, such as deep packet inspection (DPI), and potentially stateful (e.g., proxy) ones. Regardless of the deployment model and NF types, every time a packet enters a server, a fundamental problem occurs: how to locate the core within the multi-core machine that is responsible for handling this packet? This problem reoccurs every step of the chain and can cause costly inter-core transfers.



**(a)** Latency (ms). **(b)** Throughput (Gbps).

**Figure 1:** Thanks to zero inter-core transfers, Metron has almost 3x better efficiency than the state of the art when deeply inspecting (Firewall→DPI) traffic at 40 Gbps.

Our work, Metron, eliminates unnecessary inter-core transfers and in a 40-Gbps setup (Figure 1) achieves: (*i*) about a factor of 3 better efficiency, (*ii*) lower, predictable latency, and (*iii*) 2x higher throughput than OpenBox [13], a state of the art NFV system.

### 1.1 NFV Processing Challenges

To identify the core that will process an incoming packet, the NFV framework can typically only examine the header fields. Here, there is a big mismatch between the way modern servers are structured and the desired packet dispatching functionality. Figure 2 shows three widely used categories of packet processing models in NFV.

The first category (see Figure 2a), augments the weak programmability of current NICs with a software layer that acts as a programmable traffic dispatcher between the hardware and the overlay NFs. E2 [59], with its software component called SoftNIC [25], falls into this category. SoftNIC requires at least one dedicated CPU core for traffic dispatching and steering (see Figure 2a), while the NFs run on other CPU cores. Earlier works, such as ClickOS [49] and NetVM [29], also used software switches on dedicated cores to dispatch packets to virtual machines, but without the flexibility of E2.

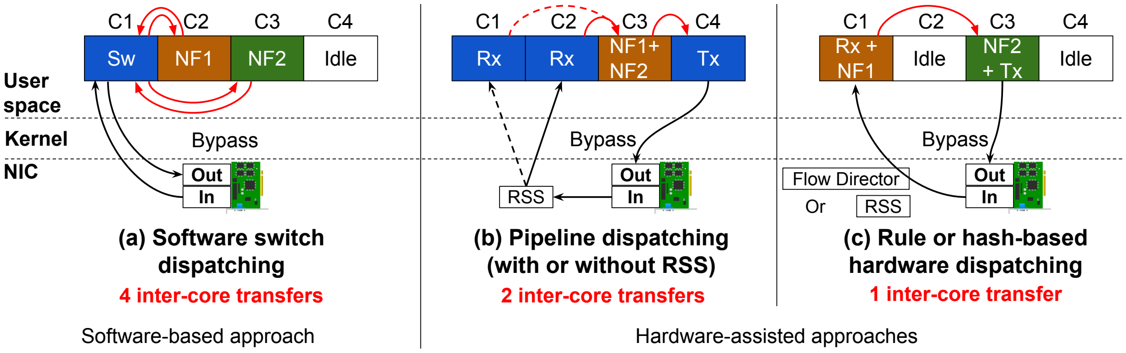


**Figure 2:** State of the art packet processing models either have too many inter-core packet transfers or load balancing problems due to load imbalance and/or idle CPU cores. RSS stands for Receive Side Scaling.

Rather than having a shim layer between the NFs and the NICs to select the next hop in a service chain, the second category of packet processing models (see Figure 2b) involves a pipeline of reception, processing, and transmission threads, each on a different (set of) core(s). If more than one reception core is required, this model uses RSS [30] as described below. For example, OpenNetVM [71], Flurries [70], and NFP [63] (a parallel version of OpenNetVM) fall into this category. Similar to E2, these works introduce programmability by augmenting the reception and processing parts of the pipeline with traffic steering abilities.

The last category of packet processing models relies on two hardware features provided by a large fraction of NIC vendors today. First, RSS uses a static function to dispatch traffic to a set of CPU cores by hashing the values of specific header fields. Second, NICs can be programmed via a vendor specific "match-action" API to dispatch traffic to specific cores (e.g., Intel's Flow Director [31]). Unlike all previous models, these approaches do not require dedicated dispatchers, hence they offer higher performance. OpenBox [1] [13], FastClick [6], SNF [36], and RouteBricks [18] use RSS, while CoMb [62] uses Flow Director.

None of these schemes guarantee that the core that receives the incoming packet will be the one processing it. Flow hashing as in RSS can introduce serious load imbalances under skewed (e.g., heavy flows with the same hashes) workloads. Flow Director permits explicit flow affinity, but suffers from the limited classification capabilities of today's commodity NICs. When there is a mismatch, the packet is handed off to the correct core. However, this requires transferring the packet via DRAM or last level cache (LLC) to the target processing core. This is a slow operation, as the LLC takes several tens of cycles even for a cache hit! Our earlier work [37] demonstrates that dramatic slowdowns occur due to this effect. In particular, an order of magnitude better performance (both higher throughput and lower latency variance) is possible if the correct core receives the packet straight from the NIC, and the packet remains in the core-specific L1 or L2 cache.

## 1.2 Metron Research Contributions

We present Metron, a system for NFV service chain placement and request dispatching. To the best of our knowledge, Metron is the first system that automatically and dynamically leverages the joint features of the network and server hardware to achieve high performance. Metron eliminates inter-core transfers (unlike recent work with 4 [59], 2 [70], or 1 [13] inter-core transfers as shown in Figure 2), making it possible to process packets potentially at L1 cache speeds. Also, we overcome the load balancing issues of "run-to-completion" approaches [18, 6, 13, 36], by combining smart identification, tagging, and dispatching techniques. We had to address a number of challenging problems to realize our vision. First, making efficient use of all the available hardware is hard because of the in-machine request dispatching overheads (described earlier). Second, discovering and dealing with the heterogeneous network (switches, NICs) and server hardware, in a generic way, is non-trivial from a management perspective. Third, detecting and dealing with load imbalances that reduce the performance of the initially placed service chains requires rapid and stable adaptation. We state our research contributions, while dealing with the aforementioned challenges:

**Contribution 1:** We orchestrate programmable network's hardware to perform stateless processing and packet classification. We deal with hardware heterogeneity by building upon the unified management abstractions of an industrial-grade SDN controller (ONOS [7]). This allows Metron to leverage popular management protocols, such as OpenFlow [50] and P4 [11], and easily integrate future ones. We contributed a new driver for programmable NICs and servers [35].

[1] Originally, OpenBox was built on top of Mininet and Click [42] using Linux-based I/O. To fairly compare it against our work, we accelerated OpenBox using FastClick's DPDK engine and RSS [4].

**Contribution 2:** We overcome the network/server architecture mismatch by instructing Metron to tag packets as early as possible, enabling them to be quickly and efficiently switched and dispatched throughout the entire chain. To do so, Metron first uses SNF [36] to identify the traffic classes of a service chain and produce a synthesized NF that performs the equivalent work of the entire chain (see §2.3.1). Then, Metron divides the synthesized NF into stateless and stateful operations (see §2.3.3) and instructs all available programmable hardware (i.e., switches and NICs) to implement the stateless operations, while dispatching incoming packets to those CPU cores that execute their stateful operations. Metron runs stateful NFs on general purpose servers, while fully leveraging their generic processing power.

**Contribution 3:** We propose a way to efficiently and quickly obtain the network state in order to make fast placement decisions at low cost with high accuracy (see §2.3.3). We devised a mechanism to coordinate load balancing among servers and their CPU cores, demonstrating that Metron provides comparable elasticity with purely software-based approaches, but at the true speed of the hardware (see §2.3.4).

Our evaluation shows that Metron realizes deep packet inspection at 40 Gbps (§3.1.1) and stateful service chains at the speed of a 100 GbE NIC on a single server (§3.1.2). This results in up to 4.7x lower latency, up to 7.8x higher throughput, and 2.75-6.5x better efficiency than the state of the art. It is difficult to improve on this performance unless we completely offload stateful chains to hardware, which is impossible with today's commodity hardware.

## 2 System Architecture

This section describes Metron's system design, starting with a high-level overview via an illustrative example in §2.1. In §2.2 we describe the Metron data plane, which is configured by the Metron controller (see §2.3).

### 2.1 Overview

To understand how Metron works, consider a simple network consisting of two OpenFlow switches connected to a server as shown at the bottom of Figure 3. Assume that an operator wants to deploy a Firewall→DPI service chain, as shown in Step 1 of Figure 3.

In Step 2, the Metron controller identifies the traffic classes [2] of the service chain, by parsing the packet processing graphs (each graph has a set of packet processing elements as in [42, 59, 13]) of the input NFs. In Step 3, Metron composes a single service chain-level graph by synthesizing the read and write operations of the individual graphs (see §2.3.1). Because Metron detects the availability of resources (i.e., the OpenFlow switches) along the path to the server, it associates stateless read and write operations with these components and automatically translates these operations into OpenFlow rules (Step 4.1). The remaining, potentially stateful, operations are translated into software instructions targeting the Metron agent at the server (Step 4.2). The key to Metron's high performance is exploiting hardware-based dispatching (Step 5) that annotates the traffic classes matched by the OpenFlow rules with tags that are subsequently matched by the server's NIC to identify the CPU core to execute the stateful operations. In this way, Metron guarantees that each traffic class will be processed by a single core, thus eliminating costly inter-core communications. This guarantee is maintained even when a CPU core becomes overloaded (see §2.3.4) as the Metron agent reports run-time statistics (Step 6) that allow the Metron controller to rebalance the load (Step 7) by splitting traffic classes into multiple groups that are dispatched to different cores using different tags (Steps 8.1 and 8.2). We conclude this overview with a survey of popular NFs; noting that in Table 1 a substantial portion of these NFs can be (fully or partially) offloaded to commodity hardware.

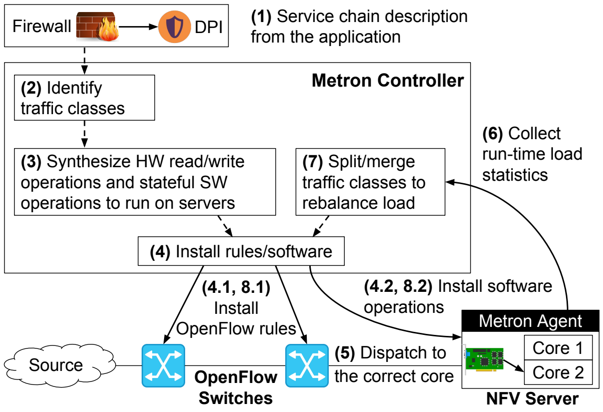

Firewall
DPI
(1) Service chain description from the application
Metron Controller
(2) Identify traffic classes
(3) Synthesize HW read/write operations and stateful SW operations to run on servers
(7) Split/merge traffic classes to rebalance load
(6) Collect run-time load statistics
(4) Install rules/software
(4.1, 8.1) Install OpenFlow rules
(4.2, 8.2) Install software operations
Metron Agent
Source
OpenFlow Switches
(5) Dispatch to the correct core
Core 1
Core 2
NFV Server

**Figure 3:** Metron overview using an example NF chain.

**Table 1:** Survey of popular NFs. The offloadability of "Hybrid" NFs depends on the use case.

| Network Function | Offloadable to Hardware |
|---:|:---|
| L2/L3 Switch, Router | Yes |
| Firewall/Access Control List (ACL) | Hybrid |
| Carrier Grade NA(P)T, IPv4 to IPv6 | No |
| Broadband Remote Access Server | Partially [17] |
| Evolved Packet Core | Partially |
| Intrusion Detection/Prevention | Partially [32] |
| Load Balancer | Hybrid |
| Flow Monitor | Yes |
| DDoS Detection/Prevention | Yes [43] |
| Congestion Control (RED, ECN) | Yes |
| Deep Packet Inspection (DPI) | No |
| IP Security, Virtual Private Network | Yes [61] |

[2] Traffic class is a (set of) flow(s) treated identically by an NF chain.

## 2.2 Metron Data Plane

The Metron data plane follows the master/slave approach depicted in Figure 4. The master process is an agent that interacts with $(i)$ the underlying hardware by establishing bindings with key components, such as NICs, memory, and CPU cores and $(ii)$ the Metron controller through a dedicated channel.

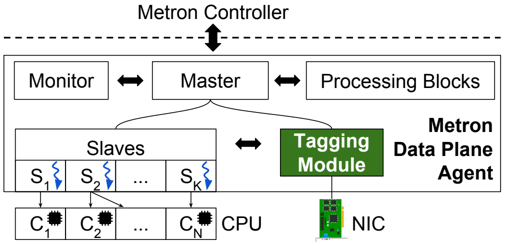


**Figure 4:** The Metron data plane.

The key differentiator between Metron and earlier NFV works is the tagging module shown in Figure 4. This module exposes a map with tag types and values that each NIC uses to interact with each CPU core of a server; this map is advertised to the Metron controller. The controller dynamically associates traffic classes with specific tags in order to enforce a specific flow affinity, thus controlling the distribution of the load. Most importantly, this traffic steering mechanism is applied by the hardware (i.e., NICs), hence Metron does not require additional CPU cores (as E2 does) to perform this task, thus packets are directly dispatched to the CPU core that executes their specific packet processing graph. In §3, we use a tagging scheme for trillions of service chains.

When the master boots, it configures the hardware and registers with the controller by advertising the server's available resources and tags. Then, the master waits for controller instructions. For example, the master executes a deployment instruction by spawning a slave process that is pinned to the requested core(s) and by passing the processing graph to the slave. In the context of service chaining, a Metron slave needs to execute multiple processing graphs, each corresponding to a different NF in the chain. Such graphs can be implemented either in hardware or software. Earlier works implement these graphs in software and use metadata to share information among NFs and to define the next hop in a chain. Although Metron supports this type of software-based chaining, as shown in §1.1, this approach introduces unnecessary overhead due to excessive inter-core communication and potentially under-utilizes the available hardware. Next, §2.3 explains how we approach and solve this problem.

## 2.3 Metron Control Plane

Here, we describe the key design choices and properties of the Metron controller.

### 2.3.1 Synthesis of Packet Processing Graphs

Given a set of input packet processing graphs, one per NF, Metron combines them into a single service chain graph. To ensure low latency, the Metron controller adopts SNF [36]; a more aggressive variant of OpenBox for merging packet processing graphs, which provides a heuristic for solving the graph embedding problem (see [68, 27, 15]) in the context of NFV. Metron uses SNF to eliminate processing redundancy by synthesizing those read and write operations that appear in a service chain as an optimized equivalent packet processing graph. SNF guarantees that each header field is read/written only once, as a packet traverses the graph.

Another benefit of SNF's integration into Metron is the ability to encode all the individual traffic classes of a service chain using a map of disjoint packet filters $(\Phi)$ to a set of operations $(\Omega)$. In §2.3.4 we use this feature to automatically scale packet processing in and out, providing greater elasticity than available today.

### 2.3.2 Initial Resource Allocation

To allocate resources for the synthesized graph, we allow application developers to input the CPU and network load requirements of their service chains. Alternatively, this information can be obtained by running a systematic NFV profiler, such as SCC [37], or by using more generic profilers, such as DProf [60]. Even in the absence of accurate resource requirements, Metron dynamically adapts to the input load as discussed in §2.3.4.

### 2.3.3 Placement

Metron needs to decide where to place the synthesized packet processing graph. Such a decision is not simple, because Metron not only considers servers but also the network elements along the path to these servers.

Table 1 showed that a large fraction of NFs cannot be implemented in commodity hardware today, mainly because they require maintaining state. This means, that the synthesized graph of such NFs cannot be completely offloaded. To solve this, we designed a graph separation module to traverse and split the synthesized graph into two subgraphs. The first subgraph contains the packet filters and operations that can be completely offloaded to the network (we call this a stateless subgraph), while the second (stateful) subgraph will be deployed on a server. The average complexity of this task is $O(\log m)$, where $m$ is the number of vertices of the synthesized graph.

Given these two subgraphs, Metron needs to find a pair of nodes (a server and a network element) that satisfy two requirements: $(i)$ the server has enough processing capacity to accommodate the stateful subgraph and $(ii)$ the network element has enough capacity to store the hardware instructions (e.g., rules) that encode the stateless subgraph.

**Scalable Placement with Minimal Overhead**
In large networks with a large number of servers and switches, it is both expensive and risky to obtain load information from all the nodes. This is expensive because a large number of requests need to be sent frequently and this would occupy bandwidth to each node, generate costly interrupts to fetch the data, and occupy additional bandwidth to return responses to the controller. This is risky because the round-trip time required to obtain the monitoring data is likely to render this data stale, leading to herd behaviors and suboptimal decisions. To make a placement decision with minimal overhead, we use the simple, yet powerful, opportunistic scheme of "the power of two random choices" [52]. According to Mitzenmacher, this number offers exponentially better load balancing than a single random choice, while the additional gain of three random choices only corresponds to a constant factor.

Metron queries the load of two randomly selected servers and selects the least loaded of them, provided that the necessary resource requirements (i.e., number of NICs and CPU cores) can be met. If the first two choices fail, then these two servers are removed from the list and the process is repeated until a server is found. Note that this scheme prioritizes deployments that exhibit spatial correlation with respect to the processing location because spreading this processing results in lower performance, which is undesirable.

This server selection procedure also greatly simplifies the second placement decision (i.e., the network element(s) to offload processing to). Well designed networks, such as datacenters, provision several fixed shortest paths between ingress nodes (e.g., core switches) and servers, where each server might be associated with a single core switch [1, 2]. Given this, we find the most suitable network element to offload the stateless graph, using the following inputs: (*i*) the topology graph, (*ii*) the server where the stateful subgraph will be deployed (chosen by the server selection scheme), and (*iii*) the rule capacity required to offload the stateless subgraph.

**Handling Partial Offloading and Rule Priorities**
Metron carefully treats the cases when (*i*) a stateless subgraph contains rules with different priorities and (*ii*) one or more rules of such a subgraph cannot be offloaded to hardware. The latter can occur, e.g., due to the hardware's inability to match specific header fields. In such a case, Metron will selectively offload only the supported rules, while respecting rule priorities. To exemplify these two cases, assume a service chain that needs to be deployed on the topology shown in Figure 3. Assume that this service chain implements four rules that can be offloaded to the first programmable switch, while the remaining part of the service chain will be deployed on the server. If rule 3 cannot be offloaded and all of the rules have the same priority, then Metron will offload rules 1, 2, and 4. However, if these rules have, e.g., decreasing priorities (i.e., rule 3 has a higher priority than rule 4), then Metron will offload only the first two rules, to guarantee that the server applies rule 4 after rule 3.

### 2.3.4 Dynamic Scaling

In §2.3.1 we explained how Metron encodes a service chain as a set of individual traffic classes, where each traffic class is a set of packet filters mapped to write operations. This abstraction gives great flexibility when scaling a service chain in/out. As an example, when E2 detects an overloaded NF, it scales this NF by introducing an additional (duplicate) instance of the entire NF and then evenly splitting the flows across the two instances. In contrast, Metron splits the traffic classes of this NF across the two instances, such that each instance executes the code responsible for each of its traffic classes (rather than the code of the entire NF).

To trigger a scaling decision, Metron gathers port statistics from key locations in the network in order to detect load changes. Such a change results in Metron asking for instantaneous CPU load and network statistics from the affected service chains. Given this information, Metron applies the following, globally orchestrated, scaling strategy to react to load imbalances.

**Traffic Class-level Scaling**
We leverage a grouping technique when creating a service chain's traffic classes. A set of T traffic classes $\{TC_i^j \mid j \in [1,T]\}$ that belong to service chain $i$ can be grouped together, if and only if their packet filters $\{\Phi_i^j \mid j \in [1,T]\}$ are mapped to the same write operations: $\forall k,l \in [1,T], \Omega_i^k = \Omega_i^l$

For example, an HTTP and an FTP traffic classes heading to a NAT will both exhibit the same stateful write operations from this NF, thus they can be grouped together. The Metron controller has this information available once the traffic classes of a service chain are created (see §2.3.1). To dynamically scale out a group of traffic classes, Metron needs to split this group into two or more subgroups, where the first subgroup remains on the same CPU core as the original group, while the other subgroup(s) are deployed and scheduled on a different (set of) CPU core(s). These new traffic classes are annotated with different tags, such that the NIC at the server can dispatch them to the appropriate CPU cores. We call this mechanism "traffic class deflation" to differentiate it from the opposite "traffic class inflation" process, where two or more groups of traffic classes that exhibit the same write operations are merged together, when Metron detects low CPU utilization.

To simplify load balancing, while keeping a reasonable degree of flexibility, the split and merge processes always use a static factor of 2 (i.e., one group is split into two, or two groups are merged into one). This decision also minimizes the amount of state that Metron needs to transfer across CPUs. A fully dynamic solution with additional visibility into the load of each traffic class would achieve better load distribution; however, such a solution is considered impractical in the case of large networks with potentially millions of traffic classes. Split and merge operations may repeat until Metron can no longer split/merge a traffic class. A single flow is an example of non-splittable traffic class. The reaction time of this strategy is mainly affected by the time required for the controller to monitor and reconfigure the data plane. In §3.2 we show how this strategy performs in practice.

Once an inflation/deflation decision has been made, Metron needs to guarantee that the state of the affected traffic classes (e.g., those being redirected to a different CPU core in the case of inflation) will remain consistent. To do so we adopt a scheme that quickly duplicates the stateful tables of a group of traffic classes across the involved CPU cores, when inflation occurs. Similarly, we merge the stateful tables of two groups during the deflation process. Although this scheme introduces some redundancy (entries of migrated traffic classes will still occupy space in the memory of the previous CPU core until they expire), it offers a quick solution to a problem that is beyond the scope of this work. StateAlyzr [39], OpenNF [23], or the work by Olteanu and Raiciu [54] could be integrated into Metron to provide more efficient state management solutions. Alternatively, state management could be delegated to a remote distributed store as per Kablan, et al. [33].

### 2.3.5 Integrating Blackbox NFs

Some NF providers might not wish to disclose the source code of their NFs. In this case we offer two integration strategies: ($i$) partially synthesize a service chain, while using DPDK ring buffers to interconnect synthesized NFs with blackbox NFs and ($ii$) input only an NF configuration (e.g., DPI rules, omitting DPI logic) using Metron's high-level API and let Metron use its own data plane elements to realize this NF (see §3.1).

## 2.4 Routing (Updates) and Failures

To explain how Metron's routing and dispatching works and how Metron reacts to routing updates and failures, we use the example shown in Figure 5. We assume a software-defined[3] network on which the network operator has deployed a routing application that routes HTTP traffic[4] between source and destination (through the path s1→s3). The routing is done using the information shown within green dashed-dotted outlines.

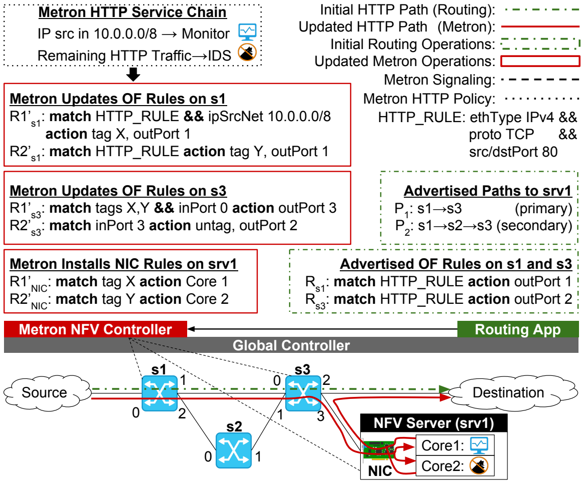

Metron HTTP Service Chain
IP src in 10.0.0.0/8 → Monitor
Remaining HTTP Traffic→IDS
Initial HTTP Path (Routing):
Updated HTTP Path (Metron):
Initial Routing Operations:
Updated Metron Operations:
Metron Signaling:
Metron HTTP Policy:
HTTP_RULE: ethType IPv4 && proto TCP && src/dstPort 80
Metron Updates OF Rules on s1
R1'_s1: match HTTP_RULE && ipSrcNet 10.0.0.0/8 action tag X, outPort 1
R2'_s1: match HTTP_RULE action tag Y, outPort 1
Metron Updates OF Rules on s3
R1'_s3: match tags X,Y && inPort 0 action outPort 3
R2'_s3: match inPort 3 action untag, outPort 2
Advertised Paths to srv1
P_1: s1→s3 (primary)
P_2: s1→s2→s3 (secondary)
Metron Installs NIC Rules on srv1
R1'_NIC: match tag X action Core 1
R2'_NIC: match tag Y action Core 2
Advertised OF Rules on s1 and s3
R_s1: match HTTP_RULE action outPort 1
R_s3: match HTTP_RULE action outPort 2
Metron NFV Controller
Routing App
Global Controller
Source
s1
1
0
2
s2
0
1
s3
0
2
1
3
Destination
NFV Server (srv1)
Core1:
Core2:
NIC

**Figure 5:** Metron's routing & CPU dispatching scheme.

A policy change forces the network operator to further process the HTTP traffic before it reaches its destination. Thus, she deploys an HTTP service chain (described by the top box with dotted outline in Figure 5) using Metron. When Metron boots it obtains the current routing policy and paths for the HTTP traffic, as advertised by the routing application. Next, the Metron controller performs a set of updates (see the left-side boxes with solid outlines, where OF stands for OpenFlow). The updates focus on two aspects: ($i$) to extend the existing HTTP rules (i.e., $R_{s1}$ and $R_{s3}$ at the bottom right box with dashed-dotted outline) with rules that also perform part of the service chain's operations (i.e., $R1'_{s1}$ and $R2'_{s1}$) and ($ii$) to tag the HTTP traffic classes to allow the NFV server to dispatch them to different CPU cores.

In this example, Metron identifies two traffic classes and tags them with tags X and Y. The tagging is applied by the first switch (i.e., s1 as explained in §2.3.3) using the rules $R1'_{s1}$ and $R2'_{s1}$ (top left box with solid outline). The next switch (s3) uses the tags (i.e., rule $R1'_{s3}$) to redirect the HTTP traffic classes to the NFV server, where Metron has installed NIC rules (i.e., $R1'_{NIC}$ and $R2'_{NIC}$) to dispatch packets with tags X and Y to CPU cores 1 and 2 respectively. The first core executes a monitoring NF, while the second core runs an intrusion detection system (IDS) NF. After traversing the service chain, the packets return to s3, where another Metron rule (i.e., $R2'_{s3}$) redirects them to their destination.

If not carefully addressed, a routing change or failure might introduce inconsistencies. Metron avoids these problems by using the paths to the NFV server (i.e., $P_1$ and $P_2$), as advertised by the routing application, to precompute: ($i$) alternative switches that can be used to offload part of a service chain's packet processing operations (see §2.3.3) and ($ii$) the actual rules to be installed in these switches. In this example, a routing change from path $P_1$ to $P_2$ (due to a routing update or

[3] Metron can also operate in legacy networks by adding one or more programmable switches before the NFV servers.

[4] We assume only HTTP traffic to keep the example simple.

a link failure between s1 and s3) will result in Metron installing 2 additional rules in s2 (these rules follow same logic with the rules in s3). Metron also updates the first rule of s3 by changing the inPort value to 1 rather than 0.

Backup configurations are kept in Metron's distributed store and are replicated across all the Metron controller instances in order to maintain a global network view. When a routing change or failure occurs, Metron applies the appropriate backup configuration. In §3.3 we show that Metron can install 1000 rules in less than 200 ms, hence quickly adapting to routing changes and failures, even those requiring a large number of rule updates.

# 3 Evaluation

**Implementation:** We built the Metron controller on top of ONOS [7, 56], an open source, industrial-grade network operating system that is designed to scale well. Key to our decision was the fact that ONOS exposes unified abstractions for a large variety of network drivers that cover popular network configuration protocols, such as OpenFlow [50], P4 [11], NETCONF/YANG [20, 10], SNMP [14], and REST. We extended ONOS with a new driver that remotely monitors and configures NFV servers and their NICs. This driver is available at [35].

Metron's data plane extends FastClick [6]. We use the Virtual Machine Device Queues (VMDq) of DPDK [19] 17.08 to implement the hardware dispatching based on the values of the destination MAC address or VLAN ID fields. Our prototype (available at [5]) uses the former header field as a filter, because the large address space of a MAC address provides unique tags for trillions [5] of service chains. To scale to 100 Gbps, Metron instructs the hardware classifier of a Mellanox NIC (§3.1.2).

**Testbed:** Our testbed consists of 3 identical servers, each with a dual socket 16-core Intel®Xeon® CPU E5-2667 v3 clocked at 3.20 GHz. The cache sizes are: 2x32 KB L1 (instruction and data caches), 256 KB L2, and 20 MB L3. Hyper-threading is disabled and the OS is the Ubuntu 16.04.2 distribution with Linux kernel v.4.4. Each server has 2 dual-port 10 GbE Intel 82599 NICs.

We deploy a testbed with a NoviFlow 1132 OpenFlow switch [53] with firmware version NW400.2.2 and we attach 2 servers to this switch. The 4 ports of the first server are connected to the first 4 ports of the switch to inject traffic at 40 Gbps. Then, ports 5-8 of the switch are connected to the 4 ports of the second server, where traffic is processed by the NFV service chains being tested and sent back to the origin server through the switch. The last server is used to run the Metron controller. In §3.3, we study how switch diversity might affect Metron, by comparing the performance and capacity of a NoviFlow 1132 switch with an HP 5130 EI Switch [28] with software version S5130-3106, and the popular Open vSwitch [57] (OVS) software switch.

Each experiment was conducted 10 times and we report the $10^{th}$, $50^{th}$ (i.e., median), and $90^{th}$ percentiles.

## 3.1 Metron Large-Scale Deployment

In this section we test Metron's performance at scale, focusing on two aspects: First, we stress Metron's data plane performance using complex service chains with a large number of deeply-inspected (§3.1.1) and stateful (§3.1.2) traffic classes at 40 and 100 Gbps respectively. In §3.1.3 we test Metron's placement on a set of topologies with a large number of nodes, on which we deploy hundreds to thousands of service chains.

### 3.1.1 Deep Packet Inspection at 40 Gbps

To test the overall system performance at scale, we deploy a service chain of a campus firewall, followed by a DPI. The firewall implements access control using a list of 1000 rules, derived from an actual campus trace. The output of the firewall is sent to a DPI NF that uses a set of regular expressions similar to Snort (see [13]).

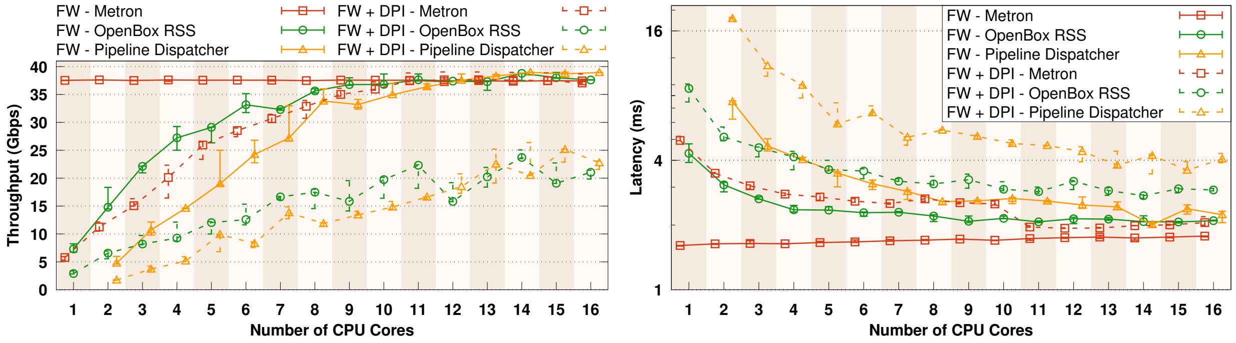
We compare Metron against two state of the art systems: $(i)$ an accelerated version of OpenBox based on RSS and $(ii)$ an emulated version of E2. In the latter case, called "Pipeline Dispatcher", we emulate E2's SoftNIC by using a dedicated CPU core (i.e., core 1) that dispatches packets to the remaining CPU cores of the server (i.e., 2-16), where the NFs of the service chain are executed. This is the reason that the graphs of the emulated E2 in Figures 6 and 7 start from core two.

We injected a campus trace, obtained from University of Liège, that exercises all the rules of the firewall at 40 Gbps and measured the performance of the three approaches. Figure 6 visualizes the results. First, we deploy only the firewall NF of this service chain to quantify the overhead of running this NF in software, as compared to an offloaded firewall (i.e., Metron). To fairly compare Metron against the other two approaches, we start a simple forwarding NF in the server, such that all packets follow the exact same path (generator, switch, server, switch, and sink) in all three experiments.

Figure 6a shows that OpenBox and the emulated E2 can realize this large firewall at line-rate. However, this is only possible if more than half of the server's CPU cores are utilized. Specifically, OpenBox requires 9 cores, while the emulated E2 requires 11 cores. In contrast, Metron completely offloads the firewall to the switch, hence easily realizing its ACL at line-rate; thus one core of the server is enough to achieve maximum throughput.

Looking at the latency of the three approaches in Figure 6b, it is evident that software-based dispatching (yellow solid line with triangles) incurs a large amount of unnecessary latency. Hardware dispatching

[5] A few thousands of tags were enough to conduct the study in §3.



**(a)** Throughput (Gbps) versus number of cores.

**(b)** Latency (ms) on a logarithmic scale versus number of cores.

**Figure 6:** Performance of a campus firewall with 1000 rules followed by a DPI at 40 Gbps, using: (*i*) Metron, (*ii*) an accelerated version of OpenBox using RSS, and (*iii*) a software-based dispatcher emulating E2.

using RSS (green solid line with circles) achieves substantially lower latency because it involves less inter-core communication. However, since the firewall executes heavy classification computations in software, OpenBox still exhibits high latency that cannot be decreased by simply increasing the number of cores. Specifically, using 16 cores has comparable latency to 4 cores. In contrast, Metron achieves nearly constant low latency (red solid line with squares) by exploiting the switch's ability to match a large number of rules at line-rate. This latency is 2.9-4.7x lower than the latency of the OpenBox and emulated E2 respectively, when each system uses one core for processing the NF (emulated E2 requires 2 cores in this case). At the full capacity of the server, the latency among the three systems is comparable; but Metron outperforms the emulated E2 and OpenBox by 30% and 19% respectively.

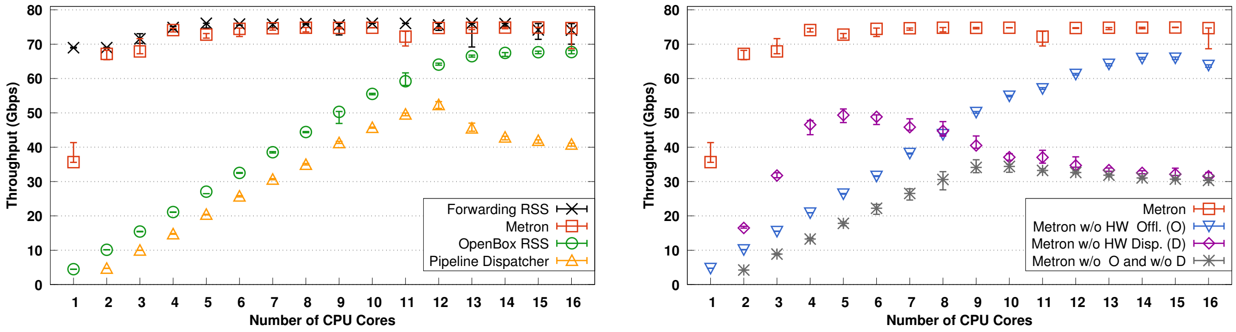
Next, we chain this firewall with a DPI NF in order to realize the entire service chain. This chaining further pushes the performance limits of the three approaches as shown by the dashed lines in Figure 6. In this case, Metron implements the DPI in software. First, we observe that even at the full capacity of the server, OpenBox and the emulated E2 can only achieve at most 25 Gbps. This performance is more than sufficient for a 10 Gbps deployment, hence some operators might not need the complex machinery of Metron. However, several studies indicate that large networks have already migrated from 10 to 40 Gbps deployments [16], while 100 Gbps networks are increasingly gaining traction [67]. In these higher data rate environments, these alternatives would require more than 16 CPU cores (and potentially more than one machine) to have sufficient throughput, and are not guaranteed to scale because of the heavy processing requirements of large service chains.

Metron exploits the joint network and server capacity to scale even complex NFs, such as DPI, at line-rate (red dashed line with squares in Figure 6a). Most importantly, Metron requires only 10 CPU cores in a single machine to achieve this result, thus substantially shifting the scaling point for large service chains. The latency results further highlight Metron's abilities. With 16 CPU cores, the Metron server deeply inspects all packets for this service chain at the cost of only 15.5% higher latency than the latency required to realize only the firewall. At the same time, OpenBox and the emulated E2 incur 35-97% more latency than Metron, while achieving almost half of Metron's throughput. This difference increases rapidly when fewer CPU cores are utilized. For example, when each system uses one CPU core Metron achieves 75% lower latency than OpenBox and 358% lower latency than the emulated E2 respectively.

### 3.1.2 Stateful Service Chaining at 100 Gbps

In this section we further stress the performance of Metron, OpenBox, and the emulated E2 systems by conducting an experiment at 100 Gbps. To achieve this new performance target we use a different testbed. We equipped two of our servers with a 100 GbE Mellanox ConnectX-4 MT27700 card and connected them back-to-back. The first server acts as a traffic generator and receiver, while the second server is the device under test.

We analyzed 4 million packets from the campus trace used in §3.1.1 and found 3117 distinct destination IP addresses. Then, we implemented a standards-compliant router and populated its routing table with these addresses. The router was chained with two stateful NFs: a NAPT and a load balancer (LB) that implements a flow-based round robin policy. In this scenario, Metron can only offload the routing table of the router to the Mellanox NIC using DPDK's flow director. The remaining functions of the router (e.g., ARP handling, IP fragmentation, TTL decrement, etc.) together with the stateful NFs (i.e., NAPT and LB) are executed in software.



**(a)** Comparison of: (*i*) Metron, (*ii*) OpenBox with RSS, and (*iii*) a software-based dispatcher emulating E2. "Forwarding RSS" shows the forwarding speed of the server (i.e., no service chain).

**(b)** Metron's hardware offloading (O) and hardware dispatching (D) contributions to the overall system's performance. The word "without" is abbreviated as "w/o".

**Figure 7:** Throughput (Gbps) of a stateful service chain (Router→NAPT→LB) at 100 Gbps.

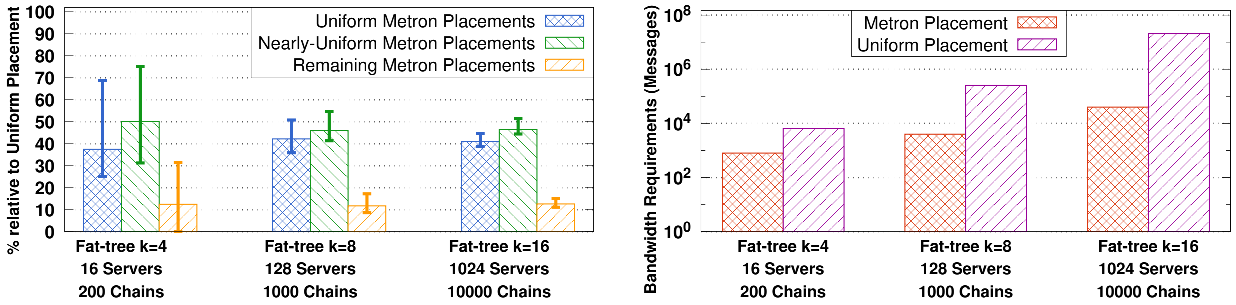
**Metron vs. State of the art:** The throughput achieved by the three systems is shown in Figure 7a. For comparison, we also show the throughput of the server when a simple RSS-assisted forwarding NF is used to send traffic back to its origin. These results show a slow but linear increase of the throughput with an increasing number of CPU cores for both OpenBox and the emulated E2 approaches. Using linear regression on the medians between 1 and 12 cores (the emulated E2 starts from 2 cores), we found that the throughput of OpenBox increases by 5.37 Gbps with each additional core, while the emulated E2 increased by 4.91 Gbps per core. However, in both cases using more than 12 CPU cores does not bring further performance gains. Specifically, the throughput of OpenBox plateaus around 67 Gbps, while the performance of the emulated E2 drops (from 53 to 41 Gbps). Moreover, with 13-16 cores, the latency of the two systems increases (up to 56% for OpenBox and up to 25% for the emulated E2); we omit the latency graph due to space limitations.

In contrast, Metron achieves 75 Gbps throughput using only a small fraction of the server's CPU cores. Key to this performance is Metron's hardware dispatcher in the NIC, which offers two advantages: (*i*) it saves CPU cycles by performing the lookup operations of the router and (*ii*) it load balances the traffic classes matched by the hardware classifier across the available CPU cores. Exploiting these advantages allows Metron (i.e., red squares in Figure 7) to quickly match the performance of the "Forwarding RSS" case (i.e., black points in Figure 7a) using only two cores, despite running several stateful operations (i.e., NAPT and LB). Moreover, according to a performance report by Mellanox [51], our NIC achieves line-rate throughput with frames greater than 512 bytes. Therefore, the 75 Gbps limit reached in this experiment with the campus trace is mainly due to the large number of small frames (26.9% of the frames are smaller than 100 bytes, while 11.8% of them are in (100, 500]). Finally, Metron's latency plateaus at a sub-millisecond level, which is 21-38% lower than the lowest latency achieved by the other two systems.

**Dissecting Metron's Performance:** To quantify the factors that contribute to Metron's high performance, we conducted an additional experiment using the same testbed, input trace, and service chain. The results are depicted in Figure 7b. Note that the red curves (i.e., Metron's throughput) in Figures 7a and 7b are identical. The purpose of Figure 7b is to showcase what performance penalties are expected when one starts removing our key contributions from Metron, as follows:

1. Metron without hardware offloading (i.e., blue triangles in Figure 7b). Hardware offloading corresponds to Contribution 1 in §1.2;
2. Metron without hardware dispatching to the correct core (purple rhombs in Figure 7b). Accurate dispatching corresponds to Contribution 2 in §1.2;
3. Metron without both of the two contributions (gray stars in Figure 7b).

Comparing "Metron" vs. "Metron w/o HW Offl." quantifies the benefits of Metron's hardware offloading feature. In the "Metron w/o HW Offl." case input packets are still dispatched to the correct core (using the Flow Director component of the Mellanox NIC), but each core executes the entire service chain logic in software. The throughput achieved in this case (i.e., blue triangles in Figure 7b) is comparable with the throughput of the "OpenBox RSS" case shown in Figure 7a. A key difference between these cases is that "Metron w/o HW Offl." performs the routing table lookup twice; once in the NIC for traffic dispatching and the second in software (to disable hardware offloading), after packets are dispatched to the correct core. In contrast, OpenBox uses RSS for dispatching and implements the routing table only once in software. Neither of these cases exploits the available capacity of the NIC to offload the routing operations, thus costing CPU cycles.



**(a)** Metron's placement relative to the uniform placement policy. Metron makes uniform or nearly uniform (with the least distance from uniform) placement decisions with ~90% median probability.

**(b)** Bandwidth requirements on a logarithmic scale with increasing number of servers and service chains. Metron requires orders of magnitude less bandwidth than the uniform placement policy.

**Figure 8:** Placement performance and bandwidth requirements on three fat-tree topologies of increasing number of servers (i.e., 16, 128, and 1024), when using (*i*) Metron or (*ii*) the uniform (equal number of CPU cores per server) placement scheme to deploy a large number of service chains.

Next, the comparison between "Metron" and "Metron w/o HW Disp." cases highlights the cost of inter-core communication. "Metron w/o HW Disp." implements the routing lookup in hardware (i.e., hardware offloading is enabled), hence reducing the processing requirements of the software part of the service chain. However, this case exhibits a serious bottleneck compared to Metron, as it requires a software component to (re-)classify input packets to decide which CPU core processes them (i.e., software dispatching similar to the emulated E2 case in Figure 7a). As shown in Figure 7, both "Metron w/o HW Disp." and the emulated E2 cases exhibit similar performance degradation as their software dispatcher communicates with an increasing number of CPU cores. This degradation appears earlier for "Metron w/o HW Disp." (i.e., after 5 cores versus 12 cores for the emulated E2 case). This is because "Metron w/o HW Disp." offloads part of the service chain's processing to the NIC, hence the inter-core communication bottleneck appears sooner. In contrast, Metron exploits the ability of the NIC to directly dispatch traffic to the correct core, thus avoiding the need for a software dispatcher and the concomitant inter-core communication.

Finally, the "Metron w/o O and w/o D" case in Figure 7b shows the throughput attainable when both hardware offloading and accurate dispatching features are disabled. In this case, input packets are always sent to an "incorrect" core (specifically the core where the software dispatcher runs) and the entire service chain runs in software. The inter-core communication bottleneck manifests itself once again, this time after using 9 or more cores.

**Key Outcome:** As explained in §2, Metron's ability to scale complex (i.e., DPI) and stateful (i.e., NAPT and LB) NFs is due to the way that the incoming traffic classes are identified, tagged, and dispatched to the CPU cores in a load balanced fashion. Metron's ability to realize these service chains at the NIC's hardware limit with a single server is an important achievement.

### 3.1.3 Metron's Placement in Large Networks

To verify that the performance of our placement scheme (see §2.3.3) can be generalized to real and potentially large networks, we conducted experiments that emulate Metron's service chain placement in datacenters, using fat-tree topologies of increasing sizes (see Figure 8). Our analytic study shows how close Metron's placement decisions are compared to uniform placement and what bandwidth requirements each approach demands for a large number of service chains. Note that the uniform placement allocates equal number of CPUs from the available servers, while a nearly uniform placement exhibits the least distance from the uniform. Note also that our approach is not restricted to datacenter topologies; Metron's placement is topology-agnostic.

Figure 8a compares Metron's placement with the uniform placement policies with increasing number of servers (i.e., 16, 128, and 1024) and service chains (i.e., 200, 1000, and 10000). The first of each set of bars indicate that Metron's placement decisions match the uniform ones with ~40% median probability, regardless of the network's size and number of service chains to be placed. For 16 servers, the upper percentile indicates that Metron makes a uniform decision with 70% probability. According to the other two sets of bars, most of the remaining decisions made by Metron fall very close to uniform (i.e., middle set of bars), confirming that our placement policy makes reasonably balanced decisions, despite its "limited" randomness.

Figure 8b shows the bandwidth savings of our placement policy, compared to the uniform one. To make a uniform placement decision, a controller has to query the CPU availability from all the available servers, thus, incurring a communication overhead proportional to the network size (which quickly becomes infeasible for large networks). This overhead is shown by the second of each set of bars in Figure 8b. To reduce this overhead, we trade-off some accuracy in placement to minimize Metron's bandwidth requirements. The first of each set of bars in Figure 8b shows that Metron requires orders of magnitude less bandwidth than the uniform policy to place a large number of service chains on these networks. An indirect (but important) benefit of our low overhead placement is that, by querying only 2 servers at a time, we generate a minimal number of events at the servers, hence preserving processing cycles for other tasks.

## 3.2 Metron's Dynamic Scaling

Next, we evaluate Metron's dynamic scaling strategy (introduced in §2.3.4) using a scenario with a service chain configuration taken from an Internet Service Provider (ISP) [65], targeting a 10 Gbps network. The service chain consists of an ACL with 725 rules, followed by a NAPT gateway that interconnects the ISP with the Internet while performing source and destination address and port translation & routing.

We deployed this service chain on a single server connected to our switch, to which a real trace was injected at variable bitrates. The solid curve in Figure 9 shows the throughput corresponding to the rate at which the trace was injected, while the dashed curve depicts the throughput achieved by Metron. To highlight Metron's ability to provision resources on demand, we plot the number of cores allocated by Metron over the course of the experiment (yellow circles and right-hand scale).

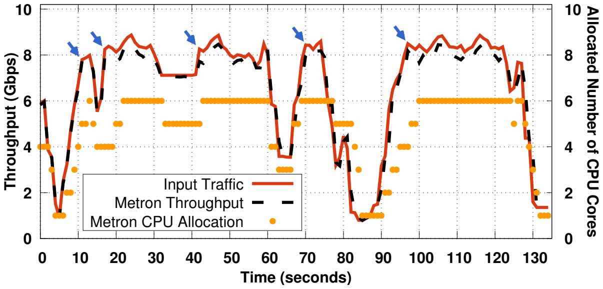


**Figure 9:** Metron under dynamic workload. Blue arrows indicate load spikes throughout the experiment.

The experiment begins with an allocation of 4 CPU cores (precalculated based upon the initial injection rate). Following this, the Metron controller makes dynamic decisions based on monitoring data gathered from the data plane and dynamically modifies the mapping of traffic classes to tags (thus affecting load distribution). In this experiment Metron requires between 1 and 6 CPU cores to accommodate the input traffic. In some cases, Metron fails to immediately adapt to sudden spikes, thus we observe a slight lag in Metron's reactions (e.g., as shown in the interval between 84 and 90 seconds). This is because our scaling approach involves interaction between the controller and the involved nodes (i.e., the server and the switch) in order to establish the CPU affinity of the traffic classes. To avoid overloading the controller, this interaction occurs every 500 ms, which contributes to the observed lag. However, Metron's throughput is not substantially affected by this lag (the blue arrows indicate the upward spikes in load at 10, 17, 42, 70, and 97 seconds). As we confirm in §3.3.2, Metron is able to quickly install the necessary rules to enforce the traffic class affinity.

## 3.3 Deployment Micro-benchmarks

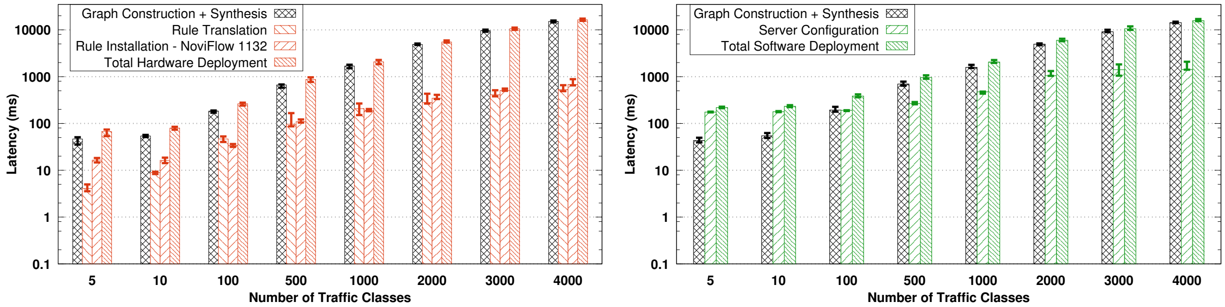
We benchmark how quickly Metron carries out important control and data plane tasks, such as hardware and software configuration, in a fully automated fashion.

### 3.3.1 Impact of Increasing # of Traffic Classes

To study the impact of increasingly complex service chains on Metron's deployment latency, we use a firewall with an increasing number of rules (up to 4000, derived from actual ISP firewalls [65]). We measure the time between when a request to deploy this NF is issued by an application and the actual NF deployment either in hardware or in software.

In either case, the first task of Metron is to construct and synthesize the packet processing graph of the service chain (as per §2.3.1), as depicted in the first of each group of bars (in black) in Figures 10a and 10b. This latency is the dominant latency in both hardware and software-based deployments (see the last set of bars in each figure). Fortunately, this is a one time overhead for each unique service chain; considering the importance of generating such an optimized processing graph, Metron precomputes and stores the synthesized graph for a given input in its distributed database.

Apart from this fixed latency operation, a purely hardware-based deployment, requires two additional operations, as shown in Figure 10a. The first operation is the automatic translation of the firewall's synthesized packet processing graph into hardware instructions targeting our OpenFlow switch (the second bar in each set of bars). This operation involves building a classification tree that encodes all the conditions of the firewall rules, therefore it has logarithmic complexity with the number of traffic classes. For example, under the specified experimental conditions, the median time to encode a large firewall with 4000 traffic classes is around 500 ms. The last operation in the hardware-based deployment is the rule installation in the OpenFlow switch (the third bar in each set of bars in Figure 10a). Note that even entry-level OpenFlow switches, such as the one used, can install thousands of rules per second;



**(a)** Hardware-based deployment on a NoviFlow 1132 switch.

**(b)** Software-based deployment on a 16-core Intel Xeon E5-2667.

**Figure 10:** Latency (ms) on a logarithmic scale for different Metron deployments with increasing complexity.

a more thorough study is provided in §3.3.2, where we discuss the effects of hardware diversity on Metron.

For a purely software-based deployment of this same service chain, we consider the time following graph construction and synthesis until the chain is deployed at a designated server. This latency is labeled "Server Configuration" in Figure 10b. Note that it takes longer per rule than for the corresponding hardware-based case for a small number of traffic classes because there is a fixed overhead to start a secondary DPDK process (i.e., a Metron slave) at the server. This overhead is ~180 ms as can be seen from the case of 5 traffic classes. However, the (median) deployment time is 0.471 ms/rule (versus 0.411 ms/rule for the hardware case shown in Table 2), hence a large firewall deployment takes a comparable amount of time either in software or hardware.

Overall, apart from the one-time precomputation overhead for constructing and synthesizing a service chain, the worst case deployment time of a firewall with 4000 traffic classes is less than 1200 ms, whereas only 100-200 ms is required for hundreds of traffic classes.

### 3.3.2 Diversity of Network Elements' Capabilities

Network elements from different vendors and of different price levels might offer different possibilities for NFV offloading. In this section we repeat the hardware-based deployment shown in Figure 10a, where we replace the NoviFlow switch with either a hybrid HP 5130 El hardware switch or the software-based OVS. Table 2 summarizes the results along with key characteristics of these switches, as they affect Metron's deployment choices and performance.

The NoviFlow switch contains 55 OpenFlow tables, each with a capacity of 4096 entries (i.e., 225280 rules in total), while the HP switch offers a single OpenFlow table with either 512/256 entries for IPv4/IPv6-based rules or 16384 entries for L2 rules. The capacity of OVS depends on the amount of memory that the host machine provides; modern servers provide ample DRAM capacity to store millions of rules.

The median rule installation speed of the NoviFlow switch is substantially higher than HP (0.411 vs. 50.25 ms/rule), with the difference being more than two orders of magnitude. However, this difference is partially reflected in the price difference between the two switches (approximately US$ 15000 vs. US$ 2000). OVS is open source, achieves lower data plane performance, but outperforms both hardware-based switches in terms of median rule installation speed (0.263 ms/rule), when running on the processor described for the testbed in §3. This finding is confirmed by earlier studies [45, 46], where the rule installation speed varied especially when priorities are involved. In our test, Metron installed rules of the same priority and we observed low variance.

In summary, today's OpenFlow switches provide Metron with fast median rule installation speed and sufficient capacity at different price/performance levels.

**Table 2:** Comparison of 3 switches used by Metron. The last column states their median rule installation speed.

| Switch | | Capacity | Speed |
|---|---|---|---|
| **Model** | **Type** | **(Rules)** | **(ms/rule)** |
| NoviFlow 1132 [53] | HW | 225280 | 0.411 |
| HP 5130 El [28] | HW | 256/512 /16384 | 50.250 |
| OVS [57] v2.5.2 | SW | Memory -bound | 0.263 |

## 4 Related Work

Here, we discuss related efforts beyond the work mentioned inline throughout this paper.

**NFV Management:** E2 [59] and Metron manage service chains mapped to clusters of servers interconnected via programmable switches. E2 only partially exploits OpenFlow switches to perform traffic steering. In contrast, Metron fully exploits the network (i.e., OpenFlow switches and NICs) to both steer traffic and to offload and load balance NFV service chains, while deliberately avoiding E2's inter-core transfers.

**NFV Consolidation:** OpenBox [13] merges similar packet processing elements into one, thus reducing redundancy. SNF [36] eliminates processing redundancy by synthesizing multiple NFs as an optimized equivalent NF. Slick [3] and CoMb [62] propose NF consolidation schemes, although these schemes reside higher in the network stack. We integrated SNF into Metron, since this is the most extensive consolidation scheme to date. Metron effectively coordinates these optimized pipelines at a large-scale, while exploiting the hardware.

**Hardware Programmability:** During the last decade, there has been a large effort to increase hardware programmability. OpenFlow [50] paved the way by enriching the programmability of switches using simple match-action rules. Increasingly, NICs are equipped with hardware components, such as RSS and Flow Director, for dispatching packets from NIC to CPU core.

In an attempt to overcome the static nature of the above solutions, more flexible programmability models have emerged. RMT [12] and its successor P4 [11] are prime examples of protocol-independent packet processors, while OpenState [8] and OPP [9] showed how OpenFlow can become stateful with minimal but essential modifications. FlexNIC [38] proposed a model for additional programmability in future NICs.

All these works have made phenomenal progress towards exposing hardware configuration knobs. Metron acts as an umbrella to foster the integration of this diverse set of programmable devices into a common management plane. In fact, our prototype integrates OpenFlow switches, DPDK-compatible NICs, and servers. Thanks to ONOS's abstractions, additional network drivers can be easily integrated.

**Hardware Offloading:** Raumer et al. [61] offloaded the cryptographic function of a virtual private network (VPN) gateway into commodity NICs, for increased performance. SwitchKV [48] offloads key-value stores into OpenFlow switches. PacketShader [26], Kargus [32], NBA [41], and APUNet [24] take advantage of inexpensive but powerful graphical processing units to offload and accelerate packet processing. We envision these works as future components of Metron to extend its offloading abilities.

ClickNP [47] showed how to achieve high performance packet processing by completely migrating NFV into reconfigurable, but specialized hardware. In contrast, our philosophy is to explore the boundaries of commodity hardware. Therefore, Metron performs stateful processing in software but combines it with smart offloading using commodity hardware.

**Server-level Solutions:** Flurries [70] builds atop OpenNetVM [71] to provide software-based service chains on a per-flow basis, while ClickOS [49] and NetVM [29] offer NFs running in VMs. NFP [63] extends OpenNetVM to allow NFs in a service chain to be executed in parallel. Dysco [69] proposes a distributed protocol for steering traffic across the NFs of a service chain. NFVnice [44] and SCC [37, 34] are efficient NFV schedulers. Click-based [42] approaches have proposed techniques to exploit multi-core architectures [6, 64, 40]. None of these works have explored the possibility of using hardware to offload parts of a service chain, nor do they support our optimized flow affinity approach.

**Industrial Efforts:** European Telecommunications Standards Institute (ETSI) has been driving NFV standardization during the last 5 years [21]. ETSI's specialized group [22] uses OpenStack [58] as an open implementation of the current NFV standards, based on a generic framework for managing compute, storage, and network resources. CORD [55] and OPNFV [66] also use OpenStack. The former re-architects the central office as a datacenter, while the latter facilitates the interoperability of NFV components across various open source ecosystems. Metron and CORD share common controller abstractions (i.e., ONOS); however, we avoid OpenStack's virtualization by integrating native, DPDK-based solutions. Unlike CORD, our controller leverages placement techniques with minimal overhead (see §2.3.3 and §3.3) and sophisticated NF consolidation (see §2.3.1) to achieve high performance.

## 5 Conclusion

We have presented Metron, an NFV platform that fundamentally changes how service chains are realized. Metron eliminates the need for costly inter-core communication at the servers by delegating packet processing and CPU core dispatching operations to programmable hardware devices. Doing so offers dramatic hardware efficiency and performance increases over the state of the art. With commodity hardware assistance, Metron fully exploits the processing capacity of a single server, to deeply inspect traffic at 40 Gbps and execute stateful service chains at the speed of a 100 GbE NIC.

## 6 Acknowledgments

We would like to thank our shepherd Vyas Sekar and the anonymous reviewers for their insightful comments on this paper. This work is financially supported by the Swedish Foundation for Strategic Research. In addition, this work was partially supported by the Wallenberg Autonomous Systems Program (WASP).

# References

[1] Al-Fares, M., Loukissas, A., and Vahdat, A. A Scalable, Commodity Data Center Network Architecture. In Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (2008), pp. 63–74.

[2] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (2010), NSDI'10.

[3] Anwer, B., Benson, T., Feamster, N., and Levin, D. Programming Slick Network Functions. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (2015), SOSR '15, pp. 14:1–14:13.

[4] Barbette, T. Repository with DPDK extensions for OpenBox, 2018. https://github.com/tbarbette/fastclick/tree/openbox.

[5] Barbette, T., and Katsikas, G. P. Metron data plane, 2018. https://github.com/tbarbette/fastclick/tree/metron.

[6] Barbette, T., Soldani, C., and Mathy, L. Fast Userspace Packet Processing. In Proceedings of the 11th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (2015), ANCS '15, IEEE Computer Society. http://dl.acm.org/citation.cfm?id=2772722.2772727.

[7] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., and Parulkar, G. ONOS: Towards an Open, Distributed SDN OS. In Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking (2014), HotSDN '14, pp. 1–6.

[8] Bianchi, G., Bonola, M., Capone, A., and Cascone, C. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. SIGCOMM Comput. Commun. Rev. (2014).

[9] Bianchi, G., Bonola, M., Pontarelli, S., Sanvito, D., Capone, A., and Cascone, C. Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing. arXiv preprint arXiv:1605.01977 (2016).

[10] Bjorklund, M. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). Internet Request for Comments (RFC) 6020 (Proposed Standard), Oct. 2010. https://www.rfc-editor.org/rfc/rfc6020.txt.

[11] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming Protocol-independent Packet Processors. SIGCOMM Comput. Commun. Rev. 44, 3 (July 2014), 87–95.

[12] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In Proceedings of the ACM SIGCOMM 2013 Conference on (2013), pp. 99–110.

[13] Bremler-Barr, A., Harchol, Y., and Hay, D. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference (2016), SIGCOMM '16, pp. 511–524.

[14] Case, J., Fedor, M., Schoffstall, M. L., and Davin, J. Simple Network Management Protocol (SNMP). Internet Request for Comments (RFC) 1157, May 1990. http://www.ietf.org/rfc/rfc1157.txt.

[15] Chowdhury, M., Rahman, M. R., and Boutaba, R. ViNEYard: Virtual Network Embedding Algorithms with Coordinated Node and Link Mapping. IEEE/ACM Trans. Netw. 20, 1 (Feb. 2012), 206–219.

[16] Cisco. Migrate to a 40-Gbps Data Center with Cisco QSFP BiDi Technology, 2013. http://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-729493.html.

[17] Dietz, T., Bifulco, R., Manco, F., Martins, J., Kolbe, H., and Huici, F. Enhancing the BRAS through virtualization. In Proceedings of the 1st IEEE Conference on Network Softwarization, NetSoft 2015 (2015), pp. 1–5.

[18] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. RouteBricks: Exploiting Parallelism to Scale Software Routers. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (2009), SOSP '09, pp. 15–28.

[19] DPDK. Data Plane Development Kit, 2018. http://dpdk.org.

[20] Enns, R., Bjorklund, M., Schoenwaelder, J., and Bierman, A. Network Configuration Protocol (NETCONF). Internet Request for Comments (RFC) 6241 (Proposed Standard), June 2011. Updated by RFC 7803, https://www.rfc-editor.org/rfc/rfc6241.txt.

[21] European Telecommunications Standards Institute. Network Functions Virtualisation, 2017. http://www.etsi.org/technologies-clusters/technologies/689-network-functions-virtualisation.

[22] European Telecommunications Standards Institute (ETSI). Open Source NFV Management and Orchestration (MANO) , 2018. https://osm.etsi.org/.

[23] Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., and Akella, A. OpenNF: Enabling Innovation in Network Function Control. In Proceedings of the 2014 ACM Conference on SIGCOMM (2014), SIGCOMM '14, pp. 163–174.

[24] Go, Y., Asim Jamshed, M., Moon, Y., Hwang, C., and Park, K. APUNet: Revitalizing GPU as Packet Processing Accelerator. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17) (2017), USENIX Association, pp. 83–96.

[25] Han, S., Jang, K., Panda, A., Palkar, S., Han, D., and Ratnasamy, S. SoftNIC: A Software NIC to Augment Hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[26] Han, S., Jang, K., Park, K., and Moon, S. PacketShader: A GPU-accelerated Software Router. In Proceedings of the ACM SIGCOMM 2010 Conference (2010), SIGCOMM '10, pp. 195–206.

[27] He, J., Zhang-Shen, R., Li, Y., Lee, C.-Y., Rexford, J., and Chiang, M. DaVinci: Dynamically Adaptive Virtual Networks for a Customized Internet. In Proceedings of the 2008 ACM CoNEXT Conference (New York, NY, USA, 2008), CoNEXT '08, ACM, pp. 15:1–15:12.

[28] Hewlett Packard. HPE FlexNetwork 5130 EI Switch Series, Jan. 2017. https://h50146.www5.hpe.com/products/networking/datasheet/HP_5130EI_Switch_Series_J.pdf.

[29] Hwang, J., Ramakrishnan, K. K., and Wood, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (2014), NSDI'14, pp. 445–458.

[30] Intel. Receive-Side Scaling (RSS), 2007. http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf.

[31] Intel. Ethernet Flow Director, 2018. http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-flow-director-video.html.

[32] Jamshed, M. A., Lee, J., Moon, S., Yun, I., Kim, D., Lee, S., Yi, Y., and Park, K. Kargus: A Highly-scalable Software-based Intrusion Detection System. In Proceedings of the 2012 ACM Conference on Computer and Communications Security (2012), CCS '12.

[33] Kablan, M., Alsudais, A., Keller, E., and Le, F. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17) (2017), pp. 97–112.

[34] Katsikas, G. P. Realizing High Performance NFV Service Chains. Licentiate Thesis (Nov. 2016). TRITA-ICT 2016:35, http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1044355&dswid=-1520.

[35] Katsikas, G. P. Metron controller's southbound driver for managing commodity servers, 2018. https://github.com/gkatsikas/onos/tree/metron-driver.

[36] Katsikas, G. P., Enguehard, M., Kuźniar, M., Maguire Jr., G. Q., and Kostić, D. SNF: Synthesizing high performance NFV service chains. PeerJ Computer Science 2 (Nov. 2016), e98. http://dx.doi.org/10.7717/peerj-cs.98.

[37] Katsikas, G. P., Maguire Jr., G. Q., and Kostić, D. Profiling and accelerating commodity NFV service chains with SCC. Journal of Systems and Software 127C (Feb. 2017), 12–27. https://doi.org/10.1016/j.jss.2017.01.005.

[38] Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., and Krishnamurthy, A. High Performance Packet Processing with FlexNIC. In Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (2016), ASPLOS '16, pp. 67–81.

[39] Khalid, J., Gember-Jacobson, A., Michael, R., Abhashkumar, A., and Akella, A. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (2016), NSDI'16, USENIX Association, pp. 239–253.

[40] Kim, J., Huh, S., Jang, K., Park, K., and Moon, S. The Power of Batching in the Click Modular Router. In Proceedings of the Asia-Pacific Workshop on Systems (2012), APSYS '12, pp. 14:1–14:6.

[41] Kim, J., Jang, K., Lee, K., Ma, S., Shim, J., and Moon, S. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In Proceedings of the 10th European Conference on Computer Systems (2015), EuroSys '15.

[42] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click Modular Router. ACM Trans. Comput. Syst. 18, 3 (Aug. 2000), 263–297.

[43] Krishnan, R., Durrani, M., and Phaal, P. Real-time SDN Analytics for DDoS mitigation, 2014.

[44] Kulkarni, S. G., Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K. K., Wood, T., Arumaithurai, M., and Fu, X. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 71–84.

[45] Kuźniar, M., Perešíni, P., and Kostić, D. What You Need to Know About SDN Flow Tables. In Passive and Active Measurement (PAM) (2015), vol. 8995 of Lecture Notes in Computer Science, pp. 347–359. https://doi.org/10.1007/978-3-319-15509-8_26.

[46] Kuźniar, M., Perešíni, P., Kostić, D., and Canini, M. Methodology, Measurement and Analysis of Flow Table Update Characteristics in Hardware OpenFlow Switches. Computer Networks: The International Journal of Computer and Telecommunications Networking, Elsevier, vol. 26 (2018). https://doi.org/10.1016/j.comnet.2018.02.014.

[47] Li, B., Tan, K., Luo, L. L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In Proceedings of the 2016 ACM SIGCOMM Conference (2016), SIGCOMM '16, pp. 1–14.

[48] Li, X., Sethi, R., Kaminsky, M., Andersen, D. G., and Freedman, M. J. Be Fast, Cheap and in Control with SwitchKV. In Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (2016), NSDI'16, USENIX Association, pp. 31–44.

[49] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. ClickOS and the Art of Network Function Virtualization. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (2014), NSDI'14, pp. 459–473.

[50] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. SIGCOMM Comput. Commun. Rev. 38, 2 (Mar. 2008), 69–74.

[51] Mellanox Technologies. Mellanox NIC's Performance Report with DPDK 17.05, 2017. Document number MLNX-15-52365, Revision 1.0, 2017, http://fast.dpdk.org/doc/perf/DPDK_17_05_Mellanox_NIC_performance_report.pdf.

[52] Mitzenmacher, M. The Power of Two Choices in Randomized Load Balancing. IEEE Trans. Parallel Distrib. Syst. 12, 10 (Oct. 2001), 1094–1104.

[53] NoviFlow. NoviSwitch 1132 High Performance OpenFlow Switch, 2013. https://noviflow.com/wp-content/uploads/NoviSwitch-1132-Datasheet-V2_1.pdf.

[54] Olteanu, V. A., and Raiciu, C. Efficiently Migrating Stateful Middleboxes. In Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (2012), SIGCOMM '12, ACM, pp. 93–94.

[55] ON.Lab. Central Office Re-architected as a Datacenter (CORD), 2018. http://opencord.org/.

[56] ON.Lab. Open Network Operating System (ONOS), 2018. http://onosproject.org/.

[57] Open vSwitch. An Open Virtual Switch, 2018. http://openvswitch.org.

[58] OpenStack. Open Source Cloud Computing Software, 2018. https://www.openstack.org/.

[59] Palkar, S., Lan, C., Han, S., Jang, K., Panda, A., Ratnasamy, S., Rizzo, L., and Shenker, S. E2: A Framework for NFV Applications. In Proceedings of the 25th Symposium on Operating Systems Principles (2015), SOSP '15, pp. 121–136.

[60] Pesterev, A., Zeldovich, N., and Morris, R. T. Locating Cache Performance Bottlenecks Using Data Profiling. In Proceedings of the 5th European Conference on Computer Systems (2010), EuroSys '10, pp. 335–348.

[61] Raumer, D., Gallenmüller, S., Emmerich, P., Märdian, L., Wohlfart, F., and Carle, G. Efficient serving of VPN endpoints on COTS server hardware. In 2016 IEEE 5th International Conference on Cloud Networking (CloudNet'16) (Pisa, Italy, Oct. 2016).

[62] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. Design and Implementation of a Consolidated Middlebox Architecture. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (2012), NSDI'12.

[63] Sun, C., Bi, J., Zheng, Z., Yu, H., and Hu, H. NFP: Enabling Network Function Parallelism in NFV. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 43–56.

[64] Sun, W., and Ricci, R. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (Piscataway, NJ, USA, 2013), ANCS '13, IEEE Press, pp. 25–36.

[65] Taylor, D. E., and Turner, J. S. ClassBench: A Packet Classification Benchmark. IEEE/ACM Trans. Netw. 15, 3 (June 2007), 499–511.

[66] The Linux Foundation. Open Platform for NFV (OPNFV), 2018. https://www.opnfv.org/.

[67] Viejo, A. QLogic and Broadcom First to Demonstrate End-to-End Interoperability for 25Gb and 100Gb Ethernet, 2015. https://globenewswire.com/news-release/2015/01/27/700249/10116850/en/QLogic-and-Broadcom-First-to-Demonstrate-End-to-End-Interoperability-for-25Gb-and-100Gb-Ethernet.html.

[68] Yu, M., Yi, Y., Rexford, J., and Chiang, M. Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration. SIGCOMM Comput. Commun. Rev. 38, 2 (Mar. 2008), 17–29.

[69] Zave, P., Ferreira, R. A., Zou, X. K., Morimoto, M., and Rexford, J. Dynamic Service Chaining with Dysco. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 57–70.

[70] Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K., and Wood, T. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In Proceedings of the 12th ACM International Conference on Emerging Networking Experiments and Technologies (2016), CoNEXT '16, pp. 3–17.

[71] Zhang, W., Liu, G., Zhang, W., Shah, N., Lopreiato, P., Todeschi, G., Ramakrishnan, K., and Wood, T. OpenNetVM: A Platform for High Performance Network Service Chains. In Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (August 2016), ACM.