Metron: High Performance NFV Service Chaining Even in the Presence of Blackboxes

GEORGIOS P. KATSIKAS[*], KTH Royal Institute of Technology, Sweden
TOM BARBETTE[†], KTH Royal Institute of Technology, Sweden
DEJAN KOSTIĆ, KTH Royal Institute of Technology, Sweden
GERALD Q. MAGUIRE JR., KTH Royal Institute of Technology, Sweden
REBECCA STEINERT, RISE, Sweden

Deployment of 100 Gigabit Ethernet (GbE) links challenges the packet processing limits of commodity hardware used for Network Functions Virtualization (NFV). Moreover, realizing chained network functions (i.e., service chains) necessitates the use of multiple CPU cores, or even multiple servers, to process packets from such high speed links.

Our system Metron jointly exploits the underlying network and commodity servers' resources: (*i*) to offload part of the packet processing logic to the network, (*ii*) by using smart tagging to setup and exploit the affinity of traffic classes, and (*iii*) by using tag-based hardware dispatching to carry out the remaining packet processing at the speed of the servers' cores, with *zero* inter-core communication. Moreover, Metron transparently integrates, manages, and load balances proprietary "blackboxes" together with Metron service chains.

Metron realizes stateful network functions at the speed of 100 GbE network cards on a single server, while elastically and rapidly adapting to changing workload volumes. Our experiments demonstrate that Metron service chains can coexist with heterogeneous blackboxes, while still leveraging Metron's accurate dispatching and load balancing. In summary, Metron has (*i*) 2.75-8x better efficiency, up to (*ii*) 4.7x lower latency, and (*iii*) 7.8x higher throughput than OpenBox, a state of the art NFV system.

Additional Key Words and Phrases: NFV, service chains, hardware offloading, tagging, accurate dispatching, elasticity, load balancing, blackboxes, 100 GbE

ACM Reference Format:
Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Gerald Q. Maguire Jr., and Rebecca Steinert. 2021. Metron: High Performance NFV Service Chaining Even in the Presence of Blackboxes . *ACM Trans. Comput. Syst.* 1, 1, Article 1 (May 2021), 44 pages. https://doi.org/10.1145/3465628

[*]Part of this work was done when Georgios P. Katsikas was at RISE, Sweden
[†]Part of this work was done when Tom Barbette was at the University of Liège, Belgium

Authors' addresses: Georgios P. Katsikas, KTH Royal Institute of Technology, Kistagången 16, Kista, Stockholm, SE-164 40, Sweden, katsikas@kth.se; Tom Barbette, KTH Royal Institute of Technology, Kistagången 16, Kista, Stockholm, SE-164 40, Sweden, barbette@kth.se; Dejan Kostić, KTH Royal Institute of Technology, Kistagången 16, Kista, Stockholm, SE-164 40, Sweden, dmk@kth.se; Gerald Q. Maguire Jr., KTH Royal Institute of Technology, Kistagången 16, Kista, Stockholm, SE-164 40, Sweden, maguire@kth.se; Rebecca Steinert, RISE, Kistagången 16, Kista, Stockholm, SE-164 40, Sweden, rebecca.steinert@ri.se.

0734-2071/2021/5-ART1
https://doi.org/10.1145/3465628

1 Introduction

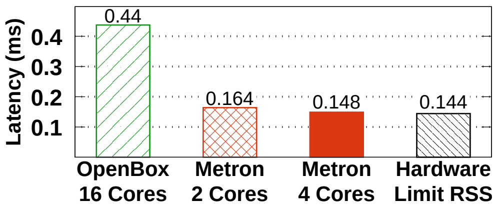
Following the success of Software-Defined Networking (SDN), Network Functions Virtualization (NFV) is poised to dramatically change the way network services are deployed. NFV advocates running chains of Network Functions (NFs) implemented as software on top of commodity hardware. This is in contrast with chaining expensive, physical middleboxes, and brings numerous benefits, such as: (*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 Gigabits per second (Gbps) switches and Network Interface Cards (NICs) are starting to be standardized and deployed [23, 27, 29, 97], maintaining high performance at the ever-increasing data rates is fundamental for the success of NFV. To quantify the challenging mission of NFV for line-rate 100 Gigabit Ethernet (GbE) networking, the available time to process a 64-byte frame at 100 Gbps is only 6.72 nanoseconds.

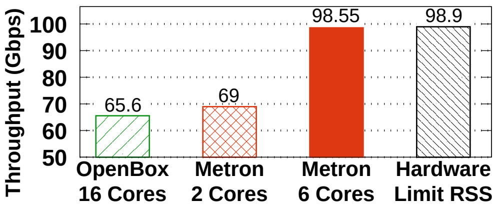
In an NFV service chain, packets move from one physical or virtual server (hereafter simply called server) to another to realize a programmable data plane. The servers themselves are predominantly multi-core machines. Different ways of structuring the NFs exist, e.g., one per physical core or using multiple threads to leverage multiple cores within each NF. NFs range from simple stateless (e.g., forwarding) to complex NFs, such as Deep Packet Inspection (DPI), and potentially stateful (e.g., proxy) NFs. Regardless of the deployment model and types of NF, 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.

Our work, Metron (originally presented in [48]), eliminates unnecessary inter-core transfers while exploiting the underlying hardware. In a 100-Gbps setup (see Figure 1) Metron achieves:

(1) a factor of 8 better efficiency,
(2) almost 3x lower and predictable latency, and
(3) 50% higher throughput than the OpenBox [16] data plane, a state of the art NFV system.



(a) Latency (ms).



(b) Throughput (Gbps).

Fig. 1. Thanks to zero inter-core transfers and the hardware exploitation, Metron has 8x better efficiency than the state of the art when realizing stateful (Router→Monitor→LB) packet processing at 100 Gbps.

1.1 NFV Processing Challenges

To identify the core that will process an incoming packet, the NFV framework typically can 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.

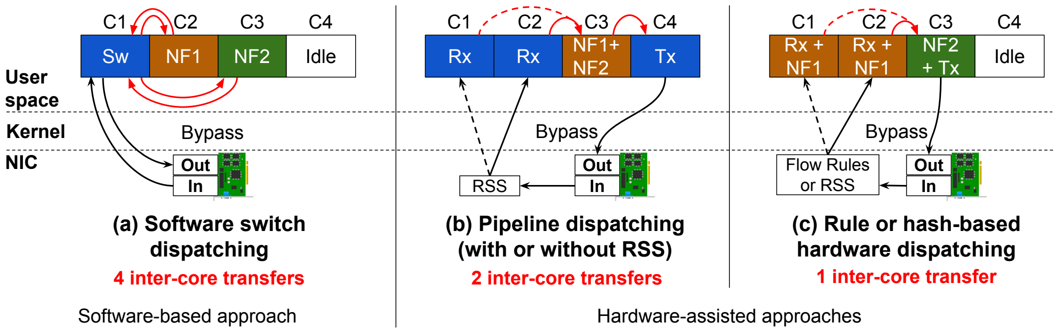


Fig. 2. State of the art packet processing models either have too many inter-core packet transfers or load balancing problems due to load imbalance and/or idle CPU cores. Receive-Side Scaling is abbreviated as RSS.

Software-based Dispatching

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 [90], with its software component called SoftNIC [31], falls into this category. SoftNIC requires at least one dedicated Central Processing Unit (CPU) core for traffic dispatching and steering (see Figure 2a), while the NFs run on other CPU cores. Earlier works, such as ClickOS [66] and NetVM [35], also used software switches on dedicated cores to dispatch packets to a Virtual Machine (VM), but without the flexibility of E2.

Pipeline Dispatching (with or without 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) utilizes 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 Receive-Side Scaling (RSS) [36] as described below. For example, OpenNetVM [115], Flurries [114], and NFP [100] (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.

Hardware-assisted Dispatching using RSS or Flow Rules

The last category of packet processing models (see Figure 2c) 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 rule-based vendor-specific "match-action" Application Programming Interface (API) to dispatch traffic to specific NIC hardware queues associated with designated CPU cores. Intel's Ethernet Flow Director [95] and the Mellanox Accelerated Switching and Packet Processing (ASAP^2) [68] are examples of this technology. The Data Plane Development Kit (DPDK) flow API [104] abstracts such multi-vendor technologies to offer a unique rule API for all DPDK-based NICs. Unlike all previous models, neither RSS nor the rule-based approaches require dedicated dispatchers, hence they achieve higher performance. OpenBox[*] [16], FastClick [8], Synthesized Network Functions (SNF) [49], and RouteBricks [22] use RSS, while CoMb [98] uses Flow Director.

Summary

None of these schemes guarantee that the core that receives an incoming packet will be the one processing it. Flow hashing as in RSS can introduce serious load imbalances under skewed workloads (due to flows with the same hashes). Rule-based flow dispatching permits explicit flow affinity,

[*]An accelerated version of the OpenBox data plane is used in this article, taken from the Metron conference paper [48].

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 the Last Level Cache (LLC) or Dynamic Random-Access Memory (DRAM) to the target processing core. Recent studies on modern servers have shown that LLC and DRAM transfers take several nanoseconds (up to 14.3 ns for LLC and up to 71.7 ns for DRAM on an Intel Xeon E5-2667 v3) [44]. These access latencies are far from the target 6.72 ns/64-byte packet to achieve line-rate processing at 100 Gbps. Therefore, there is a clear mismatch between the processing requirements of high-speed networks and the way that existing NFV systems process packets. Our earlier work [50] demonstrated that dramatic speedups (with several times lower latency and orders of magnitude lower latency variance) occur if the correct core receives the packet straight from the NIC and the packet remains in the core-specific cache(s).

1.2 Metron Research Contributions

Metron is a system for NFV service chain placement, request dispatching, and dynamic scaling. 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 [90], 2 [114], or 1 [16] inter-core transfers as shown in Figure 2), making it possible to process packets potentially *at L1 cache speeds*. Also, by combining smart identification, tagging, and dispatching techniques we overcome the load balancing issues of "run-to-completion" approaches [8, 16, 22, 49].

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 (both switches and 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. Finally, addressing all the above challenges, while allowing network operators to transparently integrate proprietary closed-source NFs (also known as "blackboxes") is extremely challenging. Our research contributions, while addressing the aforementioned challenges, are:

Contribution 1

We orchestrate programmable network's hardware to perform stateless traffic processing and classification. We deal with hardware heterogeneity by building upon the unified management abstractions of an industrial-grade SDN controller called Open Network Operating System (ONOS) [9]. This allows Metron to leverage state of the art management protocols, such as OpenFlow [67] and Programming Protocol-Independent Packet Processors (P4) [14], and easily integrate future ones. We contributed a new network driver [43] and configuration protocol [45] (also described in Appendix A) for programmable NICs and servers to ONOS.

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 [49] to identify the traffic classes of a service chain and produce a synthesized NF that performs the equivalent work of the entire service 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 relevant stateless operations, while dispatching incoming packets to those CPU cores that execute their associated 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 rapid service chain placement decisions at low cost and with high accuracy (see §2.3.3).
Contribution 4
We solve an important problem of the networking industry by enabling the coexistence of "blackbox" packet processing applications with Metron service chains. Metron provides both software and hardware-based mechanisms that allow network operators to transparently integrate, manage, and load balance blackbox NFs, the binaries of which can be deployed as native processes or inside containers/VMs. Metron's blackbox integration strategy is presented in §3.
Contribution 5
We exploit the tags, inserted by Metron's hardware dispatcher, to coordinate load balancing among the CPU cores of NFV servers, by scaling packet processing at a finer level of granularity: the level of a traffic class. The dynamic scaling approach of Metron is presented in §4.

To the best of our knowledge, Metron is the first work that deeply studies both performance and scaling aspects of NFV service chains at the challenging link speed of 100 Gbps. We envision Metron as an industrial solution, co-existing with heterogeneous open and/or proprietary NFs, therefore we encourage the networking community to use and contribute to our open source prototype [5, 45, 46].
Evaluation Summary
Metron realizes deep packet inspection at 40 Gbps (§5.3.1) and stateful service chains at the speed of 100 GbE NICs on a single server (§5.3.2). This results in up to 4.7x lower latency, up to 7.8x higher throughput, and 2.75-8x better efficiency than the state of the art. In §5.5 we show how effectively Metron scales packet processing on demand, even under highly-variable workloads up to 100 Gbps. In §5.4 we demonstrate a practical integration of a blackbox NF between two Metron service chains. In this experiment Metron load balances traffic to the blackbox NF using: (*i*) a native RSS-based dispatcher and (*ii*) Single Root I/O Virtualization (SR-IOV) [13] for NIC to VM dispatching. In both cases, Metron realizes two service chains with a blackbox NF at the speed of the underlying 100 GbE testbed. It is difficult to improve on this performance unless we completely offload stateful service chains to hardware, which is impossible with today's commodity equipment.

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 communicates with the Metron controller (§2.3) through the Metron protocol (Appendix A). In §2.4 we explain how Metron deals with routing and failures.

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 [†] of the service chain, by parsing the packet processing graphs of the input NFs. Each graph has a set of packet processing elements as in [16, 58, 90]. 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

[†] A traffic class is a (set of) flow(s) treated identically by a service chain of NFs.

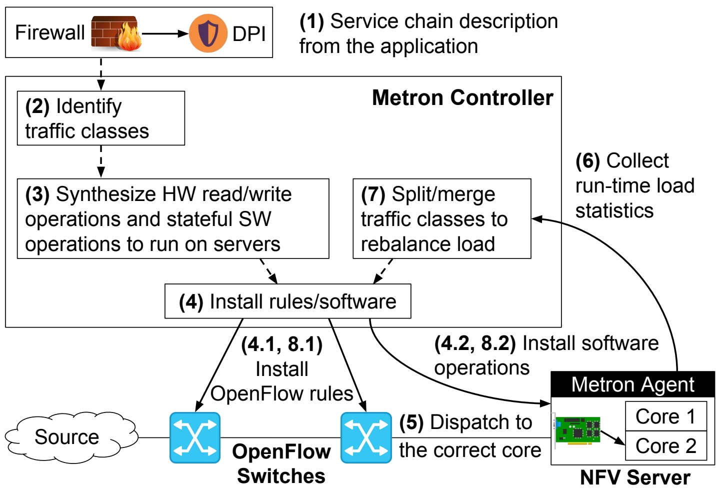

Firewall
DPI
(1) Service chain description from the application
Metron Controller
(2) Identify traffic classes
(3) Synthesize HW read/write operations and stateful SW operations to run on servers
(7) Split/merge traffic classes to rebalance load
(6) Collect run-time load statistics
(4) Install rules/software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

Fig. 3. Metron overview using an example Firewall→DPI service chain.

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). 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 specific core, thus *eliminating* costly inter-core communications. This guarantee is maintained even when a CPU core becomes overloaded (see §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 widely used NFs; noting that in Table 1 a substantial portion of these NFs can be (fully or partially) offloaded to commodity hardware. Hybrid NFs can be either stateful or stateless (thus offloadable), depending on the needs of the network operator.

Table 1. Survey of widely used 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 [21]
Evolved Packet Core	Partially
Intrusion Detection/Prevention	Partially [40]
Load Balancer	Hybrid
Flow Monitor	Yes
DDoS Detection/Prevention	Yes [59]
Congestion Control (RED, ECN)	Yes
Deep Packet Inspection	No
IP Security, Virtual Private Network	Yes [93]

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. This channel is described in Appendix A.

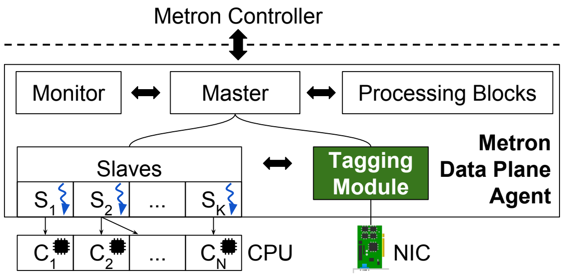


Fig. 4. The Metron data plane.

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.

When the master boots, it configures the hardware and then registers with the controller by advertising the server's available resources and tags. Now the master awaits controller instructions. As will be described in Appendix A, one such instruction is an "SC_DEPLOY_REQ" with its corresponding software configuration for a target server (see Table 4). The master executes this instruction by spawning a slave process that is pinned to the requested core(s) and by passing the processing graph described in the software configuration 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 service chain. Such graphs can be implemented either in hardware or software. Earlier works [114, 115] implement these graphs in software and use metadata to share information among NFs and to define the next hop in a service 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. We explain how to solve this problem in §2.3.

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[49]; a more aggressive variant of OpenBox for merging packet processing graphs, which provides a heuristic for solving the graph embedding problem (see [18, 33, 112]) in the context of NFV. Metron uses SNF to eliminate redundant processing 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 processing 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 (Φ) to a set of operations (Ω). In §4 we use this feature to automatically scale packet processing in and out, providing greater elasticity than possible 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 [50], or by using more generic profilers, such as DProf [91]. Even in the absence of accurate resource requirements, Metron dynamically adapts to the input load as discussed in §4.

2.3.3 Scalable Placement with Minimal Overhead Metron needs to decide where to place the synthesized packet processing graph. Such a decision is not simple, because Metron considers both a pool of servers and a pool of heterogeneous network elements, such as (programmable) switches, routers, and server NICs, 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.
Key contribution: To address this issue we solve a graph separation problem, which allows Metron to traverse and *split* the synthesized graph into two subgraphs: (*i*) a *stateless* subgraph that contains those packet filters and operations that can be completely offloaded to the network and (*ii*) a *stateful* subgraph, deployed on a server, to perform the remaining stateful operations. Packets exiting the stateless subgraph are *tagged* in a way that allows efficient (with zero inter-core communication) traversal of the stateful subgraph. 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. Metron's placement scheme deals with logical servers and network elements, hence allowing further partitioning of the graphs when no single server or network element has sufficient resources. Future work will allow Metron to prevent service chain placement when (*i*) there is not enough network throughput available or (*ii*) doing so would increase path latency between a network element (e.g., a switch) and a Metron server.

In 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 server placement decision with minimal overhead, we use the simple, yet powerful, opportunistic scheme of "the power of two random choices" [76]. 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.

Algorithm 1 outlines our server placement scheme. Metron queries the load of two randomly selected servers (line 5) and selects the least loaded of these two servers (lines 7-9), provided that this server can meet the necessary resource requirements (i.e., the server has enough NICs and CPU cores to realize this service chain). If the first two choices fail, then these two servers are removed from the list (line 16) and the process is repeated until a server is found (line 14). This algorithm indirectly prioritizes service chain deployments that exhibit spatial correlation with respect to the processing location because spreading this processing may result in lower performance, which is

ALGORITHM 1: Server selection for placing the stateful packet processing subgraph.

Input: Topology graph (T), number of NICs (NbN), and number of cores (NbC) required for a service chain.
Output: Upon success, a server to perform the stateful operations of a service chain, otherwise NULL.

```
Function selectServer($T, NbN, NbC$):
    $serversList \leftarrow T.metronServers()$;
    $t \leftarrow \emptyset$;                                   // t will store the chosen server
    while $size(serversList) > 0$ do
        $choices \leftarrow twoRandomChoices(serversList)$;
        for $s \in choices$ do
            if $(s.nics \geq NbN)$ and $(s.freeCores \geq NbC)$ then
                if $(t == \emptyset)$ or $(t.load < s.load)$ then
                    $t \leftarrow s$;                           // Better choice
                end
            end
        end
        if $t \neq \emptyset$ then
            return $t$;                                         // Server found
        end
        $serversList.remove(choices)$;
    end
    return $\emptyset$;
```

undesirable. Network operators can input a desired server per service chain to the Metron controller, thus completely bypassing the random server selection process described in Algorithm 1.

Randomly or explicitly selecting a server 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 use Algorithm 2 to find the most suitable network element to offload the stateless graph, using the following inputs:

(1) the topology graph (T);
(2) the server (Srv) where the stateful subgraph will be deployed (chosen by Algorithm 1), and;
(3) the rule capacity (D) required to offload the stateless subgraph.

Algorithm 2 intentionally prioritizes selection of the very first network element (ingress) toward the target server (line 3). There are two reasons for this. First, applying the classification operations of a service chain at the earliest possible stage, greatly simplifies traffic steering for this service chain at all subsequent network elements on the path to the NFV server. This is done by tagging the packets targeting this service chain at the ingress node and using only this tag to match traffic at any successor of the ingress node. Second, popular network protocols, such as Multi-Protocol Label Switching (MPLS) [96], consider ingress and egress switches to be more sophisticated, thus more powerful, by design. After a target network element has been selected, the rules that encode the stateless subgraph are installed in this element and a unique tag is appended to each of the rule actions. To establish the path between the selected network element and server, one rule is installed in each subsequent node along the path; this rule matches the tag of the offloaded service chain and selects the port that leads to the server that executes the stateful part of this service chain.

According to Algorithm 2, if the capacity requirements of the ingress node cannot meet the requirements for offloading a service chain (line 6), our algorithm selects a subsequent node along the path to the NFV server (line 10), sets up forwarding between the current and next node (line 14),

ALGORITHM 2: Network element selection for placing the stateless packet processing subgraph.

```
Input: Topology graph ($T$), server ($srv$), and rule capacity ($D$)
Output: Upon success, a node to perform the stateless operations of a service chain, otherwise NULL.
Function selectNetworkElement($T, Srv, D$):
    $C \leftarrow T.availableCapacityMatrix()$
    $inEl \leftarrow T.ingressNodeToward(Srv)$
    return recursiveNetworkElementSelection(T,inEl,Srv,C,D);
Function recursiveNetworkElementSelection($T, El, Srv, C, D$):
    // Demand is a fraction of the current capacity
    if $D \leq (C_{El} \cdot CAP\_THR)$ then
        return $El$;
    end
    else
        $nextEl \leftarrow T.nextNodeInPath(El, Srv)$
        if $nextEl == \emptyset$ then
            return $\emptyset$;
        end
        setupPath($El, nextEl$)
        return recursiveNetworkElementSelection($T, nextEl, Srv, C, D$);
    end
```

and applies the same logic recursively (line 15). Our decision is currently based on a single criterion (line 6); that is, the rule capacity of a candidate node must be greater than the required and at the same time the resources required must not exceed some measure of the capacity (*CAP_THR*). The latter condition ensures that this node has enough space to accommodate future rule updates. If no switch is selected by Algorithm 2, Metron checks if a NIC of the server selected by Algorithm 1 does so; if so, then NIC offloading is performed.

Handling Partial Offloading and Rule Priorities

Metron carefully handles the cases when a stateless subgraph contains rules with different priorities and 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 selectively offloads only the supported rules, while respecting rule priorities. To exemplify these two cases, assume a service chain deployed on the topology shown earlier in Figure 3. Assume that this service chain implements four rules that can be offloaded to the first switch, while the remaining (stateful) 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 Distributed Control Plane Unlike existing NFV controllers, such as OpenBox and E2, Metron is a *distributed* NFV framework that enables elasticity of both the control and data planes.

A distributed control plane provides fault-tolerance and resilience in the face of failures that might compromise the NFV services. At the same time, the system as a whole can meet performance targets that are far higher than what a single instance might be able to handle, thus allowing the control plane to scale together with the data plane. Metron's elastic control plane allows us to partition the network into multiple segments with different controller instances managing different devices, while maintaining a globally consistent network state.

Metron can be configured to operate with strong or eventual consistency guarantees, depending on the performance targets of the operator. To provide strong guarantees, Metron's distributed engine is based on the Raft [83] consensus protocol. Eventual guarantees trade some consistency for superior performance by reading and writing local state, while updates are subsequently propagated to other replicas in the background. When an application registers a service chain with Metron, its packet processing graphs are stored and replicated across all Metron instances, while one primary instance undertakes to place and deploy the service chain in the network segment with the highest availability, as explained in §2.3.3.

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 [‡] network on which the network operator has deployed a routing application that routes Hypertext Transfer Protocol (HTTP) traffic [§] between source and destination (through the path s1→s3). This routing is done using the information shown within green dashed-dotted outlines.

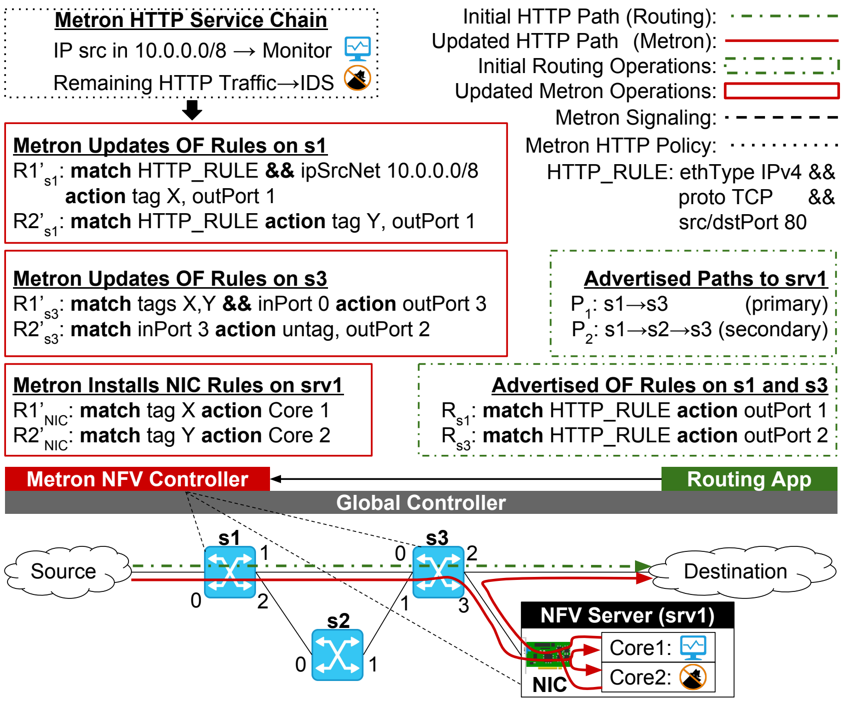

Metron HTTP Service Chain
IP src in 10.0.0.0/8 → Monitor
Remaining HTTP Traffic→IDS
Initial HTTP Path (Routing):
Updated HTTP Path (Metron):
Initial Routing Operations:
Updated Metron Operations:
Metron Signaling:
Metron HTTP Policy:
HTTP_RULE: ethType IPv4 && proto TCP && src/dstPort 80
Metron Updates 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
P1: s1→s3 (primary)
P2: 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
s2
s3
Destination
NFV Server (srv1)
Core1:
Core2:
NIC
0
1
2
3

Fig. 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 left 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).

[‡]Metron can also operate in legacy networks by adding one or more programmable switches before the NFV servers.
[§]We assume only HTTP traffic to keep the example simple.

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 per Algorithm 2 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.

Unless 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 a link failure between s1 and s3) will result in Metron installing 2 additional rules in s2 (these rules follow the same logic as 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 §5.6 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.

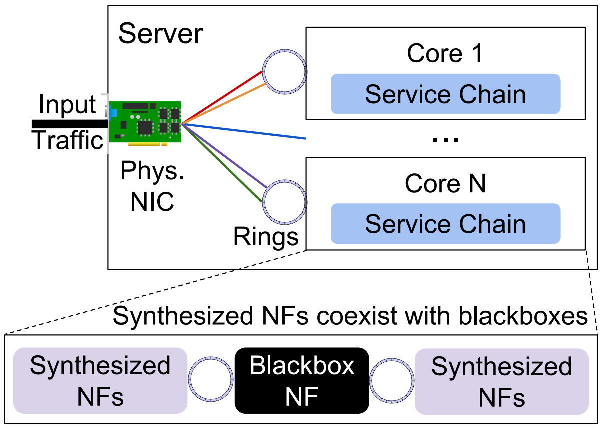
3 Integration of Blackboxes

Network operators deploy blackbox NFs ranging from dedicated hardware devices to closed-source binaries running as native processes or inside VMs or containers. These blackbox functions provide crucial functionality to the network at the cost of increasing its processing/management complexity. Despite the presence of blackboxes, it is essential to ensure that NFV service chains provide high performance, therefore we put special effort into fully integrating blackboxes in Metron.

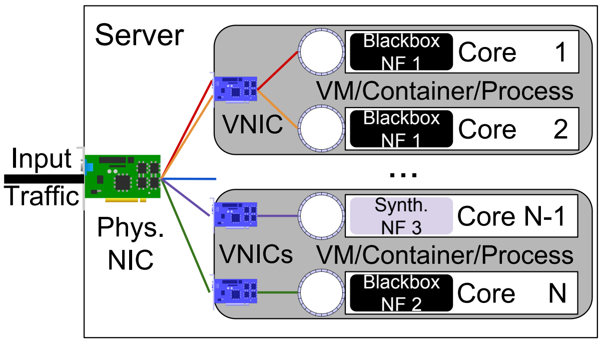
Because of the different varieties of blackbox deployments (e.g., VMs, containers, middleboxes), Metron offers several ways to co-locate Metron service chains with blackboxes, as shown in Figure 6. Because Metron can query the load of the CPUs executing the different blackbox functions and scale them automatically as described in §4, Metron's blackbox integration provides elasticity.

Software-based Blackbox Integration

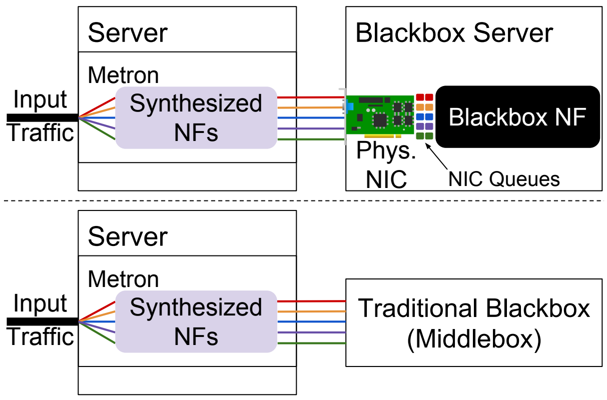
Figure 6a depicts the first integration method, which is purely software based. This method allows a blackbox function to share memory with existing Metron service chains, therefore exchange references to packets (instead of copying the packets) with these service chains. When a blackbox function needs to be placed between two Metron service chains, the Metron controller partially synthesizes the entire pipeline. Specifically, the controller synthesizes the first part of the pipeline until the blackbox function, then proceeds with the synthesis of the second part (i.e., after the blackbox), and finally stitches all three parts together using ring buffers. This way, optimized (i.e., synthesized by SNF [49]) Metron service chains can transparently coexist with blackboxes.



(a) Integration of blackbox NFs using ring buffers.



(b) Integration of blackbox NFs using Virtual NICs (VNICs).



(c) Integration of blackbox servers and traditional middleboxes. Metron load balances traffic classes across multiple NIC queues to scale multi-core blackboxes.

Fig. 6. Metron's approaches for integrating various types of blackbox packet processing functions.

Hardware-assisted Virtual Blackbox Integration

Metron also allows integration of arbitrary blackbox functions deployed on VMs and/or containers. This approach is depicted in Figure 6b. Traditional systems use software switches, such as Open vSwitch (OVS) [92], to dispatch packets to VMs/containers. In contrast, Metron uses the mapping of tags associated with its synthesized traffic classes to accurately dispatch these traffic classes to the correct VMs/containers, avoiding the need for a software switch. This mapping can efficiently be implemented through hardware rules, leveraging SR-IOV [13].

SR-IOV allows VMs to open a VNIC, allowing the VM to access a slice of a physical NIC's resources without involving the host or a hypervisor. A VNIC appears to its VM as a physical NIC running its own driver and managing its own resources. Metron instructs a physical NIC to direct packets matching the traffic classes of a VM to its dedicated VNICs. When a single VNIC is used by a VM that uses multiple cores, one could rely on RSS to dispatch packets to multiple queues. However, this approach does not leverage Metron's accurate and load-aware traffic dispatching. To that end, Metron instructs the physical NIC to tag the packets according to the core associated with this traffic class by the controller. The tag can then be used to dispatch packets to queues just as a Metron agent does (i.e., using explicit rules directing packets to cores according to the tags). Note that some older NICs, such as Intel's 82599 ES [38], provide VNICs with only a single (Rx and Tx) pair of queues. Therefore, Metron also allows allocation of one VNIC per core of the VM, thus requiring the VM operator deal with multiple VNICs.

Integration of Traditional Middleboxes

Finally, as shown in Figure 6c, Metron allows the integration of hardware blackbox devices (e.g., traditional middleboxes) in a service chain. If such a blackbox device is placed between two Metron service chains, Metron uses the same partial synthesis approach described above. The top part in Figure 6c shows how Metron instructs the NIC before a blackbox NF to load balance the various traffic classes heading to this blackbox NF. In the case of a traditional middlebox, Metron cannot access its resources (e.g., NICs), as shown at the bottom part in Figure 6c; in this case Metron's scaling will be limited by the number of physical instances of this middlebox. In §5.4 we demonstrate a practical use case with a blackbox NF between two Metron service chains.

If no classification equipment is available between the blackbox and a subsequent Metron device, then it is impossible to re-classify the traffic into the traffic classes synthesized by the second service chain. In this case, one can tag packets before they leave the first service chain and use these tags to dispatch packets to the correct cores executing the second service chain. This is only possible when the blackbox does not modify the tags. However, a blackbox might remove or rewrite tags applied by an earlier Metron instance. Consequently, packets will arrive at the subsequent Metron instance without the correct tags; therefore, Metron will need to re-classify the traffic using a software-based classifier before forwarding packets to the actual second service chain.

4 Dynamic Scaling

In §2.3.1 we explained how Metron encodes a service chain as a set of 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 splits 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 only the code responsible for each of its traffic classes (rather than the code of the entire NF).

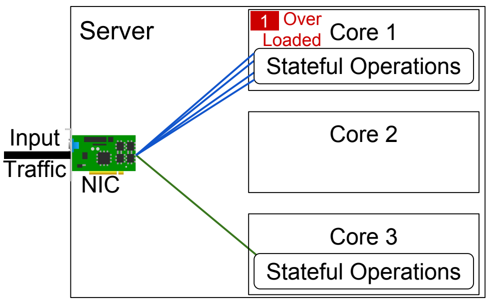
Traffic Class-level Scaling

When creating a service chain's traffic classes we leverage a grouping technique. 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 a File Transfer Protocol (FTP) traffic classes heading to a Network Address and Port Translation (NAPT) 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).

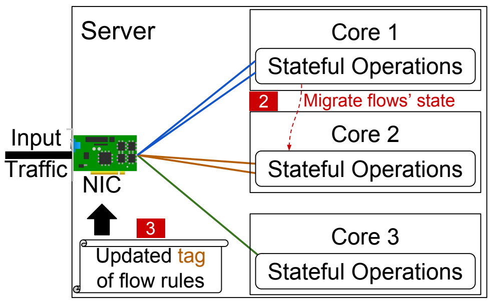
Scaling Strategy

Figure 7 visualizes the scaling strategy followed by Metron, when the load of a group of traffic classes processed by a core exceeds or falls below a predefined threshold.

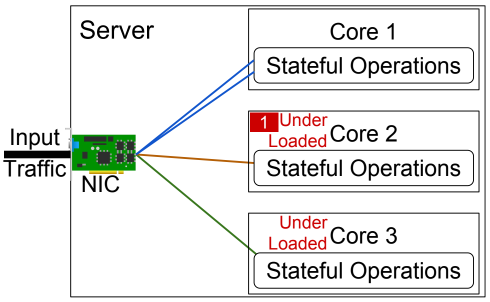
First we describe the case when a CPU core is overloaded. We use the example case shown in Figure 7a, where CPU core 1 is the overloaded core. In this case, the Metron controller dynamically scales out the group of traffic classes processed by this core by splitting this group into two subgroups. The first subgroup remains on the same CPU core as the original group (i.e., core 1), while the second subgroup is moved to a different (non-overloaded) CPU core (i.e., core 2). For this move to be successful, the Metron controller first migrates the state of the affected flows (step 2 in Figure 7b) and then updates the tag of the NIC rules that match these flows in order to dispatch them to the assigned CPU core (step 3 in Figure 7b). We call this mechanism "traffic class deflation" to differentiate it from its inverse "traffic class inflation" process (explained in the next paragraph).



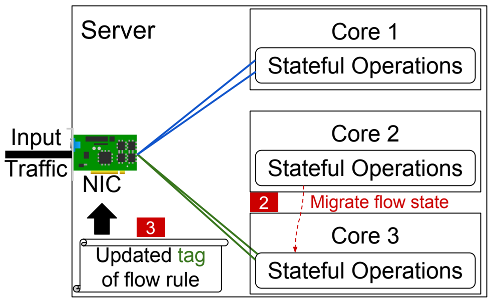
(a) Detection of overload event on CPU core 1 with 4 flows (step 1).



(b) Flows' state migration to core 2 (step 2) and update of the corresponding flow rules' tag (step 3).



(c) Detection of underload events on CPU cores 2 and 3 with 1 flow each (step 1).



(d) Flow state migration to core 3 (step 2) and update of the corresponding flow rule's tag (step 3).

Fig. 7. Metron's dynamic scaling strategy during CPU core overload (a and b) and CPU core underload (c and d) events.

When Metron detects low CPU utilization, traffic class inflation can merge two or more groups of traffic classes that exhibit the same write operations. The detection of such an event is visualized in Figure 7c, where CPU core 2 is underloaded. The Metron controller reacts to this event by first moving the state of the flow served by core 2 to another underloaded core (i.e., core 3), as shown by step 2 in Figure 7d. Then, a rule update follows, which instructs the server's NIC to dispatch the affected flow to CPU core 3 using a tag associated with that core (step 3 in Figure 7d). If this tag is the same as that of an existing traffic class then inflation of that traffic class has occurred.

The following paragraphs provide details about the internals of Metron's scaling mechanism, including a suggestion for future work on how to provide strong state consistency guarantees.

Details about Scale Out Events (or Traffic Class Deflation)

The Metron controller periodically queries the load of the active CPU cores (i.e., those executing stateful operations) of the servers. The default frequency for monitoring server resources is set to 1 Hz, but Metron allows network operators to tune this parameter dynamically. The controller remembers the load of these CPU cores at the last clock tick and computes the expected future load at the next clock tick ($\hat{L}_{t+1}$). This value is computed as the sum of the current load and the difference between the current load and the load at the last clock tick, as follows:

$$\hat{L}_{t+1} = L_t + (L_t - L_{t-1}) \tag{1}$$

Then, the controller stores the set of overloaded CPU cores, i.e., those that exhibit current or expected future load above a given threshold that we call the overload threshold. The default value of the overload threshold is set to 75%. The predicted future load is used to ensure that a core is correctly and timely (i.e., early on) characterized as "loaded". More sophisticated load prediction schemes are left as future work.

Details about Scale In Events (or Traffic Class Inflation)

A second set of underloaded CPU cores with their previous, current, and predicted future load values is stored by the controller. A core is characterized as underloaded when its previous, current, and future load values are below the underload threshold (with a default value of 25%). We consider all three values (i.e., previous, current, and future load) to avoid oscillations due to temporary events, e.g., deactivating a core because its load is temporarily low. Metron exposes knobs so that network operators can dynamically adjust the overload/underload thresholds.

Our experiments in §5.6.2 demonstrate that the (re)configuration of a network's classification equipment (e.g., an OpenFlow switch or a NIC) may take up to (several) seconds, depending on the device and the number of rules to be installed/updated. For this reason, recovering from a bad scaling decision may take a long time. To prevent this from happening, it is crucial to prioritize one of the two scaling events. We designed Metron's scaling system with a focus on prioritizing scale out events, thus Metron splits the load of the CPU core with the highest load first. After all scale out events are addressed, then Metron handles scale in events for the underloaded cores.

Ensuring Stable Scaling Decisions

Migrating the traffic classes handled by an underloaded core to a non-overloaded core might sometimes be risky. The controller must choose a candidate core that is unlikely to become overloaded after the migration has concluded. Therefore, Metron migrates traffic classes of underloaded cores only to other underloaded cores. Similarly a scale in operation should be avoided if the sum of the load on the core to be released together with the load on the core to host the migrated traffic class(es) exceeds the overload threshold. This countermeasure prevents oscillations between scale out and scale in events, which might increase the latency of the flows being migrated.

For the same reason, Metron prevents a scaling decision being made while the effect of the previous scaling event has not yet been experienced. The controller remembers the time when the last scaling event of each CPU core occurred. In case of a scale out operation, the time is measured from when the core starts to report some load. Further scaling operations are then inhibited for each core for at least 5 seconds since the last scaling operation.

Split and merge operations may repeat until Metron can no longer split/merge a group of traffic classes. A group with a single traffic class is an example of an unsplittable group of traffic classes. The reaction time of our scaling strategy is primarily affected by the time required for the controller to monitor and reconfigure the data plane. §5.5 shows how this strategy performs in practice.

Discussion about State Migration

We use the concrete example shown in Figure 7 to explain how Metron performs state migration during scaling operations. S6 [110], StateAlyzr [53], OpenNF [28], or the work by Olteanu and Raiciu [82] 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. [41]. We leave the integration of sophisticated state migration techniques as future work, as this is outside the focus of this article.

5 Evaluation

In this section we evaluate performance and scalability aspects of Metron as well as the interoperability between Metron and blackbox NFs. §5.1 describes how we implemented Metron, while §5.2 outlines the testbed used to conduct the experiments. §5.3 benchmarks the data plane performance of standalone Metron service chains, while §5.4 demonstrates how blackboxes are integrated with Metron service chains. In §5.5, §5.6, and §5.7 Metron's dynamic scaling, deployment micro-benchmarks, and large scale placement emulations are presented (respectively).

5.1 Implementation

The Metron controller [46] is built on top of ONOS [9, 85], an open source, industrial-grade system that is designed to scale well. Key to this decision was the fact that ONOS exposes unified abstractions for a large variety of drivers that cover popular network configuration protocols, such as OpenFlow [67], P4 [14], Network Configuration Protocol (NETCONF)/YANG [12, 24], REST, and Simple Network Management Protocol (SNMP) [17]. ONOS was extended with a new driver to remotely monitor and configure servers and their NICs. This driver is available at [43].

Metron's data plane [5] extends FastClick [8]. The Virtual Machine Device queues (VMDq) of DPDK [103] are used for hardware dispatching based on the value of the destination Medium Access Control (MAC) address or Virtual Local Area Network (VLAN) ID fields. Metron uses the former header field as a filter, because the large address space of a MAC address provides unique tags for trillions of service chains. Each server requires as many tags (or virtual MAC addresses) as the number of its CPU cores. To scale to 100 Gbps, Metron configures the hardware classifiers of Mellanox ConnectX-4 MT27700 [70] and ConnectX-5 MT27800 [73] NICs (see §5.3.2 and §5.4). In this case, Metron associates NIC hardware queue IDs with a server's CPU cores, acting as tags.

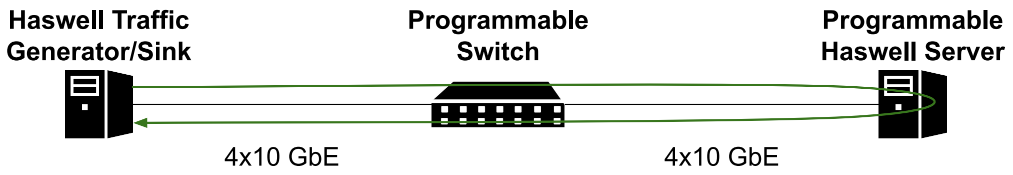
Metron uses DPDK for packet input-output (I/O). DPDK completely bypasses the Operating System (OS)'s kernel at run-time, by implementing Poll Mode Drivers (PMDs) for packet I/O, directly in user-space. PMDs do not rely on hardware interrupts to wait for incoming packets; instead, the CPU polls the state of the hardware ring buffers to process packets as soon as they are available. While this approach has low latency and high throughput, the CPU load will always be 100%. In order to make scaling decisions (see §4) based upon CPU load information, Metron's data plane counts the number of cycles spent for actual packet processing (defined as UsefulCycles) and the number of cycles spent trying to query a NIC for packets but that find an empty ring (UselessCycles). At a frequency of 10 Hz, we divide the UsefulCycles by the sum of UsefulCycles and UselessCycles to compute the actual current load per core.

5.2 Testbed

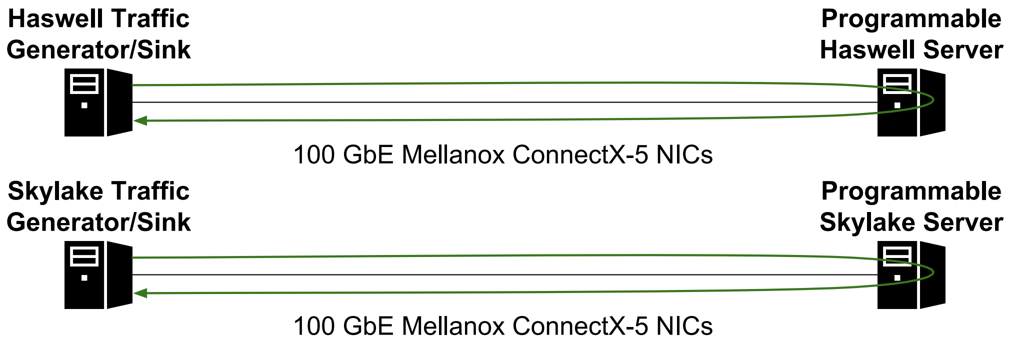
Our testbed consists of five almost identical servers, each with a dual-socket CPU with 8 cores per socket. Three of these servers have an Intel®Xeon® E5-2667 v3 (Haswell) CPU clocked at 3.20 GHz and 128 GB of DDR4 RAM at 2133 MHz. Each core has 2x32 KB L1 (instruction and data caches) and 256 KB L2 caches, while one 20480 KB L3 cache is shared among the cores in each socket. The remaining two servers have an Intel®Xeon® Gold 6134 (SkyLake) CPU clocked at 3.20 GHz and 256 GB of DDR4 RAM at 2666 MHz. Each core has 2x32 KB L1 (instruction and data caches) and 1024 KB L2 caches, while one 25344 KB L3 cache is shared among the cores in each socket. Hyper-threading is disabled on all servers and the OS is the Ubuntu 16.04.2 distribution with Linux kernel v.4.4.

Testbed at 40 Gbps

Network operators typically use a switched infrastructure to interconnect multiple servers, thus facilitating scaling and traffic steering. Modern switches are programmable, hence some packet processing operations can be offloaded, thus reducing the processing demands at the servers. Two Haswell servers are used in the testbed shown in Figure 8a, each with two dual port Intel 82599 ES NICs [38] (with a total capacity at 40 Gbps). In this testbed, two programmable switches are used interchangeably: a NoviFlow 1132 switch with firmware version NW400.2.2 and an HP 5130 EI switch [34] with software version S5130-3106. The former is a powerful multi-port 10 GbE OpenFlow switch used to evaluate the deployment presented in §5.3.1. The latter is a hybrid (i.e., legacy and OpenFlow-based) switch used to assess how hardware diversity affects Metron (§5.6.2).



(a) Deployment at 40 Gbps using two Haswell servers, each with two dual-port 10 GbE Intel 82599 ES NICs [38], interconnected through a programmable (i.e., OpenFlow) switch.


Haswell Traffic Generator/Sink
Programmable Haswell Server
100 GbE Mellanox ConnectX-5 NICs
Skylake Traffic Generator/Sink
Programmable Skylake Server
100 GbE Mellanox ConnectX-5 NICs

(b) Deployment at 100 Gbps using two (Haswell or SkyLake) servers connected back-to-back, each with one 100 GbE Mellanox ConnectX-5 MT27800 [73]) NIC.

Fig. 8. Topologies used to evaluate different aspects of Metron throughout this article. The left-most server in each topology acts as a traffic generator and sink. The remaining nodes (servers and/or switches) are used for packet processing.

Testbeds at 100 Gbps
To further stress the performance limits of NFV service chains, a 100 GbE testbed is used as shown in Figure 8b. This testbed has two variants. First, the top part in Figure 8b shows two back-to-back connected Haswell servers using 100 GbE Mellanox ConnectX-5 MT27800 NICs [73]. This testbed is similar to the 100 GbE testbed used by an earlier version of Metron [48] with the only difference being the replacement of the 100 GbE Mellanox ConnectX-4 MT27700 NICs [70] with the newer 100 GbE Mellanox ConnectX-5 MT27800 NICs [73]. This replacement allows us to improve upon the results reported in [48], as the NIC was one of the hardware limitations of the earlier testbed.

To completely remove the bottlenecks of the testbed shown in the top of Figure 8b, thereby enabling line-rate processing at 100 Gbps, the configuration shown in the bottom part of Figure 8b was also deployed. In this latter testbed, two SkyLake servers are equipped with the same 100 GbE Mellanox ConnectX-5 MT27800 NICs [73]. In §5.3.2 a stateful service chain is evaluated on both hardware architectures. In §5.4 we use a larger 100 GbE testbed with both Haswell and SkyLake servers to showcase how Metron coexists with blackboxes.

Relevant details
The leftmost server in each topology shown in Figure 8 acts as a traffic generator and receiver. In all of the experiments, another server (not present in Figure 8) is used to run the Metron controller. Each experiment was conducted 10 times and we report the 10^{th}, 50^{th}, and 90^{th} percentiles.

5.3 Large-Scale Deployment of Standalone Metron Service Chains

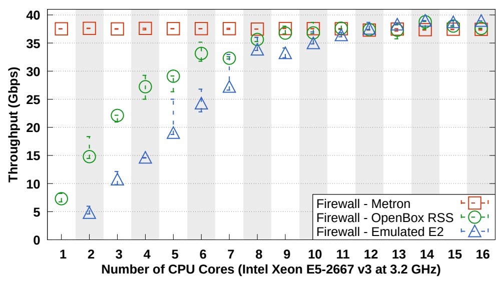
We test Metron's data plane performance at scale using complex service chains with a large number of deeply-inspected (§5.3.1) and stateful (§5.3.2) traffic classes at 40 and 100 Gbps (respectively).

5.3.1 Deep Packet Inspection at 40 Gbps We begin with an experiment at 40 Gbps using a service chain of a campus firewall followed by a DPI NF, deployed on the programmable Haswell server of the testbed shown in Figure 8a. The firewall NF implements an Access Control List (ACL) of 1000 rules, derived from a campus trace; these rules match all the packets of this trace. The output of the firewall is sent to a DPI NF that uses a set of regular expressions similar to Snort (see [16]).

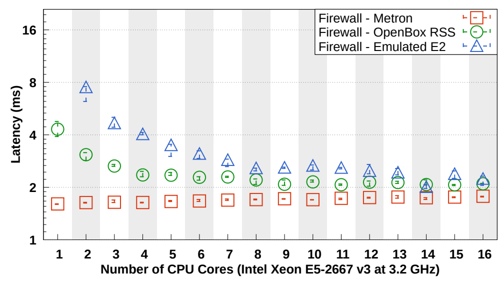
We compare Metron against two state of the art systems: (i) an accelerated version of the OpenBox data plane based on RSS and (ii) an emulated version of E2. In the latter case, we emulate E2's SoftNIC by using a dedicated CPU core (i.e., core 1) to dispatch packets to the remaining CPU cores of the server (i.e., cores 2-16), where the NFs of the service chain are executed. As a result, all of the graphs of the emulated E2 in §5.3 starting from core two.

We injected the campus trace at 40 Gbps and measured the performance of the three approaches. 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 (Metron offloaded the firewall to the programmable switch as shown in Figure 8a). Figures 9a and 9b show the results of this experiment, while Figure 9c visualizes the CPU core allocation with respect to the server's Non-Uniform Memory Access (NUMA) regions used in this experiment. To fairly compare Metron against the other two approaches, we run a simple RSS-based forwarding NF in the server, such that all packets follow the exact same path (generator, switch, server, switch, and sink) in all three experiments.

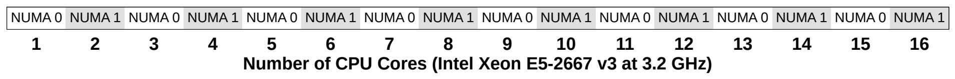
Figure 9a shows that OpenBox and the emulated E2 can realize the firewall NF at line-rate. However, this is only possible if more than half of the server's CPU cores are utilized. Specifically, OpenBox requires 9 CPU cores, while the emulated E2 requires 11 CPU cores. In contrast, Metron completely offloaded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



(a) Throughput (Gbps).



(b) Latency (ms) on a logarithmic scale.



NUMA 0	NUMA 1	NUMA 0	NUMA 1	NUMA 0	NUMA 1	NUMA 0	NUMA 1	NUMA 0	NUMA 1	NUMA 0	NUMA 1	NUMA 0	NUMA 1	NUMA 0	NUMA 1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Number of CPU Cores (Intel Xeon E5-2667 v3 at 3.2 GHz)

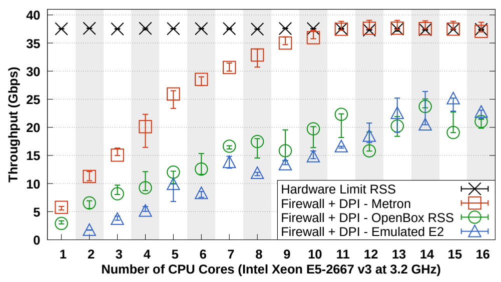
(c) Mapping of physical CPU core IDs to NUMA nodes on an Intel®Xeon® E5-2667 v3.

Fig. 9. Performance versus number of CPU cores of a campus firewall with 1000 rules using: (*i*) Metron with the firewall being offloaded, (*ii*) an accelerated version of OpenBox using RSS, and (*iii*) a software-based dispatcher emulating E2.

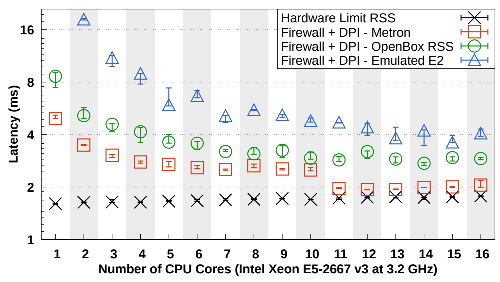
Looking at the latency of the three approaches in Figure 9b, it is evident that software-based dispatching (dark blue triangles) incurs a large amount of latency. Hardware dispatching using RSS (green circles) achieves substantially lower latency because it involves less inter-core communication. However, since the firewall executes computationally demanding classification operations in software, OpenBox still exhibits high latency that cannot be decreased simply by increasing the number of cores. As an example, using 16 CPU cores has comparable latency to 4 CPU cores. In contrast, Metron achieves nearly constant low latency (red 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 OpenBox and the emulated E2 (respectively), when each system uses one core for processing the NF (note that the emulated E2 requires 2 CPU cores in this case). At the full capacity of the server (i.e., 16 cores), the latency of the three systems is comparable; but Metron achieves 30% and 19% lower latency than the emulated E2 and OpenBox systems respectively.

Next, the example campus firewall is chained with a DPI NF in order to realize the entire service chain. This chain pushes the performance limits of the three approaches as shown in Figure 10. In this scenario, Metron implements DPI in software. First, we observe that even when using all of the server's CPU cores, OpenBox and the emulated E2 can only achieve at most 25 Gbps (see Figure 10a). 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 [19], while 100 Gbps networks are increasingly gaining traction [109]. In these higher data rate environments, these alternatives would require more than 16 CPU cores (and potentially more than one server) to achieve sufficient throughput, and are not guaranteed to scale because of the processing requirements of large service chains.

Metron exploits the combined network and server capacity to scale even complex NFs, such as DPI, at the speed of the hardware. This can be confirmed by comparing the red squares (i.e., "Metron") with the black crosses (i.e., "Hardware Limit RSS") in Figure 10a. Metron requires only 10 CPU cores in a single server to achieve this result, thus substantially shifting the scaling point for large service chains. The latency results in Figure 10b further highlight Metron's abilities. With 16 CPU cores, Metron deeply inspects all packets for this service chain at the cost of only 15.5%



(a) Throughput (Gbps).



(b) Latency (ms) on a logarithmic scale.

Fig. 10. Performance versus number of CPU cores of a campus firewall with 1000 rules followed by a DPI using: (*i*) Metron with the firewall being offloaded, (*ii*) an accelerated version of OpenBox using RSS, and (*iii*) a software-based dispatcher emulating E2. "Hardware Limit RSS" showcases the speed of the hardware, using the firewall NF offloaded into an OpenFlow switch followed by an RSS-based forwarding NF at the server. The two different font colors represent the location of each core with respect to the server's NUMA regions, as shown in Figure 9c.

higher latency than the minimum latency of this testbed, shown with black crosses. At the same time, OpenBox and the emulated E2 incur 35-97% higher latency than Metron, while achieving roughly half of Metron's throughput. The difference in latency increases rapidly when fewer cores are utilized. For example, when each system uses one core, Metron achieves 75% and 358% lower latency than OpenBox and the emulated E2 (respectively). Moreover, Metron's performance increases *linearly* with increasing number of cores, while both OpenBox and the emulated E2 systems exhibit severe performance fluctuations (e.g., OpenBox with 11 cores performs better than OpenBox with 15 or 16 cores). This highlights that part of Metron's performance stems from better load balancing.

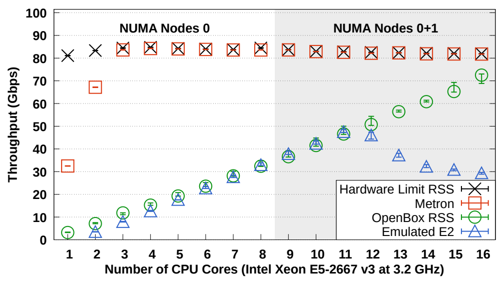
5.3.2 Stateful Service Chaining at the Speed of 100 Gbps NICs The emerging 100 GbE deployments in datacenters will challenge the performance limits of NFV systems [23, 57, 109]. In this section we show how Metron allows network operators to deliver the expected levels of performance.

To further stress the performance of Metron, OpenBox, and the emulated E2 systems, we conducted experiments using the 100 GbE testbeds shown in Figure 8b. All the experiments in this section were conducted with NUMA-aware CPU core allocation, which is different from the CPU core allocation followed in §5.3.1 and [48].

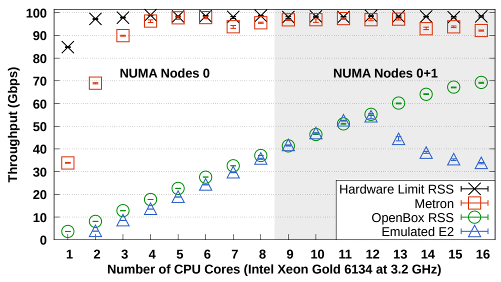
We analyzed all 24 million packets of the campus trace used in §5.3.1 and derived 3597 distinct Internet Protocol (IP) prefixes that match all destination IP addresses. Then, we implemented a standards-compliant router and populated its routing table with these prefixes. This router was chained with two stateful NFs: a NAPT and a Load Balancer (LB) that implements a flow-based round robin policy across 5 destination servers. In this scenario, Metron can only offload the routing table of the router to the Mellanox NIC using DPDK's flow API [104]. The remaining functions of the router, such as Address Resolution Protocol (ARP) handling, IP fragmentation, Time to Live (TTL) decrement, etc., together with the stateful NFs (i.e., NAPT and LB) are executed in software.

Metron vs. State of the art

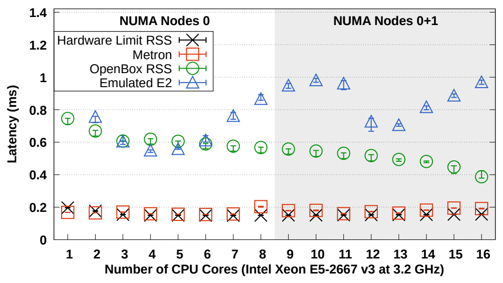
The throughput achieved by the three systems on two different hardware architectures is shown in Figures 11a and 11b. The results on both hardware architectures show a slow but linear increase in throughput with an increasing number of CPU cores for both OpenBox and the emulated E2 approaches. Using linear regression on the medians, we found that the throughput of OpenBox



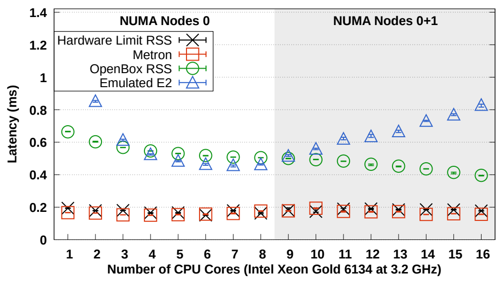
(a) Throughput (Gbps) on an Intel Xeon E5-2667 v3.



(b) Throughput (Gbps) on an Intel Xeon Gold 6143.



(c) Latency (ms) on an Intel Xeon E5-2667 v3.



(d) Latency (ms) on an Intel Xeon Gold 6143.

Fig. 11. Performance of a stateful service chain (Router→NAPT→LB) at 100 Gbps on two different hardware architectures by Intel (Haswell on the left and Skylake on the right) using a 100 GbE Mellanox ConnectX-5 NIC, achieved by: (*i*) Metron with the routing table of the Router NF being offloaded, (*ii*) an accelerated version of OpenBox using RSS, and (*iii*) a software-based dispatcher emulating E2. "Hardware Limit RSS" shows the forwarding speed of the server (i.e., no service chain) using RSS as a traffic dispatcher.

increases by 4.54 Gbps/core on the Haswell architecture and by 4.27 Gbps/core on the SkyLake architecture. The maximum throughput achieved by OpenBox is 73 Gbps and 65.6 Gbps on the Haswell and SkyLake architectures respectively. Despite using the same Haswell server, Figure 11a reports higher throughput than reported in Figure 7a in [48]. This is due to: (*i*) the different CPU core allocation strategy and (*ii*) a different Mellanox NIC than used in [48].

Performing a similar linear regression on the medians between 2 and 12 cores for the emulated E2 system (dark blue triangles in Figures 11a and 11b) shows an increase by 4.94 Gbps/core and by 4.91 Gbps/core on the Haswell and SkyLake architectures respectively. However, in both cases using more than 12 CPU cores results in substantial performance degradation due to an increasing amount of inter-core communication.

In contrast, Metron matches the performance limits of the underlying hardware on both architectures. This is shown by comparing the red squares with the black crosses in Figures 11a and 11b. More importantly, Metron achieves these results while 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: it (*i*) saves CPU cycles by performing the destination IP lookup operations of the router and (*ii*) load balances the traffic classes matched by the hardware classifier across the available CPU cores. Exploiting these advantages allows Metron to quickly achieve throughput comparable to the "Hardware Limit RSS" case using only 3-4 cores, despite performing several stateful operations (i.e., NAPT and LB) in software.

Note that the maximum attainable throughput of the hardware on the Haswell architecture is limited to 85 Gbps. The maximum attainable throughput on the same architecture using an older generation of Mellanox NICs (i.e., ConnectX-4 instead of ConnectX-5) and different CPU core allocation strategy was 76 Gbps [48]. This means that with the upgraded testbed we managed to partially address the hardware performance bottlenecks reported in [48]. In contrast, near line-rate processing (i.e., 98.5-98.9 Gbps) is achieved by Metron and "Hardware Limit RSS" cases on the SkyLake architecture as shown in Figure 11b. According to a performance report by Mellanox [69], the underlying ConnectX-5 NIC exhibits a slight throughput degradation when processing 64-byte frames, while performing line-rate processing of frames larger or equal than 128 bytes. In the injected trace, 26.9% of the frames have a size in the range of [50, 100] bytes, which may explain why the maximum attainable throughput in Figure 11b tops out at 98.9 instead of 100 Gbps.

Figure 11b shows a marginal throughput degradation of "Metron" as compared to the "Hardware Limit RSS" for some CPU cores (i.e., 5 to 16) on the SkyLake architecture. This is a limitation of the classification capabilities of the underlying NIC. To confirm this, we performed an additional experiment (not shown in Figure 11), in which we replaced the Metron service chain's (stateful) software processing with a simple forwarding NF, while still using the same rules for NIC to CPU core dispatching. This rule-based forwarding NF resulted in the same marginal throughput degradation as "Metron", when using 5-16 CPU cores. Despite this hardware limitation, Metron's rule-based load balancing outperforms RSS. We conclude this after checking the state (i.e., packet and byte counters) of the active hardware queues in the NIC during this experiment.

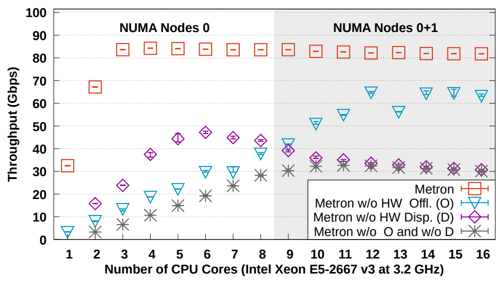
Figures 11c and 11d show the latency of the three systems on the same two Intel architectures. First, all three systems achieve sub-millisecond latency on both architectures. Second, by comparing the dark blue triangles in Figures 11c and 11d we observe that SkyLake hides a large fraction of the inter-core communication latency introduced by E2, resulting in comparable latency with OpenBox (in some cases). This is in contrast with the case on the Haswell architecture were the latency introduced by the emulated E2 dramatically increases between 5 and 16 cores, especially once packets must cross the Quick Path Interconnect (QPI) between the two sockets. On the Haswell architecture OpenBox achieves more than 4x lower latency than E2 (see Figure 11c), while up to 2x latency decrease is achieved by OpenBox on the SkyLake architecture. A comparison of the red squares with the black crosses in Figures 11c and 11d shows that Metron achieves similar or sometimes lower latency than the "Hardware Limit RSS" case. The latter is possible due to Metron's *better load balancing* of traffic across the available hardware queues. Specifically, Metron's latency ranges between 0.148-0.20 ms on both architectures. This latency is at least 2x lower than the lowest latency achieved by the other two systems on both architectures.

Dissecting Metron's Performance

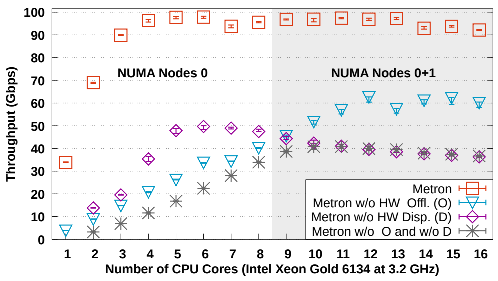
To quantify the factors that contribute to Metron's high performance, we conducted an additional experiment using the same testbeds, input trace, and service chain. The results are depicted in Figure 12. Note that the curves with the red squares (i.e., Metron's performance) in Figures 11 and 12 are identical. The purpose of Figure 12 is to showcase the expected performance penalties when one starts removing our key contributions from Metron. More specifically:

(1) Metron without hardware offloading (i.e., dark cyan triangles in Figures 12). Hardware offloading corresponds to Contribution 1 in §1.2;
(2) Metron without accurate hardware dispatching to the correct core (purple rhombs in Figure 12). Accurate dispatching corresponds to Contribution 2 in §1.2;
(3) Metron without either of the contributions (gray stars in Figure 12).

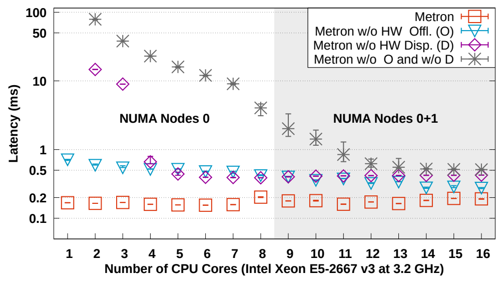
Comparing "Metron" vs. "Metron w/o HW Offl." (red squares vs. dark cyan triangles in Figure 12) quantifies the benefits of Metron's hardware offloading feature. In the "Metron w/o HW Offl." case input packets are still dispatched to the correct core (using the flow rule-based classifier of the



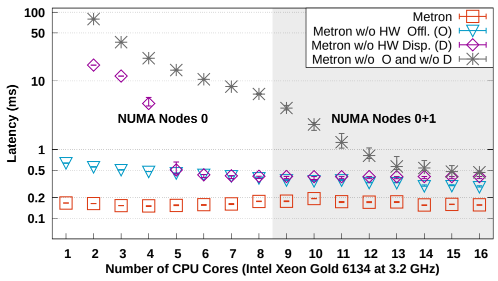
(a) Throughput (Gbps) on an Intel Xeon E5-2667 v3.



(b) Throughput (Gbps) on an Intel Xeon Gold 6143.



(c) Latency (ms), on a logarithmic scale, on an Intel Xeon E5-2667 v3.



(d) Latency (ms), on a logarithmic scale, on an Intel Xeon Gold 6143.

Fig. 12. Metron's hardware offloading (O) and dispatching (D) contributions to the performance of a stateful service chain (Router→NAPT→LB) on two different hardware architectures by Intel (Haswell on the left and Skylake on the right) using a 100 GbE Mellanox ConnectX-5 NIC. The word "without" is abbreviated as "w/o".

Mellanox NIC), but each core executes the entire service chain logic in software. The maximum throughput achieved in this case (i.e., dark cyan triangles in Figures 12a and 12b) is slightly lower than the throughput of the "OpenBox RSS" case shown in Figures 11a and 11b. However, as shown in Figures 12c and 12d Metron realizes more than 2x lower latency than "Metron w/o HW Offl." due to its ability to perform hardware offloading. A key difference between "OpenBox RSS" and "Metron w/o HW Offl." is that the latter 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. This explains why the throughput of "Metron w/o HW Offl." does not further increase after using 13 or more CPU cores. Neither of these cases exploits the NIC's ability to offload the routing operations, thus costing CPU cycles.

Next, the comparison between "Metron" and "Metron w/o HW Disp." (red squares vs. purple rhombs in Figure 12) 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 should process them (i.e., software dispatching similar to the emulated E2 case in Figures 11a and 11b). As shown in Figures 12a and 12b, both "Metron w/o HW Disp." (with purple rhombs) and the emulated E2 (with dark blue triangles in Figures 11a and 11b) cases exhibit similar throughput 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. As shown in Figures 12c and 12d, Metron's accurate dispatching results in 2-80x and 2-100x lower latency than "Metron w/o HW Disp." on the Haswell and SkyLake architectures respectively.

Finally, the "Metron w/o O and w/o D" case in Figures 12a and 12b 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 on both hardware architectures, this time after using 9 or more cores. The gray stars in Figures 12c and 12d indicate that the latency of this system is substantially higher than Metron's latency, especially when using few CPU cores. This occurs because the processing demands of the service chain combined with the excessive amount of inter-core communication prevent the system from effectively using its fast caches.

Key Outcome

As explained in §4, 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 *NICs' hardware limit* with a single server is an important achievement.

5.4 Metron with Integrated Blackbox NF

Blackboxes are typically employed by network operators to execute proprietary packet processing code for specific purposes, such as routing and monitoring NFs using Cisco's Cloud Services Router (CSR) [20] and Snort's DPI [99]). In the presence of blackboxes, achieving high performance while provisioning resources on demand is hard, mainly because different blackboxes (*i*) exhibit different processing bottlenecks and (*ii*) have different (or sometimes no) ways of adapting to changing workloads. In this section we demonstrate how Metron orchestrates, not only native Metron service chains (as demonstrated in §5.3), but also blackboxes.

Testbed

To show how Metron works in the case of a multi-server testbed, we implemented a service chaining scenario that involves 4 servers as shown in Figure 13. From left to right, the traffic generator/sink and server 2 are based on Intel's SkyLake architecture, while servers 1 and 3 use Intel's older Haswell architecture. Each server is equipped with a dual-port 100 GbE Mellanox ConnectX-5 NIC, plugged into a Peripheral Component Interconnect Express (PCIe) 3.0 slot with 16 lanes associated with CPU socket 0.

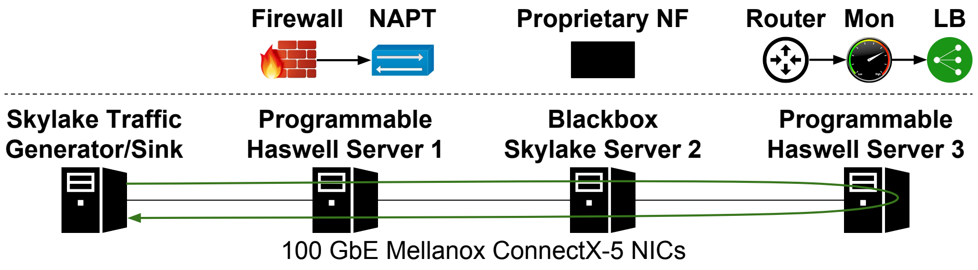

Firewall
NAPT
Proprietary NF
Router
Mon
LB
Skylake Traffic Generator/Sink
Programmable Haswell Server 1
Blackbox Skylake Server 2
Programmable Haswell Server 3
100 GbE Mellanox ConnectX-5 NICs

Fig. 13. Full-duplex deployment at 100 Gbps using two SkyLake and two Haswell servers connected back-to-back, each with a dual-port 100 GbE Mellanox ConnectX-5 NIC. Server 1 runs a Firewall→NAPT service chain, server 2 is used as a blackbox device, while server 3 runs a Router→Monitor→LB service chain.

Service Chaining Scenario

The first server acts as a traffic generator and sink; the injected traffic follows the green arrow shown in Figure 13. The second server runs a firewall NF with 2384 rules followed by a NAPT. The third server acts as a blackbox device on which a network operator can deploy proprietary packet processing NFs. Later in this section we show different ways of integrating blackbox NFs with Metron service chains. The fourth server executes a router with 2004 entries in its routing table followed by a stateful monitor, and a stateful LB (identical to the LB used in §5.3.2).

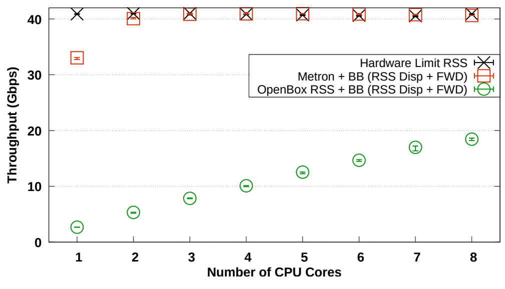
Systems under Test

Using the service chaining scenario above, we compare three different Metron deployments against two baseline approaches: The first baseline approach is a pure forwarding case in which all the service chains on the 3 processing servers are replaced by simple forwarding NFs; this case is labeled "Hardware Limit RSS" in Figure 14. The second baseline approach is the RSS-based OpenBox integrated with a forwarding blackbox NF running on server 2; this system is labeled "OpenBox RSS + BB (RSS Disp + FWD)".

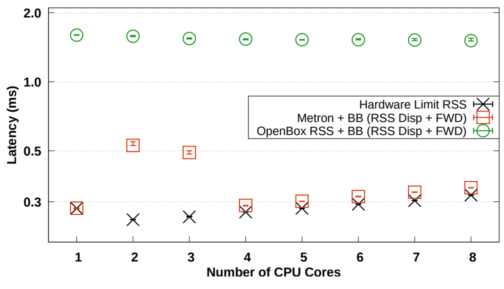
The three Metron deployments run identical code on servers 1 and 3, but differ in the blackbox NF on server 2. In the first Metron deployment, labeled "Metron + BB (RSS Disp + FWD)", the blackbox server runs the same native RSS-based forwarding application used by the baseline cases (i.e., OpenBox and the "Hardware Limit RSS"). In the second ("Metron + VM BB (TC Disp + FWD)") and third ("Metron + VM BB (TC Disp + DPI)") Metron deployments, we replace the native forwarding NF with a VM running the same NF (i.e., RSS-based forwarding) and a DPI NF (i.e., the same DPI NF used in §5.3.1) respectively. In both cases, Metron uses SR-IOV to dispatch traffic to the correct VNIC of the VM in a traffic class-aware manner as explained below. In the following paragraph, we discuss some practical considerations that motivated our evaluation choices.

Handling Flow States

In a real deployment scenario, the return path (i.e., from right to left towards the traffic sink) of the testbed shown in Figure 13 must be carefully provisioned to ensure that all returning flows are dispatched to the core that previously handled these flows in the forward direction. This ensures flow state consistency with respect to the stateful NAPT and monitor tables of the corresponding NFs. However, due to the presence of the NAPT NF, flow headers are modified by server 1, hence the RSS-based baseline systems (i.e., OpenBox and the "Hardware Limit RSS") hash (some of) these



(a) Throughput (Gbps).



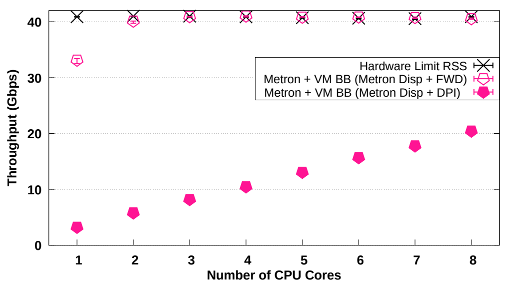
(b) Latency (ms) on a logarithmic scale.

Fig. 14. Full-duplex performance of two stateful service chains running on two different servers and a blackbox NF running on a third server at 100 Gbps using: (*i*) Metron and (*ii*) an accelerated version of OpenBox using RSS. The blackbox server executes a native RSS-based forwarding NF. "Hardware Limit RSS" shows the forwarding speed of the testbed (i.e., no service chain) using RSS as a traffic dispatcher on all servers. All the CPU cores used in this experiment belong to NUMA node 0.

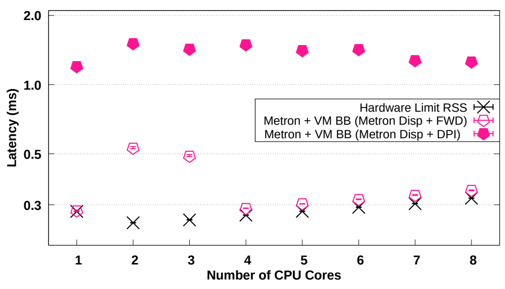
modified flows differently, on their way back to the traffic sink. Metron has an efficient solution to this problem as it can install a mirrored version of the forward path's rules in the corresponding NICs to perform traffic dispatching of packets on the return path. Metron also detects the presence of flow modification elements (e.g., a NAPT) in a service chain, hence the rules for the return path could be crafted accordingly. Implementing a flow state affinity element for OpenBox and the "Hardware Limit RSS" systems was possible, but would introduce performance degradation. Therefore, to provide a fair comparison between Metron and Openbox without affecting the performance of the latter system, we deliberately simplified Metron's return processing path to use the same RSS-based forwarding NF used by OpenBox and the "Hardware Limit RSS" systems.

Performance Evaluation (Experiments with a Native Blackbox NF)
As shown by the black crosses in Figure 14a, the end-to-end throughput of this testbed is limited to around 40 Gbps because the Haswell servers (i.e., servers 1 and 3) cannot forward more than 80 Gbps of full-duplex traffic using the Mellanox ConnectX-5 NICs. The green circles in Figure 14a show that the throughput of OpenBox is capped around 19 Gbps using 8 CPU cores. According to the green circles in Figure 14b, the latency introduced by OpenBox is between 1.5 and 1.6 ms. This performance is several times worse than the maximum attainable performance (i.e., the "Hardware Limit RSS" case), mainly because the firewall at server 1 implements 2384 rules in software, which becomes the biggest bottleneck. On server 3 OpenBox also implements a software IP lookup element with 2004 rules, but this classifier is faster than the firewall as it contains fewer rules, only involving a single header field (i.e., destination IP address).

In contrast, the native Metron case, labeled "Metron + BB (RSS Disp + FWD)" and depicted by the red squares in Figure 14, offloads the classifiers of the Firewall and Router NFs in the corresponding Mellanox NICs. Therefore, Metron quickly realizes the maximum attainable throughput with 2 cores and the lowest attainable latency using only 4 cores. Although Metron matches the speed of the "Hardware Limit RSS", its performance can be further optimized. Specifically, the "Metron + BB (RSS Disp + FWD)" case only loosely integrates the blackbox server, as Metron has no way to scale this server along with the other Metron service chains. Instead, the blackbox server relies on RSS to perform traffic dispatching, independently of the other two servers, which are dynamically controlled by Metron. To solve this problem, we implemented an alternative approach to integrate blackboxes (introduced in §3), while dynamically adjusting the amount of traffic sent to each of the available cores of the blackbox. We evaluate this approach below (see Figure 15).



(a) Throughput (Gbps).



(b) Latency (ms) on a logarithmic scale.

Fig. 15. Full-duplex performance of two stateful service chains running on two different servers and a blackbox NF running on a third server at 100 Gbps using Metron. The blackbox server runs a VM, which executes either (*i*) a forwarding or (*ii*) a DPI NF, orchestrated by Metron. Metron dispatches traffic classes to the VM. "Hardware Limit RSS" shows the forwarding speed of the testbed (i.e., no service chain) using RSS as a traffic dispatcher on all servers. All the CPU cores used in this experiment belong to NUMA node 0.

Performance Evaluation (Experiments with a Virtualized Blackbox NF)
Blackboxes might run pieces of software that cannot be synthesized, such as a closed-source VM. To evaluate this case, we implemented "Metron + VM BB (TC Disp + FWD)" by placing the forwarding NF inside a VM. This case is depicted with the empty pink pentagons in Figure 15. Metron manages and load balances this VM, even though it has no insight about what code the VM executes. This is possible by designating a set of traffic classes of interest to this VM. Metron then employs SR-IOV on the physical NIC of the blackbox to load balance incoming packets matching those traffic classes to the VNICs of the VM. As mentioned in §3, incoming packets are also tagged, by rewriting their destination MAC address according to the target CPU core assigned by Metron. In this experiment Metron installs the rules in the Mellanox NICs; however, a different tagging scheme using e.g., an OpenFlow switch before the blackbox device, could be used as well (see the 40 Gbps experiment shown in §5.3.1). If a VM uses fewer VNICs than the number of processing CPU cores, Metron can associate tags with hardware queues which can be accessed by the target CPU cores. The empty pentagons in Figure 15 show that Metron's hardware-based VM dispatching performs as well as the native Metron dispatching. This is because Metron facilitates direct traffic class dispatching to the correct core of the correct VM, avoiding the need for costly software-based switching.

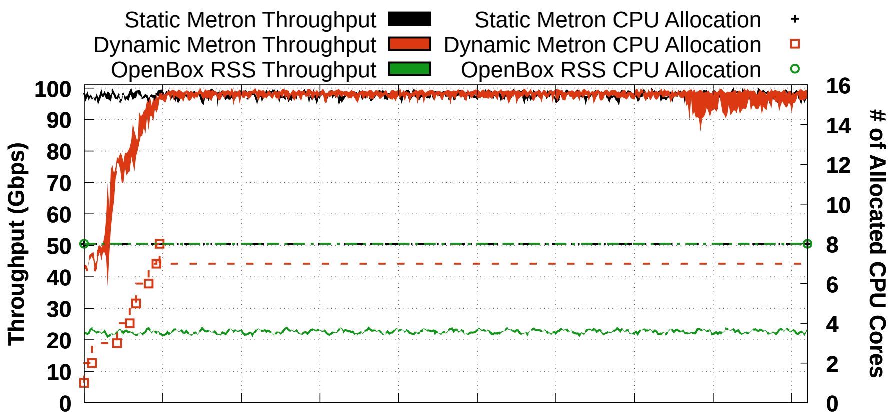
Finally, in the "Metron + VM BB (TC Disp + DPI)" case, shown with filled pentagons in Figure 15, we replaced the VM's forwarding NF with a DPI NF. We study this case to show Metron's performance when a blackbox becomes the bottleneck. Even with such a demanding blackbox, Metron can easily saturate a 10 Gbps link using 4 cores, while achieving an almost linear throughput increase with an increasing number of CPU cores. Specifically, fitting a linear equation to the median values of the filled pentagons in Figure 15a strongly suggests (R^2 of 0.999) that each additional core contributes 2.45 Gbps of additional throughput.

Note that the performance of the "Metron + VM BB (TC Disp + DPI)" case (filled pentagons in Figures 15a and 15b) is slightly higher than the performance achieved by the "OpenBox RSS + BB (RSS Disp + FWD)" case (green circles in Figures 14a and 14b). The key difference between these two cases is that Metron executes a heavy packet processing NF (i.e., DPI), while OpenBox executes simple forwarding.

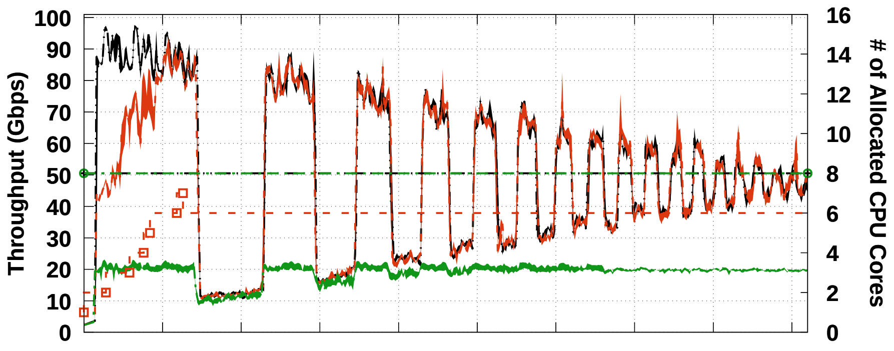
Key Outcome
We showed how network operators can in practice integrate Metron-based service chains with proprietary pieces of code executed by blackboxes, either as native processes or as VMs. With this important extension presented in §3, Metron can be used by production systems to dynamically and transparently manage & load balance *custom* software stacks at high performance, while still eliminating inter-core communication (outside of the blackboxes' scope). If a network operator desires even lower virtualization overhead at the same hardware-level performance, then container-based blackboxes should replace blackbox VMs, while using the same SR-IOV dispatching.

5.5 Metron's Dynamic Scaling at 100 Gbps

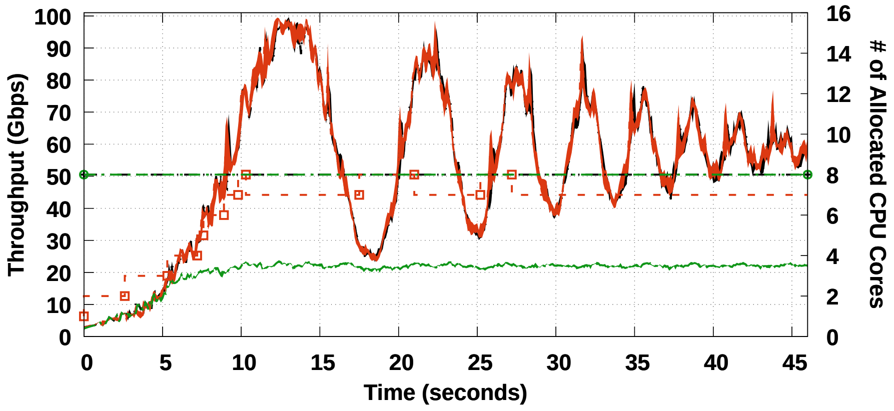
Using a similar Router→NAPT→LB service chain to the service chain used in §5.3.2, we evaluate Metron's dynamic scaling introduced in §4. In this experiment we inject the same campus trace, but used deeper analysis of this trace to derive a larger number of routes (i.e., 5462 rules vs. 3597 rules used in §5.3.2) for the router of the target service chain. We deployed this service chain on a single SkyLake server equipped with a Mellanox ConnectX-5 NIC, to which the campus trace was injected at an accelerated speed. We manually tuned the acceleration of the trace according to 3 different functions: (*i*) a constant load function at the speed of the hardware, i.e., 100 Gbps (Figure 16a), (*ii*) a square load function with increasing frequency and decreasing amplitude (Figure 16b), and (*iii*) a sinusoidal load function with increasing frequency and decreasing amplitude (Figure 16c).



(a) Constant load at the speed of a 100 GbE NIC.



(b) Square load change with increasing frequency and decreasing amplitude.



(c) Sinusoidal load change with increasing frequency and decreasing amplitude.

Fig. 16. Dynamic and static Metron vs. the RSS-based OpenBox under different types of dynamic workloads. Static Metron and OpenBox exhibit a static CPU core allocation of 8 cores. Dynamic Metron always begins with 1 core and allocates resources on demand. The borders of the areas filled with colors are defined by the 10th and 90th percentiles of the throughput across 10 runs.

The server under test runs the example service chain using three different systems. First, an RSS-based implementation of the service chain using OpenBox with 8 cores (depicted with green dashed-dotted lines showing throughput and circles showing the number of allocated CPU cores in Figure 16). Second, Metron with 8, statically allocated, CPU cores (depicted with black dashed and double dotted lines and crosses in Figure 16). Third, Metron with its dynamic CPU allocation scheme enabled (depicted with red dashed lines and squares in Figure 16). The throughput curves in Figure 16 are drawn using the median of the achieved throughput computed over 10 runs. To highlight the variance around the medians of the throughput, we fill the areas between the medians and the 10th and 90th percentiles accordingly. For better visibility, the CPU core allocation on the right hand vertical axis in Figure 16 shows the number of CPU cores allocated by each system over the course of a single execution of the experiment.

First we observe that in all cases the performance of the RSS-based OpenBox is far from the performance limit of the underlying hardware, despite using 8 CPU cores. In contrast, with the same CPU core allocation, Figure 16a shows that the static Metron is able to operate at the speed of the hardware, achieving up to 99-100 Gbps. This is confirmed by the "Hardware Limit RSS" curve (i.e., black crosses) in Figure 11b, measured on the same server with the same input trace.

The greatest advantage of Metron is its ability to adapt to changing workloads. This is shown by the performance of the dynamic Metron highlighted by the red lines and squares in Figure 16. In most cases, dynamic Metron accurately tracks the performance of its static counterpart (i.e., black lines and plus signs in Figure 16), while using only a fraction of the CPU cores used by static Metron. Specifically, in Figures 16a and 16b dynamic Metron deliberately begins with 1 CPU core and scales up to the throughput level of static Metron (i.e., 80-100 Gbps) within less than 5 seconds. After this point in time, dynamic and static Metron achieve similar performance despite any subsequent workload changes. Finally, in Figure 16c, the offered throughput increases less drastically towards the line-rate (i.e., between 0 and 13 seconds), which allows the dynamic Metron to completely match the performance of static Metron.

As summarized in Table 2, in all of these experiments dynamic Metron provisions resources on demand, thus achieving up to 5x higher throughput than OpenBox, while using 27.8-44.3% less CPU resources than OpenBox.

Table 2. Performance and CPU utilization gains of Metron vs. OpenBox during dynamic scaling.

Experiment	**Metron improvement over OpenBox (%)**	
	Throughput	**Average CPU Utilization**
Constant load (Figure 16a)	up to 500	27.8
Square load change (Figure 16b)		44.3
Sinusoidal load change (Figure 16c)		37.9

A current limitation of Metron is that it cannot load balance at a finer granularity than the traffic classes defined by SNF. This might be a problem when the derived traffic classes aggregate large subnets, which might potentially result in thousands (or even millions) of (concurrent) flows ending up at the same CPU core. In recent work (specifically RSS++ [6]), we look at ways to automatically derive sub-traffic classes of a given traffic class (i.e., by tweaking a NIC's RSS indireciton table) to perform load balancing even in the presence of a few (large) traffic classes.

5.6 Deployment Micro-benchmarks

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

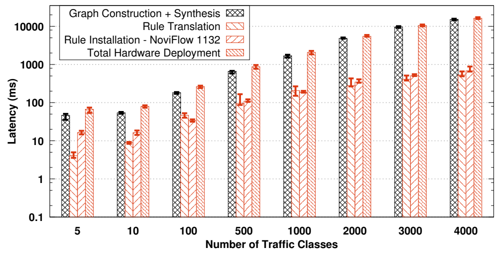
5.6.1 Impact of Increasing the number of Traffic Classes To study the impact of increasingly complex service chains on Metron's deployment latency, we use a firewall with an increasing number of rules (up to 4000), derived from actual Internet Service Provider (ISP) firewalls [102]. We measure the time between when a request to deploy this NF is issued by an application and the actual NF is deployed 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 17a and 17b. 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. Moreover, given the importance of generating such an optimized processing graph, Metron precomputes and stores the synthesized graph for a given input in its distributed database.

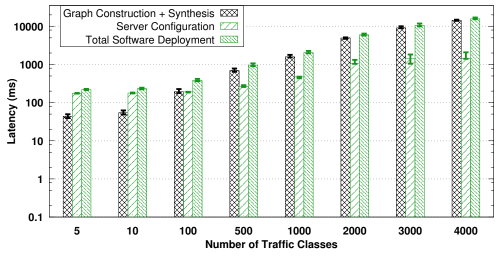
Apart from this fixed latency operation, a purely hardware-based deployment, requires two additional operations, as shown in Figure 17a. The first operation is the automatic translation of the firewall's synthesized packet processing graph into hardware instructions targeting our OpenFlow switch (the 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 17a). Note that even entry-level OpenFlow switches, such as the one used, can install thousands of rules per second; a more thorough study is provided in §5.6.2, where we discuss the effects of networking 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 service chain is deployed at a designated server. This latency is labeled "Server Configuration" in Figure 17b. Note that this 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 deployment time is 0.471 ms/rule (versus 0.459 ms/rule for the hardware case shown in Table 3), 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.



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



(b) Software-based deployment on a 16-core Intel Xeon E5-2667 v3.

Fig. 17. Latency (ms), on a logarithmic scale, for different Metron deployments with increasing complexity.

5.6.2 Diversity of Networking Hardware Networking hardware from different vendors and of different price levels might offer various possibilities for NFV offloading. In this section the hardware-based deployment shown in Figure 17a is repeated, but with the NoviFlow 1132 switch [81] replaced with (*i*) a 100 GbE Mellanox ConnectX-4 NIC [70], (*ii*) a 100 GbE Mellanox ConnectX-5 NIC [73], (*iii*) a 200 GbE Mellanox ConnectX-6 NIC [74], (*iv*) a 100 GbE Mellanox Bluefield NIC [72], (*v*) a 10 GbE Intel 82599 ES NIC [38], (*vi*) the software-based OVS [88], or (*vii*) a hybrid HP 5130 El hardware switch [34]. Figure 18 shows Metron's rule installation latency on these eight devices with increasing number of Internet Protocol version 4 (IPv4)-based L3/L4 rules. Table 3 summarizes the results of Figure 18 along with key characteristics of these devices, as they affect Metron's deployment choices and performance.

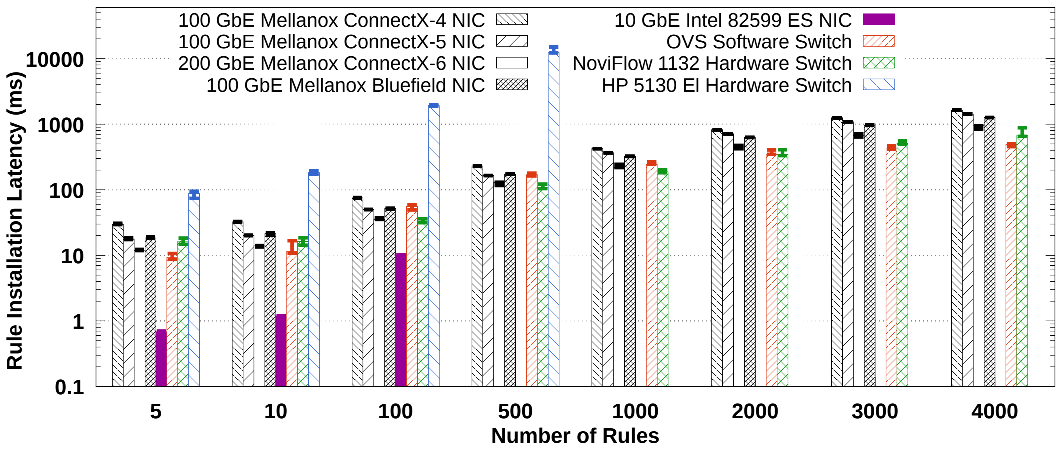


Fig. 18. Rule installation latency (in ms), on a logarithmic scale, of five DPDK-based NICs (a 100 GbE Mellanox ConnectX-4 [70], a 100 GbE Mellanox ConnectX-5 [73], a 200 GbE Mellanox ConnectX-6 [74], a 100 GbE Mellanox Bluefield [72], and a 10 GbE Intel 82599 ES [38]), as well as a software-based OpenFlow switch (OVS [88] v.2.5.2) and two hardware-based OpenFlow switches (a NoviFlow 1132 [81] and an HP 5130 El [34]). The default table (i.e., table 0) is used to store the input rules on all of these devices. The Intel 82599 ES NIC and the HP 5130 El switch can accommodate up to 128 and 512 IPv4-based L3/L4 rules respectively.

Table 3. Comparison of 5 NICs and 3 switches used by Metron as offloading devices. The median and average rule installation latencies per rule (i.e., rule installation speeds) are computed using the median values from the bars shown in Figure 18.

Device		**Speed (ms/rule)**		**Capacity**
Model	**Type**	**Median**	**Average**	**(Number of Rules)**
10 GbE Intel 82599 ES [38]	HW	0.119	0.119	128
100 GbE Mellanox ConnectX-4 [70]	HW	0.443	1.505	65536
100 GbE Mellanox ConnectX-5 [73]	HW	0.364	0.975	Memory-bounded
200 GbE Mellanox ConnectX-6 [74]	HW	0.247	0.664	Memory-bounded
100 GbE Mellanox Bluefield [72]	HW	0.335	0.995	Memory-bounded
OVS v2.5.2 [88]	SW	0.286	0.343	Memory-bounded
NoviFlow 1132 [81]	HW	0.203	0.459	225280
HP 5130 El [34]	HW	9.93	11.33	256/512/16384

Switches: The NoviFlow switch contains 55 OpenFlow tables, each with 4096 entries (i.e., 225280 rules in total), while the HP switch has a single OpenFlow table with either 512/256 entries for IPv4/Internet Protocol version 6 (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 average rule installation speed of the NoviFlow 1132 switch is substantially higher than the HP 5130 El (0.459 vs. 11.33 ms/rule), with the difference being more than an order of magnitude. Moreover, this difference is reflected in the price difference between the two switches (approximately US$15 000 vs. US$2000). In contrast, OVS is open source software and has lower data plane performance, but outperforms both hardware-based switches in terms of average rule installation speed (0.343 ms/rule), when running on the processor described for the testbed in §5.2. This finding is confirmed by earlier studies [61, 62], where the rule installation speed varied, especially when priorities were involved. However, in this experiment Metron installed rules of the same priority and low variance was observed. OVS achieves the fastest average rule installation speed out of all the switches tested with 0.343 ms/rule (i.e., 28.9% faster than the Noviflow 1132).

NICs: According to Intel's documentation [37], the Intel 82599 ES NIC can only accommodate up to 128 IPv4-based L3/L4 rules. We verified this observation by conducting our own benchmarks using DPDK 19.11 and DPDK's flow API [104]. As the capacity of the Mellanox (i.e., ConnectX-4, ConnectX-5, ConnectX-6, and Bluefield) NICs is not disclosed by Mellanox, we performed additional benchmarks with DPDK 19.11 and Mellanox OFED 4.7.1 firmware version, in order to infer the capacity of these NICs by installing an increasing number of rules. The outcome of this experiment revealed that all four Mellanox NICs could accommodate up to 65536 IPv4-based L3/L4 rules into table 0; this capacity is substantially greater than Intel's 82599 ES. While the Mellanox ConnectX-4 NIC uses a single flow table, the ConnectX-5, ConnectX-6, and Bluefield NICs allow creation of multiple subsequent flow tables (i.e., groups), each with a much larger capacity than table 0. To extend the pipeline of these NICs with a new table, one stores a rule that matches the desired traffic and then associates this match operation with a 'jump group X' action, where X is the number of the table to jump into [105]. We progressively increased the number of installed flow rules into table 1 (of each NIC) up to 6 million, without exhausting the NIC's capacity; hence, we conjecture that these NICs use in-NIC storage for additional flow tables. In addition to the large flow rule capacity of these NICs, all tables beyond table 0 have higher rule installation speeds. According to a recent study, the rule installation performance of the Mellanox NICs in table 1 can reach rates beyond half a million of rules per second using a single CPU core to inject NIC rules [47]. This performance can be further improved using multiple cores for rule injection [106].

As some of the NICs under test possess a single flow table, in the rest of this section, we compare these NICs solely based upon the data shown in Table 3, which reports the rule installation speeds using the default table 0 and solely IPv4-based L3/L4 rules. In this case, the Intel 82599 ES NIC outperforms all the other devices with its average rule installation speed of 0.119 ms/rule. Despite its limited capacity, this NIC's rule installation speed is almost 4 times faster on average than the NoviFlow 1132 switch, while being several orders of magnitude lower in cost (and port density). The Mellanox ConnectX-5 and Bluefield NICs have similar rule installation speeds (the latter uses a ConnectX-5 network adapter [72]), which is half of the average rule installation speed achieved by the NoviFlow 1132 switch (i.e., 0.975 vs. 0.459 ms/rule). The older generation ConnectX-4 is on average 40% slower (in terms of rule installation speed) than its two successors (ConnectX-5 and Bluefield) and almost 2.3x times slower than the latest ConnectX-6 NIC.

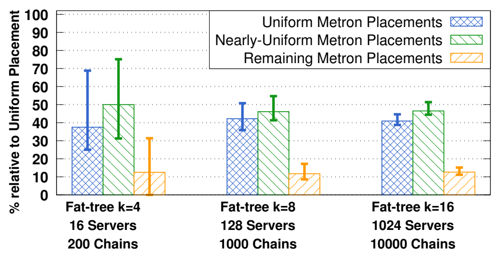
5.7 Metron's Placement in Large Networks

In this section we test Metron's placement scheme, presented in §2.3.3, on a set of topologies with a large number of nodes, on which we deploy hundreds to thousands of service chains.

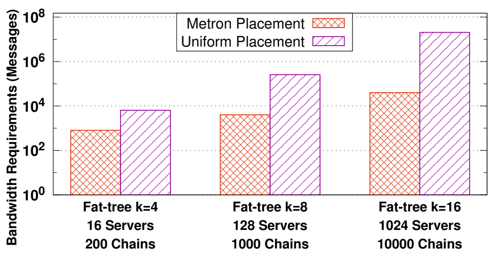
To verify that the performance of our placement scheme can be generalized to real and potentially large networks, we conducted experiments that emulate Metron's service chain placement in datacenters, using fat-tree topologies of increasing sizes (see Figure 19). Our analytic study shows how close Metron's 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 scheme is topology-agnostic.

Figure 19a 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 matches 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 19b 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 19b. 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 19b 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.



(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 in terms of number of messages exchanged between the controller and the servers, on a logarithmic scale.

Fig. 19. 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.

6 Related Work

Here, we discuss related works that were not mentioned earlier in this article.

6.1 NFV Management

E2 [90] and Metron [48] both manage service chains mapped to clusters of servers interconnected via programmable switches. However, E2 only partially exploits OpenFlow switches to perform traffic steering. In contrast, Metron fully exploits the network (both 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.

Inspired by the SDN networking paradigm, OpenBox [16] decouples the control plane of NFs from the data plane using the OpenBox protocol. However, the OpenBox protocol lacks important abstractions with regard to managing NIC resources of commodity servers. In contrast, the Metron protocol, introduced in Appendix A, extends the OpenBox protocol in this important direction. The Metron controller monitors NIC resources and performs service chain offloading (i.e., by installing/deleting NIC rules) using the Metron protocol. A technical description of the Metron protocol is available at [45].

6.2 NFV Consolidation

OpenBox [16] merges similar packet processing elements into one, thus reducing redundancy. Slick [3] and CoMb [98] propose NF consolidation schemes, although these schemes reside higher in the network stack. mOS [39] introduces stateful flow processing in middlebox applications through a reusable networking stack. Microboxes [65] propose a customizable Transmission Control Protocol (TCP) stack for various types of middleboxes ranging from lightweight TCP state monitoring to proxies with full TCP termination. MiddleClick [7] is a set of low-level solutions for building NFV service chains with unified abstractions for TCP session management and flow classification. This is done by introducing a per-session per-NF "scratchpad", which acts as dedicated memory to store and quickly lookup NF state information [4]. Similar to MiddleClick, SNF [49] eliminates processing redundancy by synthesizing multiple NFs as an optimized equivalent NF. A more detailed performance evaluation of SNF is provided in [44].

We integrated SNF into Metron, as it is the most extensive consolidation scheme to date. Metron effectively coordinates these optimized pipelines on a large-scale, while attempting to fully exploit the underlying hardware. MiddleClick could be integrated into Metron in the future, bringing additional capabilities, such as unified TCP flow state management.

6.3 Hardware Programmability

During the last decade, there has been a major effort to increase hardware programmability. OpenFlow [67] paved the way by enriching the programmability of switches using simple match-action rules. Increasingly, NICs are equipped with hardware features, such as RSS and Flow Director, for dispatching packets directly from the NIC to a specific CPU core.

In an attempt to overcome the static nature of the above solutions, more flexible programmability models have emerged. Reconfigurable Match Tables (RMT) [15] and its successor P4 [14] are prime examples of protocol-independent packet processors, while OpenState [10] and Open Packet Processor (OPP) [11] showed how OpenFlow can become stateful with minimal but essential modifications. FlexNIC [52] proposed a model for additional programmability in future NICs.

All these works further expose the hardware's 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.

6.4 Hardware Offloading

The rapid increase of link speeds has increased the complexity and processing requirements of software-based network stacks and functions, despite the cloud providers' efforts to built increasingly well-tuned and efficient host SDN packet processing solutions [27]. Consequently, cloud providers are striving for ways to offload network services to programmable hardware, thus dedicating CPU resources to application-layer processing. In the rest of this section, we first present today's hardware offloading developments and then discuss how Metron expands to account for these developments.

Commodity NICs and Graphics Processing Units (GPUs): For increased performance, Raumer et al. [93] offloaded the cryptographic function of a Virtual Private Network (VPN) gateway to commodity NICs. HyperLoop [54] leverages Remote Direct Memory Access (RDMA) to remove the CPU from the critical path of replicated transactions in storage systems. PacketShader [32], Kargus [40], NBA [56], and APUNet [30] take advantage of inexpensive but powerful GPUs to offload and accelerate packet processing. We envision these works to be future components of Metron, thus extending its offloading abilities.

Recently, RSS++ [6] exploited available commodity NICs' functionality to achieve stateful intra-server load balancing with minimal overhead. Metron dispatches traffic through explicit (NIC and/or OpenFlow) rules, while RSS++ steers flows to cores by modifying a NIC's RSS indirection table. When *no* hardware offloading can be exploited, Metron can readily leverage RSS++.

Smart NICs: Certain types of packet processing functions are not supported by commodity NICs, therefore networking hardware vendors offer NICs with advanced processing capabilities, also called Smart NICs [77–79]. Examples of Smart NICs' advanced features are: (*i*) OVS offload and acceleration [77, 94], (*ii*) cryptography for various cipher suites and key sizes, (*iii*) DPI [78], (*iv*) *stateful* load balancing, and (*v*) virtual evolved packet core functions. For example, AccelNet [27] is Microsoft Azure's custom Smart NIC solution for offloading host networking to hardware, using Field-Programmable Gate Arrays (FPGAs). ClickNP [63] showed how to achieve high performance packet processing by completely migrating NFV into reconfigurable, but specialized hardware.

Programmable switches: SwitchKV [64] offloads key-value stores into OpenFlow switches. As traditional Application-Specific Integrated Circuits (ASICs) or OpenFlow-based hardware cannot maintain per-connection state, SilkRoad [75] exploits *programmable* switch ASICs to provide high performance layer-4 load balancing. SilkRoad builds upon earlier load balancing efforts [51]. Dejavu [111] argues about offloading entire service chains to programmable networking hardware, i.e., a Tofino-based P4 switch ASIC. With Dejavu, edge cloud providers could dedicate their (limited) CPU resources to other types of applications, while using programmable ASICs to accommodate their packet processing requirements.

Our strategy for future developments: Metron provides full support for OpenFlow switches and all sorts of NICs that support DPDK. Such NICs span across commodity devices (e.g., 10GbE Intel 82599 [38]), Smart NICs (e.g., Mellanox Bluefield [72] or Bluefield-2 [71], and Netronome Agilo series [78–80]), while also supporting DPDK-based FPGAs as provided by e.g., Netcope [77]. P4 support is the next target of our prototype. We aim to achieve P4 compatibility through ONOS's P4 primitives [86] and the Stratum project [87].

6.5 Server-level Solutions

Flurries [114] builds atop OpenNetVM [115] to provide software-based service chains on a per-flow basis, while ClickOS [66] and NetVM [35] offer NFs running in VMs. NFP [100] extends OpenNetVM to allow NFs in a service chain to be executed in parallel. Dysco [113] proposes a distributed protocol for steering traffic across the NFs of a service chain. NFVnice [60] and SCC [42, 50] are efficient NFV schedulers, optimized for high throughput and low latency respectively. Click-based [58] approaches have proposed techniques to exploit multi-core architectures [8, 55, 101]. However, none of these earlier approaches have explored the possibility of using hardware to offload parts of a service chain nor do they support our optimized flow affinity mechanism.

6.6 Industrial Efforts

European Telecommunications Standards Institute (ETSI) has been driving NFV standardization over the last few years [25]. ETSI's specialized group [26] uses OpenStack [89] as an open implementation of the current NFV standards, based on a generic framework for managing compute, storage, and network resources. Central Office Re-architected as a Datacenter (CORD) [84] and Open Platform for NFV (OPNFV) [108] are deployed on top of OpenStack and/or Kubernetes [107]. CORD leverages SDN and NFV to build agile datacenters for the network edge, while OPNFV facilitates the interoperability of NFV components across various open source ecosystems. Metron and CORD share common controller abstractions (i.e., ONOS); however, we avoid virtualization by integrating native DPDK-based solutions. Unlike CORD, Metron employs placement techniques (§2.3.3) with minimal overhead (§5.6) and sophisticated NF consolidation (§2.3.1) to achieve high performance.

7 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 commodity programmable hardware devices. Doing so offers dramatic hardware efficiency and performance increases over the state of the art. Metron solves an important problem of the networking industry by transparently integrating, managing, and load balancing blackbox NF implementations, while providing dynamic on demand resource allocation under highly-variable workload volumes at 100 Gbps. 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 100 GbE NICs.

We envision Metron to be a key component of the emerging 100 GbE deployments, therefore we encourage the networking industry and academic community to use and contribute to our open source prototype.

8 Acknowledgments

We thank the anonymous referees, the associate editor, and the editor-in-chief Michael Swift, for their insightful comments on earlier drafts of this manuscript. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 770889). This work was also funded by the Swedish Foundation for Strategic Research (SSF).

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* (New York, NY, USA, 2008), SIGCOMM '08, ACM, pp. 63–74. http://doi.acm.org/10.1145/1402958.1402967.

[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* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 19–19. https://static.usenix.org/events/nsdi10/tech/full_papers/al-fares.pdf.

[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* (New York, NY, USA, 2015), SOSR '15, ACM, pp. 14:1–14:13. http://doi.acm.org/10.1145/2774993.2774998.

[4] Barbette, T. *Architecture for programmable network infrastructure.* Doctoral thesis, University of Liege, Faculty of Applied Sciences, Department of Electricity, Electronics and Informatics, Liege, Belgium, July 2018. http://hdl.handle.net/2268/226257.

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

[6] Barbette, T., Katsikas, G. P., Maguire, Jr., G. Q., and Kostić, D. RSS++: load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (New York, NY, USA, 2019), CoNEXT '19, ACM, pp. 318–333. http://doi.acm.org/10.1145/3359989.3365412.

[7] Barbette, T., Soldani, C., Gaillard, R., and Mathy, L. Building a chain of high-speed VNFs in no time. In *Proceedings of the IEEE International Conference on High Performance Switching and Routing* (2018), HPSR'18. https://www.tombarbette.be/wp-content/uploads/2018/10/barbette.pdf.

[8] 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* (Washington, DC, USA, 2015), ANCS '15, IEEE Computer Society, pp. 5–16. https://doi.org/10.1109/ANCS.2015.7110116.

[9] 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* (New York, NY, USA, 2014), HotSDN '14, ACM, pp. 1–6. http://doi.acm.org/10.1145/2620728.2620744.

[10] Bianchi, G., Bonola, M., Capone, A., and Cascone, C. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *SIGCOMM Comput. Commun. Rev. 44*, 2 (Apr. 2014), 44–51. http://doi.acm.org/10.1145/2602204.2602211.

[11] 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). https://arxiv.org/abs/1605.01977.

[12] 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.

[13] Boon Ang et al. Single Root I/O Virtualization and Sharing Specification Revision 1.1, Jan. 2010. https://composter.com.ua/documents/sr-iov1_1_20Jan10_cb.pdf.

[14] 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. http://doi.acm.org/10.1145/2656877.2656890.

[15] 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. *SIGCOMM Comput. Commun. Rev. 43*, 4 (Aug. 2013), 99–110. https://doi.org/10.1145/2534169.2486011.

[16] 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* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 511–524. http://doi.acm.org/10.1145/2934872.2934875.

[17] 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.

[18] 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. http://dx.doi.org/10.1109/TNET.2011.2159308.

[19] 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.

[20] Cisco. Cisco CSR 1000v and Cisco ISRv Software Configuration Guide, 2020. https://www.cisco.com/c/en/us/td/docs/routers/csr1000/software/configuration/b_CSR1000v_Configuration_Guide/b_CSR1000v_Configuration_Guide_chapter_010001.html.

[21] 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. https://doi.org/10.1109/NETSOFT.2015.7116144.

[22] 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* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 15–28. http://doi.acm.org/10.1145/1629575.1629578.

[23] Eckert, A., MartinGarcia, L., Niazmand, R., and Wang, X. Wedge 100: More open and versatile than ever, Oct. 2016. https://code.facebook.com/posts/1802489260027439/wedge-100-more-open-and-versatile-than-ever/.

[24] 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.

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

[26] European Telecommunications Standards Institute (ETSI). Open Source NFV Management and Orchestration (MANO) , 2020. https://osm.etsi.org/.

[27] Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., Chandrappa, H. K., Chaturmohta, S., Humphrey, M., Lavier, J., Lam, N., Liu, F., Ovtcharov, K., Padhye, J., Popuri, G., Raindel, S., Sapre, T., Shaw, M., Silva, G., Sivakumar, M., Srivastava, N., Verma, A., Zuhair, Q., Bansal, D., Burger, D., Vaid, K., Maltz, D. A., and Greenberg, A. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association, pp. 51–66. https://www.usenix.org/system/files/conference/nsdi18/nsdi18-firestone.pdf.

[28] 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* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 163–174. http://doi.acm.org/10.1145/2619239.2626313.

[29] Gilad Shainer, Network Computing. 100 Gbps Headed For The Data Center, Nov. 2014. https://www.networkcomputing.com/data-centers/100-gbps-headed-data-centers.

[30] Go, Y., Jamshed, M. A., Moon, Y., Hwang, C., and Park, K. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *14th USENIX Symposium on Networked Systems Design and Implementation* (2017), NSDI'17, USENIX Association, pp. 83–96. https://www.usenix.org/system/files/conference/nsdi17/nsdi17-go.pdf.

[31] 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. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[32] Han, S., Jang, K., Park, K., and Moon, S. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 195–206. http://doi.acm.org/10.1145/1851182.1851207.

[33] 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. http://doi.acm.org/10.1145/1544012.1544027.

[34] Hewlett Packard. HPE FlexNetwork 5130 EI Switch Series, Jan. 2017. https://h20195.www2.hpe.com/v2/getpdf.aspx/c04394228.pdf.

[35] 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* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 445–458. https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-hwang.pdf.

[36] Intel. Improving Network Performance in Multi-Core Systems, 2007. http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf.

[37] Intel. Introduction to Intel® Ethernet Flow Director and Memcached Performance, 2014. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf.

[38] Intel. 82599 10 GbE Controller Datasheet, 2016. http://www.intel.com/content/www/us/en/embedded/products/networking/82599-10-gbe-controller-datasheet.html.

[39] Jamshed, M., Moon, Y., Kim, D., Han, D., and Park, K. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2017), NSDI'17, pp. 113–129. https://www.usenix.org/system/files/conference/nsdi17/nsdi17-jamshed.pdf.

[40] 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. http://doi.acm.org/10.1145/2382196.2382232.

[41] 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* (2017), NSDI'17, pp. 97–112. https://www.usenix.org/system/files/conference/nsdi17/nsdi17-kablan.pdf.

[42] Katsikas, G. P. *Realizing High Performance NFV Service Chains.* Licentiate thesis, KTH Royal Institute of Technology, School of Information and Communication Technology, Kista, Sweden, Nov. 2016. TRITA-ICT 2016:35, http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-195352.

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

[44] Katsikas, G. P. *NFV Service Chains at the Speed of the Underlying Commodity Hardware.* Doctoral thesis, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Kista, Sweden, Sept. 2018. TRITA-EECS-AVL-2018:50,http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-233629.

[45] Katsikas, G. P., and Barbette, T. ONOS Server Device Driver Tutorial, 2018. https://wiki.onosproject.org/display/ONOS/Server+Device+Driver+Tutorial.

[46] Katsikas, G. P., and Barbette, T. Metron control plane as an ONOS application, 2020. https://github.com/gkatsikas/onos/tree/metron-ctrl/apps/metron.

[47] Katsikas, G. P., Barbette, T., Chiesa, M., Kostić, D., and Maguire Jr., G. Q. What You Need to Know About (Smart) Network Interface Cards. In *Passive and Active Measurement*, O. Hohlfeld, A. Lutu, and D. Levin, Eds., PAM '21. Springer International Publishing, 2021, pp. 319–336. https://www.pam2021.b-tu.de/papers/10.1007978-3-030-72582-2_19.pdf.

[48] Katsikas, G. P., Barbette, T., Kostić, D., Steinert, R., and Maguire Jr., G. Q. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, 2018), NSDI'18, USENIX Association, pp. 171–186. https://www.usenix.org/system/files/conference/nsdi18/nsdi18-katsikas.pdf.

[49] 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.

[50] 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.

[51] Katta, N., Hira, M., Kim, C., Sivaraman, A., and Rexford, J. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2016), SOSR '16, ACM, pp. 10:1–10:12. http://doi.acm.org/10.1145/2890955.2890968.

[52] 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* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 67–81. http://doi.acm.org/10.1145/2872362.2872367.

[53] 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. https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-khalid.pdf.

[54] Kim, D., Memaripour, A., Badam, A., Zhu, Y., Liu, H. H., Padhye, J., Raindel, S., Swanson, S., Sekar, V., and Seshan, S. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 297–312. http://doi.acm.org/10.1145/3230543.3230572.

[55] 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* (New York, NY, USA, 2012), APSYS '12, ACM, pp. 14:1–14:6. http://doi.acm.org/10.1145/2349896.2349910.

[56] 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* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 22:1–22:14. http://doi.acm.org/10.1145/2741948.2741969.

[57] Kim, J. F. Mellanox Blog: 25 Is the New 10, 50 Is the new 40, 100 Is the New Amazing, Mar. 2016. http://www.mellanox.com/blog/2016/03/25-is-the-new-10-50-is-the-new-40-100-is-the-new-amazing/.

[58] 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. http://doi.acm.org/10.1145/354871.354874.

[59] Krishnan, R., Durrani, M., and Phaal, P. Real-time SDN and NFV Analytics for DDoS Mitigation, 2014. https://blog.sflow.com/2014/02/nfd7-real-time-sdn-and-nfv-analytics_1986.html.

[60] 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. http://doi.acm.org/10.1145/3098822.3098828.

[61] 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.

[62] 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.

[63] 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, ACM, pp. 1–14. http://doi.acm.org/10.1145/2934872.2934897.

[64] 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* (Berkeley, CA, USA, 2016), NSDI'16, USENIX Association, pp. 31–44. https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-li_xiaozhou-update.pdf.

[65] Liu, G., Ren, Y., Yurchenko, M., Ramakrishnan, K. K., and Wood, T. Microboxes: High Performance NFV with Customizable, Asynchronous TCP Stacks and Dynamic Subscriptions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 504–517. http://doi.acm.org/10.1145/3230543.3230563.

[66] 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* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 459–473. https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-martins.pdf.

[67] 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. http://doi.acm.org/10.1145/1355734.1355746.

[68] Mellanox. Mellanox ASAP^2: Accelerated Switching and Packet Processing, 2017. https://www.mellanox.com/related-docs/products/SB_asap2.pdf.

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

[70] Mellanox. ConnectX®-4 EN Card 100Gb/s Ethernet Adapter Card, 2018. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_EN_Card.pdf.

[71] Mellanox. BlueField-2® SmartNIC for InfiniBand & Ethernet, 2019. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField-2_SmartNIC_VPI.pdf.

[72] Mellanox. BlueField® SmartNIC for Ethernet, 2019. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[73] Mellanox. ConnectX®-5 EN Card 100Gb/s Ethernet Adapter Card, 2019. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-5_EN_Card.pdf.

[74] Mellanox. ConnectX®-6 EN IC 200GbE Ethernet Adapter IC, 2019. https://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-6_EN_IC.pdf.

[75] Miao, R., Zeng, H., Kim, C., Lee, J., and Yu, M. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 15–28. https://doi.org/10.1145/3098822.3098824.

[76] Mitzenmacher, M. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst. 12*, 10 (Oct. 2001), 1094–1104. http://dx.doi.org/10.1109/71.963420.

[77] Netcope Technologies. Netcope P4 Cloud: Online P4 to FPGA synthesis and in-hardware evaluation, 2020. https://www.netcope.com/en/products/netcopep4.

[78] Netronome. Agilio LX 1x100GbE SmartNIC, 2018. https://www.netronome.com/m/documents/PB_Agilio_Lx_1x100GbE-7-20.pdf.

[79] Netronome. Agilio CX SmartNICs, 2020. https://www.netronome.com/products/agilio-cx/.

[80] Netronome. Agilio FX SmartNICs, 2020. https://www.netronome.com/products/agilio-fx/.

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

[82] 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. http://doi.acm.org/10.1145/2342356.2342376.

[83] Ongaro, D., and Ousterhout, J. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 305–320. https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf.

[84] Open Networking Foundation (ONF). Central Office Re-architected as a Datacenter (CORD), 2020. https://www.opennetworking.org/cord/.

[85] Open Networking Foundation (ONF). Open Network Operating System (ONOS), 2020. http://onosproject.org/.

[86] Open Networking Foundation (ONF). P4 brigade, 2020. https://wiki.onosproject.org/display/ONOS/P4+brigade.

[87] Open Networking Foundation (ONF). Stratum, 2020. https://www.opennetworking.org/stratum/.

[88] Open vSwitch. An Open Virtual Switch. http://openvswitch.org.

[89] OpenStack. Open Source Cloud Computing Software, 2020. https://www.openstack.org/.

[90] 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* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 121–136. http://doi.acm.org/10.1145/2815400.2815423.

[91] 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* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 335–348. http://doi.acm.org/10.1145/1755913.1755947.

[92] Pfaff, B., Pettit, J., Koponen, T., Jackson, E. J., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., and Casado, M. The Design and Implementation of Open vSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 117–130. https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-pfaff.pdf.

[93] 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). https://ieeexplore.ieee.org/document/7776595.

[94] Renwick, R. Increase Application Performance with SmartNICs, 2017. https://www.openstack.org/assets/presentation-media/Netronome-OpenStack-Summit-Marketplace-presentation.pdf.

[95] Robison, C. B. How to Set Up Intel Ethernet Flow Director, June 2017. https://software.intel.com/en-us/articles/setting-up-intel-ethernet-flow-director.

[96] Rosen, E. C., Viswanathan, A., and Callon, R. Multiprotocol Label Switching Architecture. Internet Request for Comments (RFC) 3031, Jan. 2001. Updated by RFCs 6178, 6790, https://www.rfc-editor.org/rfc/rfc3031.txt.

[97] Schmidtke, Katharine. Facebook: Designing 100G optical connections, Mar. 2017. https://code.facebook.com/posts/1633153936991442/designing-100g-optical-connections/.

[98] 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* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 24–24. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final96.pdf.

[99] Snort. Network Intrusion Detection & Prevention System, 2020. https://www.snort.org/.

[100] 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. http://doi.acm.org/10.1145/3098822.3098826.

[101] 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. http://dl.acm.org/citation.cfm?id=2537857.2537861.

[102] Taylor, D. E., and Turner, J. S. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Trans. Netw. 15*, 3 (June 2007), 499–511. http://dx.doi.org/10.1109/TNET.2007.893156.

[103] The Linux Foundation. Data Plane Development Kit (DPDK). http://dpdk.org.

[104] The Linux Foundation. DPDK Generic flow API, 2020. https://doc.dpdk.org/guides/prog_guide/rte_flow.html.

[105] The Linux Foundation. DPDK MLX5 poll mode driver, 2020. https://doc.dpdk.org/guides/nics/mlx5.html.

[106] The Linux Foundation. DPDK's Flow Performance Tool, 2020. https://doc.dpdk.org/guides/tools/flow-perf.html.

[107] The Linux Foundation. Kubernetes, 2020. https://kubernetes.io/.

[108] The Linux Foundation. Open Platform for NFV (OPNFV), 2020. https://www.opnfv.org/.

[109] 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.

[110] Woo, S., Sherry, J., Han, S., Moon, S., Ratnasamy, S., and Shenker, S. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association, pp. 299–312. https://www.usenix.org/system/files/conference/nsdi18/nsdi18-woo.pdf.

[111] Wu, D., Chen, A., Ng, T. S. E., Wang, G., and Wang, H. Accelerated Service Chaining on a Single Switch ASIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2019), HotNets '19, ACM, pp. 141–149. http://doi.acm.org/10.1145/3365609.3365849.

[112] 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. http://doi.acm.org/10.1145/1355734.1355737.

[113] 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. http://doi.acm.org/10.1145/3098822.3098827.

[114] 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* (New York, NY, USA, 2016), CoNEXT '16, ACM, pp. 3–17. http://doi.acm.org/10.1145/2999572.2999602.

[115] 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. http://faculty.cs.gwu.edu/~timwood/papers/16-HotMiddlebox-onvm.pdf.

A Metron Protocol

The Metron controller needs to manage the resources of the underlying network. Such a network might be comprised of programmable network elements (e.g., switches and routers) and/or servers with programmable CPU cores and NICs. In the former case, the Metron controller relies on existing network management protocols, such as OpenFlow [67], NETCONF [24], etc. Due to the absence of protocols to manage CPU and NIC resources on servers, Metron introduces its own protocol [45] and driver [43] to manage these resources, through a set of messages shown in Table 4.

Table 4. The Metron protocol specification for managing programmable resources on commodity servers.

Channel	**Message**	
	Name	**Payload**
Controller ↓ Server	FEATURES_REQ	Server ID
	GLOBAL_STATS_REQ	Server ID
	SC_DEPLOY_REQ	Server ID, Service Chain ID, SW Configuration
	SC_RECONF_REQ	Server ID, Service Chain ID, SW Reconfiguration
	SC_STATS_REQ	Server ID, Service Chain ID
	SC_DELETE_REQ	Server ID, Service Chain ID
	RULES_MONITOR_REQ	Server ID, NIC ID
	RULES_INSTALL_REQ	Server ID, NIC ID, [Rule IDs, Rules]
	RULES_DELETE_REQ	Server ID, NIC ID, [Rule IDs]
	SET_CTRL_REQ	Server ID, Controller IP address & Port
	GET_CTRL_REQ	Server ID
	DEL_CTRL_REQ	Server ID, Controller IP address & Port
Server ↓ Controller	SERVER_REGISTER	Server ID, Server IP address & Port
	FEATURES_REP	Server ID, Server features
	GLOBAL_STATS_REP	Server ID, NIC & CPU Statistics
	SC_DEPLOY_REP	Server ID, Service Chain ID, Status
	SC_RECONF_REP	Server ID, Service Chain ID, Status
	SC_STATS_REP	Server ID, Service Chain ID & Statistics
	SC_DELETE_REP	Server ID, Service Chain ID, Status
	RULES_MONITOR_REP	Server ID, NIC ID, [Existing Rule IDs]
	RULES_INSTALL_REP	Server ID, NIC ID, [Installed Rule IDs]
	RULES_DELETE_REP	Server ID, NIC ID, [Deleted Rule IDs]
	SET_CTRL_REP	Server ID, Status
	GET_CTRL_REP	Server ID, Controller IP address & Port
	DEL_CTRL_REP	Server ID, Status

When a server boots, a "SERVER_REGISTER" message is sent to the controller. This message contains the IP address and port that the server uses to communicate with the controller. The controller replies with a "FEATURES_REQ" message, asking the server for system-related features. The three-way handshaking is completed after a "FEATURES_REP" from the server that contains its ID, serial number, manufacturer, hardware and software versions. Moreover, the server advertises the ID, vendor, and frequency of its CPU cores along with the technical characteristics of its NICs. These characteristics include the ID, vendor, driver, speed, port type, hardware address, and traffic dispatching abilities of each NIC. Note that modern commodity NICs offer different traffic dispatching abilities, such as hash-based (i.e., using RSS) or flow-based (i.e., using NIC rules) dispatching. As we introduced in §2.3, Metron exploits the deterministic nature of flow-based dispatching to match and redirect each input packet to a designated CPU core, thus eliminating inter-core packet transfers.

At this point the controller is fully aware of the available resources of a server and this information is stored in Metron's distributed storage, as described in §2.3.4. A "GLOBAL_STATS_REQ" message is periodically issued by the controller to obtain global monitoring statistics, such as NIC and CPU load counters. These counters are encoded in a "GLOBAL_STATS_REP" message, which is sent as a response by the server. The controller can deploy a service chain by sending an "SC_DEPLOY_REQ" together with a service chain ID and the software part of the service chain's configuration. This configuration contains the packet processing graph to be instantiated by the server, the number of CPUs required for this service chain, and the NIC IDs associated with the I/O vertices of the service chain's processing graph. If the deployed service chain contains packet processing operations that can be offloaded to the server's NIC(s), the "SC_DEPLOY_REQ" is also followed by a "RULES_INSTALL_REQ", which encodes these operations as NIC rules. Once the server installs these rules in the corresponding NIC(s), a "RULES_INSTALL_REP" is sent back to the controller to advertise the installed rules per NIC. When the service chain is fully deployed, the server replies to the controller with an "SC_DEPLOY_REP". After a service chain has been deployed, the controller can ask for service chain-specific run-time statistics by issuing an "SC_STATS_REQ" message with the ID of the service chain. The Metron agent on the server collects NIC and CPU core statistics (e.g., transmitted/received/dropped packets/bytes and per-core load) related to this particular service chain and reports them with an "SC_STATS_REP" message.

If a service chain exhibits a load imbalance, then a scaling operation can be requested by the controller by issuing an "SC_RECONF_REQ" message. The payload of this message contains the ID of the service chain to be load balanced and the reconfiguration instructions to be executed by the server that runs the software part of the service chain. For example, in §4 we discuss how the controller splits traffic classes across multiple groups, each associated with a different tag, when the load of these traffic classes exceeds a predefined threshold. In such a case, the controller will instruct the server (through an "SC_RECONF_REQ" message) to allocate an additional CPU core, on which the stateful processing graph of the split traffic classes will be instantiated. A reconfiguration is complete when the dispatching policy for the split traffic classes is updated in the hardware via a sequence of "RULES_INSTALL_REQ" and "RULES_DELETE_REQ" messages. These messages will install the necessary rules to realize the updated dispatching policy, while removing the old rules that correspond to the target traffic classes. That said, incoming packets that belong to the affected traffic classes will now be tagged differently, thus dispatched to less loaded CPU cores.

To tear down a service chain, the controller issues both an "SC_DELETE_REQ" with the ID of the service chain to be torn down and a "RULES_DELETE_REQ" to remove all of the rules installed by this service chain. The server reacts by removing those rules ("RULES_DELETE_REP"), while killing the software process responsible for the stateful part of this service chain ("SC_DELETE_REP").

Finally, the Metron protocol allows dynamic (re-)association of servers with controller instances. To do so, a network operator issues a "SET_CTRL_REQ" to the Metron controller's RESTful API. This request contains a server's ID and the controller's IP address and port, triggering a corresponding request from the controller to the designated server. Once a server receives such a message, it attempts to associate with the controller by initiating a three-way handshaking process ("SERVER_REGISTER", "FEATURES_REQ', and "FEATURES_REP") as described above. Then, the server responds with a "SET_CTRL_REP", which informs the controller of the status of the association command. Similar messages are offered to observe the association status of a server (i.e., through the "GET_CTRL_REQ" and "GET_CTRL_REP") as well as to disassociate a server with a controller (i.e., through the "DEL_CTRL_REQ" and "DEL_CTRL_REP" messages).

Received December 2018; revised December 2019; accepted May 2021