E2: A Framework for NFV Applications

Shoumik Palkar *
UC Berkeley
sppalkar@berkeley.edu

Chang Lan*
UC Berkeley
clan@eecs.berkeley.edu

Sangjin Han
UC Berkeley
sangjin@eecs.berkeley.edu

Keon Jang
Intel Labs
keon.jang@intel.com

Aurojit Panda
UC Berkeley
apanda@cs.berkeley.edu

Sylvia Ratnasamy
UC Berkeley
sylvia@eecs.berkeley.edu

Luigi Rizzo
Università di Pisa
rizzo@iet.unipi.it

Scott Shenker
UC Berkeley and ICSI
shenker@icsi.berkeley.edu

Abstract

By moving network appliance functionality from proprietary hardware to software, Network Function Virtualization promises to bring the advantages of cloud computing to network packet processing. However, the evolution of cloud computing (particularly for data analytics) has greatly benefited from application-independent methods for scaling and placement that achieve high efficiency while relieving programmers of these burdens. NFV has no such general management solutions. In this paper, we present a scalable and application-agnostic scheduling framework for packet processing, and compare its performance to current approaches.

*Joint first authors

SOSP'15, October 04-07, 2015, Monterey, CA.

ACM 978-1-4503-3834-9/15/10.
http://dx.doi.org/10.1145/2815400.2815423

1. Introduction

The proliferation of network processing appliances ("middleboxes") has been accompanied by a growing recognition of the problems they bring, including expensive hardware and complex management. This recognition led the networking industry to launch a concerted effort towards Network Function Virtualization (NFV) with the goal of bringing greater openness and agility to network dataplanes [8]. Inspired by the benefits of cloud computing, NFV advocates moving *Network Functions* (NFs) out of dedicated physical boxes into virtualized software applications that can be run on commodity, general purpose processors. NFV has quickly gained significant momentum with over 220 industry participants, multiple proof-of-concept prototypes, and a number of emerging product offerings [2, 9].

While this momentum is encouraging, a closer look "under the hood" reveals a less rosy picture: NFV products and prototypes tend to be merely virtualized software implementations of products that were previously offered as dedicated hardware appliances. Thus, NFV is currently replacing, on a one-to-one basis, monolithic hardware with monolithic software. While this is a valuable first step – as it is expected to lower capital costs and deployment barriers – it fails to provide a coherent management solution for middleboxes. Each software middlebox still comes as a closed implementation bundled with a custom management solution that addresses issues such as overload detection, load balancing, elastic scaling, and fault-tolerance for that particular NF.

This leads to two problems. First, the operator must cope with many NF-specific management systems. Second, NF developers must invent their own solutions to common but non-trivial problems such as dynamic scaling and fault tolerance; in the worst case this results in inadequate solutions (e.g., solutions that do not scale well) and in the best case results in vendors constantly reinventing the wheel.

Inspired by the success of data analytic frameworks (e.g., MapReduce, Hadoop and Spark), we argue that NFV needs a *framework*, by which we mean a software environment for packet-processing applications that implements *general* techniques for *common* issues. Such issues include: placement (which NF runs where), elastic scaling (adapting the number of NF instances and balancing load across them), service composition, resource isolation, fault-tolerance, energy management, monitoring, and so forth. Although we are focusing on packet-processing applications, the above are all *systems* issues, with some aiding NF development (*e.g.*, fault-tolerance), some NF management (*e.g.*, dynamic scaling) and others orchestration across NFs (*e.g.*, placement, service interconnection).

In this paper, we report on our efforts to build such a framework, which we call Elastic Edge (E2). From a practical perspective, E2 brings two benefits: (i) it allows developers to rely on external framework-based mechanisms for common tasks, freeing them to focus on their core application logic and (ii) it simplifies the operator's responsibilities, as it both automates and consolidates common management tasks. To our knowledge, no such framework for NFV exists today, although several efforts explore individual aspects of the problem (as we discuss in §9).

From a conceptual perspective, our contributions are also twofold. First, we describe algorithms to automate the common tasks of placement, service interconnection, and dynamic scaling. In other work, we also address the issue of fault-tolerance [46], with other issues such as performance isolation, energy management and monitoring left for future work. Second, we present a system architecture that simplifies building, deploying and managing NFs. Our architecture departs from the prevailing wisdom in that it blurs the traditional distinction between applications and the network. Typically one thinks of ap-

plications as having fully general programming abstractions while the network has very limited abstractions (essentially that of a switch); this constrains how functionality is partitioned between application and network (even when network processing is implemented at end-hosts [31, 39]) and encourages separate management mechanisms for each. In contrast, because we focus on more limited packet-processing applications and fully embrace software switches, we can push richer programming abstractions into the network layer.

More concretely, because of the above reasoning, we eschew the dominant software switch, OVS, in favor of a more modular design inspired by Click [30]. We also depart from the traditional SDN/NFV separation of concerns that uses SDN to route packets between NFs and separately lets NFV manage those NFs [17, 21, 40]; instead, in E2, a single controller handles both the management and interconnection of NFs based on a global system view that spans application and network resources (e.g., core occupancy and number of switch rules available). We show that E2's flexibility together with its coordinated approach to management enables significant performance optimizations; e.g., offering a 25-41% reduction in CPU use through flexible system abstractions (§7.1) and a 1.5-4.5x improvement in overall system throughput through better management decisions (§7.2).

2. Context and Assumptions

We now provide a motivating context for the deployment of a framework such as E2, describe the form of hardware infrastructure we assume, and briefly sketch the E2 design.

2.1 Motivation: A Scale-Out Central Office

We present a concrete deployment context that carriers cite as an attractive target for NFV: a carrier network's broadband and cellular edge, as embodied in their *Central Offices (COs)* [1]. A CO is a facility commonly located in a metropolitan area to which residential and business lines connect. Carriers hope to use NFV to transform their COs to more closely resemble modern datacenters so they can achieve: a uniform architecture based on commodity hardware, efficiency through statistical multiplexing, centralized management across CO locations, and the flexibility and portability of software services. Carriers cite two reasons for overhauling CO designs [1].

First, the capital and operational expenses incurred by a carrier's COs are very high. This is because there are many COs, each of non-trivial scale; e.g., AT&T reports 5,000 CO locations in the US alone, with 10-100K subscribers per CO. These COs contain specialized devices such as *Broadband Network Gateways (BNGs)* [3, 4] that connect broadband users to the carrier's IP backbone, and *Evolved Packet Core (EPC)* gateways that connect cellular users to the IP backbone. These are standalone devices with proprietary internals and vendor-specific management APIs.[1] NFV-based COs would enable operators to utilize commodity hardware while a framework such as E2 would provide a unified management system.

Secondly, carriers are seeking new business models based on opening up their infrastructure to 3rd party services. Hosting services in their COs would enable carriers to exploit their physical proximity to users, but this is difficult when new features require custom hardware; an NFV-based CO design would address this difficulty. In fact, if carriers succeed in opening up their infrastructure, then one might view the network as simply an

[1] Standardization efforts such as OpenFlow target L2 and L3 forwarding devices and do not address the complexity of managing these specialized systems or middleboxes more generally [44, 47].

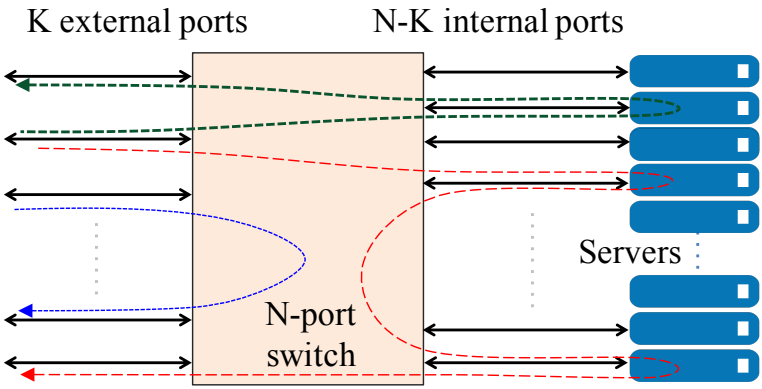


Figure 1: Hardware infrastructure that E2 manages. We show three examples of possible forwarding paths through the cluster, including one that involves no server.

extension (closer to the user) of existing cloud infrastructure in which case the transition to NFV becomes necessary for portability between cloud and network infrastructures.

Carrier incentives aside, we note that a CO's *workload* is ideally suited to NFV's software-centric approach. A perusal of broadband standards [7] and BNG datasheets [4] reveals that COs currently support a range of higher-level traffic processing functions – e.g., content caching, Deep Packet Inspection (DPI), parental controls, WAN and application acceleration, traffic scrubbing for DDoS prevention and encryption – in addition to traditional functions for firewalls, IPTV multicast, DHCP, VPN, Hierarchical QoS, and NAT. As CO workloads grow in complexity and diversity, so do the benefits of transitioning to general-purpose infrastructure, and the need for a unified and application-independent approach to dealing with common management tasks.

Thus, E2 addresses the question of how you efficiently manage a diverse set of packet processing applications without knowing much about their internal implementation. "Efficient" here means both that the management system introduces little additional overhead, and that it enables high utilization of system resources.

2.2 Hardware Infrastructure

E2 is designed for a hardware infrastructure composed of general-purpose servers (residing in racks) interconnected by commodity switches. As shown in Figure 1, we assume a fabric of commodity switches with N ports, of which K are dedicated to be 'externally' facing (i.e., carrying traffic to/from the E2 cluster) while the remaining N-K interconnect the servers running NFV services. This switch fabric can be a single switch, or multiple switches interconnected with standard non-blocking topologies. Our prototype uses a single switch but we expect our design to scale to larger fabrics.

E2 is responsible for managing system resources and hence we briefly elaborate on the main hardware constraints it must accommodate. First, E2 must avoid over-booking the CPU and NIC resources at the servers. Second, E2 must avoid overloading the switch capacity by unnecessarily placing functions on different servers; e.g., a flow processed by functions running at two servers will consume 50% more switching capacity than if the two functions were placed on the same server (Figure 1). Third, since commodity switches offer relatively small flow tables that can be slow to update, E2 must avoid excessive use of the flow table at the switch (see §5.3).

Our current prototype has only a single rack. We presume, based on current packet processing rates and CO traffic volumes, that a CO can be serviced by relatively small cluster sizes (1-10 racks); while we believe that our architecture will easily scale to such numbers, we leave an experimental demonstration of this to future work.

2.3 Design Overview

Before presenting E2 in detail in the following sections, we first provide a brief overview.

E2 Context. We assume that COs reside within an overall network architecture in which a global SDN controller is given (by the operator) a set of network-wide policies to implement. The SDN controller is responsible for translating these network-wide policies into instructions for each CO, and the E2 cluster within each CO is responsible for carrying out these instructions.The E2 cluster is managed by an E2 Manager, which is responsible for communicating with the global SDN controller.

E2 Interface. Akin to several recent network management systems [12, 15–17, 20, 37, 49], E2 provides a declarative interface through which the global SDN controller tells each E2 cluster how traffic should be processed. It does so by specifying a set of policy statements that we call *pipelets*. Each pipelet defines a *traffic class* and a corresponding directed acyclic graph (DAG) that captures how this traffic class should be processed by NFs. A traffic class here refers to a subset of the input traffic; the DAG is composed of nodes which represent NFs (or external ports of the switch) and edges which describe the type of traffic (e.g., 'port 80') that should reach the downstream NF. Figure 2 shows a simplified example of a pipelet.

Thus, the global SDN controller hands the E2 Manager a set of pipelets. The E2 Manager is responsible for executing these pipelets on the E2 cluster as described below, while communicating status information – *e.g.*, overall load or hardware failure – back to the global controller.

In addition to policy, E2 takes two forms of external input: (i) a *NF description* enumerating any NF-specific constraints (*e.g.*, whether the NF can be replicated across servers), configuration directives (*e.g.*, number and type of ports), resource requirements (*e.g.*, per-core throughput), and (ii) a *hardware description* that enumerates switch and server capabilities (*e.g.* number of cores, flow table size).

E2 Internal Operation. Pipelets dictate *what* traffic should be processed by *which* NFs, but not *where* or *how* this processing occurs on the physical cluster. E2 must implement the policy directives expressed by the pipelets while respecting NF and hardware constraints and capabilities, and it does so with three components, activated in response to configuration requests or overload indications. (i) The *scaling* component (§5.3) computes the number of NF *instances* needed to handle the estimated traffic demand, and then dynamically adapts this number in response to varying traffic load. It generates an instance graph, or *iGraph*, reflecting the actual number of instances required for each NF mentioned in the set of pipelets, and how traffic is spread across these instances. (ii) The *placement* component (§5.1) translates the iGraph into an assignment of NF instances to specific servers. (iii) The *interconnection* component (§5.2) configures the network (including network components at the servers) to steer traffic across appropriate NF instances.

In the following sections we describe E2's system architecture (§3), its dataplane design (§4), and its control plane design (§5). We present the implementation (§6) and evaluation (§7) of our E2 prototype then discuss related work (§8) before concluding in §9.

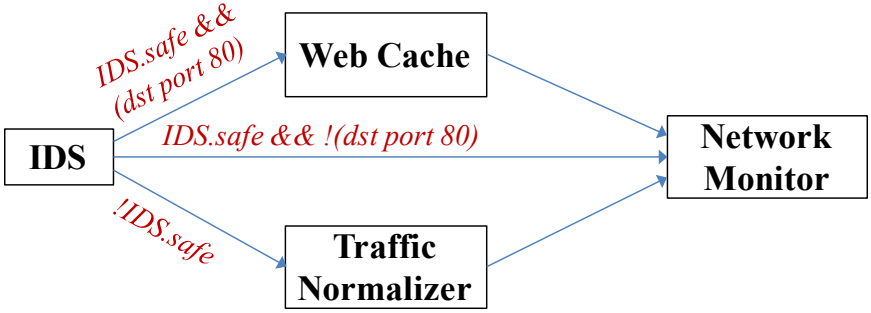


Figure 2: An example pipelet. Input traffic is first sent to an IDS; traffic deemed safe by the IDS is passed to a Web Cache if it's destined for TCP port 80 and to a Network Monitor otherwise. Traffic that the IDS finds unsafe is passed to a Traffic Normalizer; all traffic leaving the Traffic Normalizer or the Web Cache are also passed to the Network Monitor.

3. E2 System Architecture

We now describe E2's API, inputs, and system components.

3.1 System API

As mentioned in §2, an operator expresses her policies via a collection of pipelets, each describing how a particular *traffic class* should be processed. This formulation is declarative, so operators can generate pipelets without detailed knowledge of per-site infrastructure or NF implementations. The necessary details will instead be captured in the NF and hardware descriptions. We now elaborate on how we express pipelets. Additional detail on the policy description language we use to express pipelets can be found in the Appendix.

Each pipelet defines a *traffic class* and a corresponding directed acyclic graph (DAG) that captures how this traffic class should be processed by NFs. In our current implementation, we define traffic classes in terms of packet header fields and physical ports on the switch; for example, one might identify traffic from a particular subscriber via the physical port, or traffic destined for another provider through address prefixes.

A node in the pipelet's DAG represents a NF or a physical port on the switch, and edges describe the traffic between nodes. Edges may be annotated with one or more traffic *filters*. A filter is a boolean expression that defines what subset of the traffic from the source node should reach the destination node.

Filters can refer to both, the contents of the packet itself (e.g., header fields) and to semantic information associated with the packet. For example, the characterization of traffic as "safe" or "unsafe" in Figure 2 represents semantic information inferred by the upstream IDS NF. Filters can thus be viewed as composed of general attribute-value pairs, where attributes can be *direct* (defined on a packet's contents) or *derived* (capturing higher-level semantics exposed by network applications). A packet follows an edge only if it matches all of the traffic filters on the edge. Note that a traffic filter only defines which traffic flows between functions; E2's interconnection component (§5.2) addresses *how* this traffic is identified and forwarded across NF ports.

In addition to traffic filters, an edge is optionally annotated with an estimate of the expected rate of such traffic. E2's placement function uses this rate estimate to derive its

initial allocation of resources; this estimate can be approximate or even absent because E2's dynamic scaling techniques will dynamically adapt resource allocations to varying load.

3.2 System Inputs

In addition to pipelets, E2 takes an *NF description* that guides the framework in configuring each NF, and a *hardware description* that tells the framework what hardware resources are available for use. We describe each in turn.

NF descriptions. E2 uses the following pieces of information for each NF. We envisage that this information (except the last one) will be provided by NF developers.

(1) Native vs. Legacy. E2 exports an optional API that allow NFs to leverage performance optimizations (§4). NFs that use this API are considered "native", in contrast to unmodified "legacy" NFs running on the raw socket interface provided by the OS; we discuss the native API further in §7.

(2) Attribute-Method bindings. Each derived attribute has an associated *method* for associating packets with their attribute values. Our E2 prototype supports two forms of methods: ports and per-packet metadata (§4).

With the port method, all traffic with an attribute value will be seen through a particular (virtual or physical) port. Since a port is associated with a specific value for an attribute, ports are well-suited for "coarse-grained" attributes that take on a small number of well-known values. E.g., in Figure 2, if the IDS defines the method associated with the "safe" attribute to be "port," all safe traffic exits the IDS through one virtual port, and all unsafe traffic through another. Legacy applications that cannot leverage the metadata method described below fit nicely into this model.

The metadata method is available as a native API. Conceptually, one can think of metadata as a per-packet annotation [30] or tag [17] that stores the attribute-value pair; §4 describes how our system implements metadata using a custom header. Metadata is well-suited for attributes that take many possible values; e.g., tagging packets with the URL associated with a flow (versus using a port per unique URL).

(3) Scaling constraints tell E2 whether the application can be scaled across servers/cores or not, thus allowing the framework to react appropriately on overload (§5.3).

(4) Affinity constraints. For NFs that scale across servers, the affinity constraints tell the framework how to split traffic across NF instances. Many NFs perform stateful operations on individual flows and flow aggregates. The affinity constraints define the traffic aggregates the NF acts on (*e.g.*, "all packets with a particular TCP port," or "all packets in a flow"), and the framework ensures that packets belonging to the same aggregate are consistently delivered to the same NF instance. Our prototype accepts affinity constraints defined in terms of the 5-tuple with wildcards.

(5) NF performance. This is an estimate of the per-core, per-GHz traffic rate that the NF can sustain[2]. This is optional information that E2's placement function uses to derive a closer-to-target initial allocation of cores per NF.

Hardware description. In our current prototype, the hardware constraints that E2 considers when making operational decisions include: (1) the number of cores (and speed) and

[2]Since the performance of NFs vary based on server hardware and traffic characteristics, we expect these estimates will be provided by the network operator (based on profiling the NF in their environment) rather than by the NF vendor.

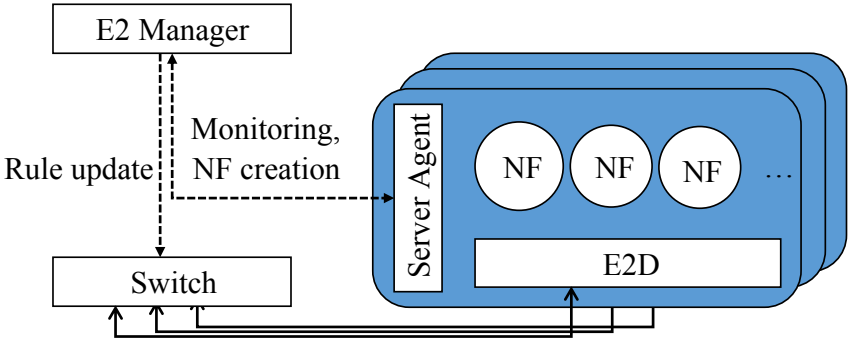


Figure 3: The overall E2 system architecture.

the network I/O bandwidth per server, (2) the number of switch ports, (3) the number of entries in the switch flow table, and (4) the set of available switch actions. Our hardware description thus includes this information. We leave to future work the question of whether and how to exploit richer models – e.g., that consider resources such as the memory or CPU cache at servers, availability of GPUs or specialized accelerators [24], programmable switches [15], and so forth.

3.3 System Components

Figure 3 shows the three main system components in E2: the *E2 Manager* orchestrates overall operation of the cluster, a *Server Agent* manages operation within each server, and the *E2 Dataplane* (E2D) acts as a software traffic processing layer that underlies the NFs at each server. The E2 Manager interfaces with the hardware switch(es) through standard switch APIs [6, 14, 36] and with the Server Agents.

4. The E2 Dataplane, E2D

In the following subsections we describe the design of the E2 Dataplane (E2D). The goal of E2D is to provide flexible yet efficient "plumbing" across the NF instances in the pGraph.

4.1 Rationale

Our E2D implementation is based on SoftNIC [23], a high-performance, programmable software switch that allows arbitrary packet processing *modules* to be dynamically configured as a data flow graph, in a similar manner to the Click modular router [30].

While the Click-style approach is widely used in various academic and commercial contexts, the de-facto approach to traffic management on servers uses the Open vSwitch (OVS) and the OpenFlow interface it exports. OVS is built on the abstraction of a conventional hardware switch: it is internally organized as a pipeline of tables that store 'match-action' rules with matches defined on packet header fields plus some limited support for counters and internal state. Given the widespread adoption of OVS, it is reasonable to ask why we adopt a different approach. In a nutshell, it is because NFV does not share many of the design considerations that (at least historically) have driven the architecture of OVS/Openflow and hence the latter may be unnecessarily restrictive or even at odds with our needs.

More specifically, OVS evolved to support "network virtualization platforms" (NVPs) in multi-tenant datacenters [31]. Datacenter operators use NVPs to create multiple virtual networks, each with independent topologies and addressing architectures, over the same physical network; this enables (for example) tenants to 'cut-paste' a network configuration

from their local enterprise to a cloud environment. The primary operation that NVPs require on the dataplane is the emulation of a packet's traversal through a series of switches in the virtual topology, and thus OVS has focused on fast lookups on OpenFlow tables; *e.g.*, using multiple layers of caching internally [39] and limited actions.

NFV does not face this challenge. Instead, since most cycles will likely be consumed in NFs, we are more interested in performance optimizations that improve the efficiency of NFs (e.g., our native APIs below). Thus, rather than work to adapt OVS to NFV contexts, we chose to explore a Click-inspired dataflow design more suited to our needs. This choice allowed us to easily implement various performance optimizations (§7) and functions in support of dynamic scaling (§5.3) and service interconnection (§5.2).

4.2 SoftNIC

SoftNIC exposes virtual NIC ports (vports) to NF instances; vports virtualize the hardware NIC ports (pports) for virtualized NFs. Between vports and pports, SoftNIC allows arbitrary packet processing *modules* to be configured as a data flow graph, in a manner similar to the Click modular router [30]. This modularity and extensibility differentiate SoftNIC from OVS, where expressiveness and functionality are limited by the flow-table semantics and predefined actions of OpenFlow.

SoftNIC achieves high performance by building on recent techniques for efficient software packet processing. Specifically: SoftNIC uses Intel DPDK [27] for low-overhead I/O to hardware NICs and uses pervasive batch processing within the pipeline to amortize per-packet processing costs. In addition, SoftNIC runs on a small number of dedicated processor cores for high throughput (by better utilizing the CPU cache) and sub-microsecond latency/jitter (by eliminating context switching cost). The SoftNIC core(s) continuously polls each physical and virtual port for packets. Packets are processed from one NF to another using a push-to-completion model; once a packet is read from a port, it is run through a series of modules (*e.g.* classification, rate limiting, etc.) until it reaches a destination port.

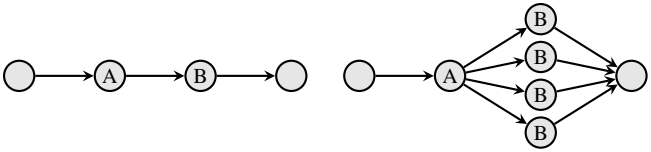
In our experiments with the E2 prototype (§7), we dedicate only one core to E2D/SoftNIC as we find a single core was sufficient to handle the network capacity of our testbed; [23] demonstrates SoftNIC's scalability to 40 Gbps per core.

4.3 Extending SoftNIC for E2D

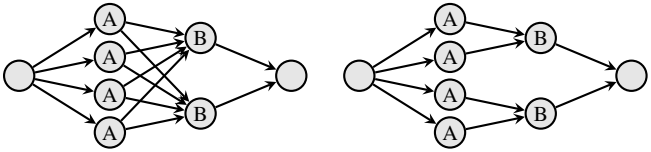
We extend SoftNIC in the following three ways. First, we implement a number of modules tailored for E2D including modules for load monitoring, flow tracking, load balancing, packet classification, and tunneling across NFs. These modules are utilized to implement E2's components for NF placement, interconnection, and dynamic scaling, as will be discussed in the rest of this paper.

Second, as mentioned earlier, E2D provides a native API that NFs can leverage to achieve better system-wide performance and modularity. This native API provides support for: *zero-copy* packet transfer over vports for high throughput communication between E2D and NFs, and rich message abstractions which allow NFs to go beyond traditional packet-based communication. Examples of rich messages include: (i) reconstructed TCP bytestreams (to avoid the redundant overhead at each NF), (ii) per-packet metadata tags that accompany the packet even across NF boundaries, and (iii) inter-NF signals (*e.g.*, a notification to block traffic from an IPS to a firewall).

The richer cross-NF communication enables not only various performance optimizations but also better NF design by allowing modular functions – rather than full-blown



(a) Original pGraph (b) iGraph with split NF B



(c) iGraph with split NF A and B (d) Optimized iGraph

Figure 4: Transformations of a pGraph (a) into an iGraph (b, c, d).

NFs– from different vendors to be combined and reused in a flexible yet efficient manner. We discuss and evaluate the native API further in §7.

Lastly, E2D extends SoftNIC with a control API exposed to E2's Server Agent, allowing it to: (i) dynamically create/destroy vports for NF instances, (ii) add/remove modules in E2D's packet processing pipeline, stitching NFs together both within and across servers, and (iii) receive notifications of NF overload or failure from the E2D (potentially triggering scaling or recovery mechanisms).

5. The E2 Control Plane

The E2 control plane is in charge of (i) placement (instantiating the pipelets on servers), (ii) interconnection (setting up and configuring the interconnections between NFs), (iii) scaling (dynamically adapting the placement decisions depending on load variations), and (iv) ensuring affinity constraints of NFs.

5.1 NF Placement

The initial placement of NFs involves five steps:

Step 1: Merging pipelets into a single policy graph. E2 first combines the set of input pipelets into a single policy graph, or *pGraph*; the pGraph is simply the union of the individual pipelets with one node for each NF and edges copied from the individual pipelets.

Step 2: Sizing. Next, E2 uses the initial estimate of the load on a NF (sum of all incoming traffic streams), and its per-core capacity from the NF description, to determine how many instances (running on separate cores) should be allocated to it. The load and capacity estimates need not be accurate; our goal is merely to find a reasonable starting point for system bootstrapping. Dynamically adapting to actual load is discussed later in this section.

Step 3: Converting the pGraph to an iGraph. This step transforms the pGraph into the "instance" graph, or *iGraph*, in which each node represents an instance of a NF. Splitting a node involves rewiring its input and output edges and Figure 4 shows some possible cases. In the general case, as shown in Figure 4(b) and 4(c), splitting a node requires distributing the input traffic across all its instances in a manner that respects all affinity constraints and generating the corresponding edge filters. As an example, NF B in Figure 4(b) might

require traffic with the same 5-tuple go to the same instance, hence E2 inserts a filter that hashes traffic from A on the 5-tuple and splits it evenly towards B's instances.

When splitting multiple adjacent nodes, the affinity constraints may permit optimizations in the distribute stages, as depicted in Figure 4(d). In this case, node B from the previous example is preceded by node A that groups traffic by source IP addresses. If the affinity constraint for A already satisfies the affinity constraint for B, E2 does not need to reclassify the outputs from A's instances, and instead can create direct connections as in Figure 4(d). By minimizing the number of edges between NF instances, instance placement becomes more efficient, as we explain below.

Step 4: Instance placement. The next step is to map each NF instance to a particular server. The goal is to minimize inter-server traffic for two reasons: (i) software forwarding within a single server incurs lower delay and consumes fewer processor cycles than going through the NICs [19, 43] and (ii) the link bandwidth between servers and the switch is a limited resource. Hence, we treat instance placement as an optimization problem to minimize the amount of traffic traversing the switch. This can be modeled as a graph partition problem which is NP-hard and hence we resort to an iterative local searching algorithm, in a modified form of the classic Kernighan-Lin heuristic [28].

The algorithm works as follows: we begin with a valid solution that is obtained by bin-packing vertices into partitions (servers) based on a depth-first search on the iGraph; then in each iteration, we swap a pair of vertices from two different partitions. The pair selected for a swap is the one that leads to the greatest reduction in cross-partition traffic. These iterations continue until no further improvement can be made. This provides an initial placement of NF instances in $O(n^2 \lg n)$ time where n is the number of NF instances.

In addition, we must consider incremental placement as NF instances are added to the iGraph. While the above algorithm is already incremental in nature, our strategy of migration avoidance (§5.4) imposes that we do not swap an existing NF instance with a new one. Hence, the incremental placement is much simpler: we consider all possible partitions where the new instance may be placed, and choose the one that will incur the least cross-partition traffic by simply enumerating all the neighboring instances of the new NF instance. Thus the complexity of our incremental placement algorithm is $O(n)$, where n is the number of NF instances.

Step 5: Offloading to the hardware switch. Today's commodity switch ASICs implement various low-level features, such as L2/L3-forwarding, VLAN/tunneling, and QoS packet scheduling. This opens the possibility of offloading these functions to hardware when they appear on the policy graph, similar to Dragonet [48] which offloads functions from the end-host network stack to NIC hardware). On the other hand, offloading requires that traffic traverse physical links and consume other hardware resources (table entries, switch ports, queues) that are also limited, so offloading is not always possible. To reduce complexity in the placement decisions, E2 uses an opportunistic approach: a NF is considered as a candidate for offloading to the switch only if, at the end of the placement, that NFs is adjacent to a switch port, and the switch has available resources to run it. E2 does not preclude the use of specialized hardware accelerators to implement NFs, though we have not explored the issue in the current prototype.

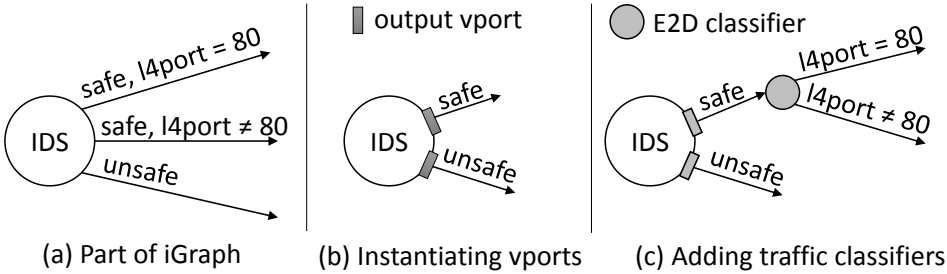


Figure 5: E2 converts edge annotations on an iGraph (a) into output ports (b) that the applications write to, and then adds traffic filters that the E2D implements (c).

5.2 Service Interconnection

Recall that edges in the pGraph (and by extension, iGraph) are annotated with filters. Service interconnection uses these annotations to steer traffic between NF instances in three stages.

Instantiating NFs' ports. The NF description specifies how many output ports are used by a NF and which traffic attributes are associated with each port. E2D instantiates vports accordingly as per the NF description and the iGraph. For example, Fig. 5(b) shows an IDS instance with two vports, which output "safe" and "unsafe" traffic respectively.

Adding traffic filters. An edge may require (as specified by the edge's filters) only a subset of the traffic generated by the NF instance it is attached to. In this case, E2 will insert an additional classification stage, implemented by the E2D, to ensure that the edge only receives traffic matching the edge filters. Figure 5(c) illustrates an example where "safe" traffic is further classified based on the destination port number. While E2's classifier currently implements BPF filtering [35] on packet header fields and metadata tags, we note that it can be extended beyond traditional filtering to (for example) filter packets based on CPU utilization or the active/standby status of NF instances. To disambiguate traffic leaving 'mangling' NFs that rewrite key header fields (*e.g.*, NAT), the E2D layer dynamically creates disambiguating packet steering rules based on the remaining header fields. [3]

Configuring the switch and the E2D. After these steps, E2 must configure the switch and E2D to attach NF ports to edges and instantiate the necessary filters. Edges that are local to one server are implemented by the E2D alone. Edges between servers also flow through the E2D which routes them to physical NICs, possibly using tunneling to multiplex several edges into available NICs. Packet encapsulation for tunneling does not cause MTU issues, as commodity NICs and switches already support jumbo frames.

5.3 Dynamic Scaling

The initial placement decisions are based on estimates of traffic and per-core performance, both of which are imperfect and may change over time. Hence, we need solutions for dynamically scaling in the face of changing loads; in particular we must find ways to split

[3] Our approach to handling mangling NFs is enabled by the ability to inject code inline in the E2D layer. This allows us to avoid the complexity and inefficiency of solutions based on legacy virtual switches such as OVS; these prior solutions involve creating multiple instances of the mangling NF, one for each downstream path in the policy graph [20] and invoke the central controller for each new flow arrival [17, 20].

the load over several NF instances when a single instance is no longer sufficient. We do not present the methods for contraction when underloaded, but these are similar in spirit. We provide hooks for NFs to report on their instantaneous load, and the E2D itself detects overloads based on queues and processing delays.

We say we *split* an instance when we redistribute its load to two or more instances (one of which is the previous instance) in response to an overload. This involves placing the new instances, setting up new interconnection state (as described previously in this section), and must consider the affinity requirements of flows (discussed later in this section), so it is not to be done lightly.

To implement splitting, when a node signals overload the Server Agent notifies the E2 Manager, which uses the incremental algorithm described in §5.1 to place the NF instances. The remaining step is to correctly split incoming traffic across the new and old instances; we address this next.

5.4 Migration Avoidance for Flow Affinity

Most middleboxes are stateful and require *affinity*, where traffic for a given flow must reach the instance that holds that flow's state. In such cases, splitting a NF's instance (and correspondingly, input traffic) requires extra measures to preserve affinity.

Prior solutions that maintain affinity either depend on state migration techniques (moving the relevant state from one instance to another), which is both expensive and incompatible with legacy applications [21], or require large rule sets in hardware switches [41]; we discuss these solutions later in §7.

We instead develop a novel *migration avoidance* strategy in which the hardware and software switch act in concert to maintain affinity. Our scheme does not require state migration, is designed to minimize the number of flow table entries used on the hardware switch to pass traffic to NF instances, and is based on the following assumptions:

- each flow f can be mapped (for instance, through a hash function applied to relevant header fields) to a flow ID $H(f)$, defined as an integer in the interval $R = [0, 2^N)$;
- the hardware switch can compute the flow ID, and can match arbitrary ranges in R with a modest number of rules. Even TCAM-based switches, without a native *range filter*, require fewer than $2N$ rules for this;
- each NF instance is associated with one subrange of the interval R;
- the E2D on each server can track individual, active flows that each NF is currently handling. [4] We call $F_{old}(A)$ the current set of flows handled by some NF A.

When an iGraph is initially mapped onto E2, each NF instance A may have a corresponding range filter $[X, Y) \rightarrow A$ installed in the E2D layer or in the hardware switch. When splitting A into A and A', we must partition the range $[X, Y)$, but keep sending flows in $F_{old}(A)$ to A until they naturally terminate.

Strawman approach. This can be achieved by replacing the filter $[X, Y) \rightarrow A$ with two filters

$$[X, M) \rightarrow A,\ [M, Y) \rightarrow A'$$

and higher priority filters ("exceptions") to preserve affinity:

$$\forall f : f \in F_{old}(A) \wedge H(f) \in [M, Y) : f \rightarrow A$$

[4] The NF description indicates how to aggregate traffic into flows (i.e., the same subset of header fields used to compute the flow ID).

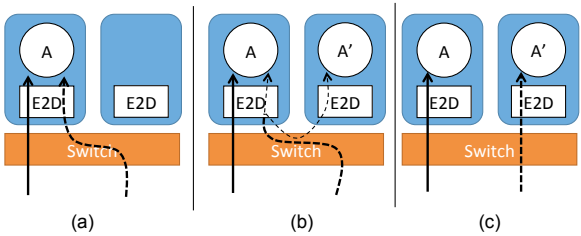


Figure 6: (a) Flows enter a single NF instance. (b) Migration avoidance partitions the range of Flow IDs and punts new flows to a new replica using the E2D. Existing flows are routed to the same instance. (c) Once enough flows expire, E2 installs steering rules in the switch.

The number of exceptions can be very large. If the switch has small filtering tables (hardware switches typically have only a few thousand entries), we can reduce the range $[M, Y)$ to keep the number of exceptions small, but this causes an uneven traffic split. This problem arises when the filters must be installed on a hardware switch, and A and A' reside on different servers.

Our solution To handle this case efficiently, our *migration avoidance* algorithm uses the following strategy (illustrated in Figure 6) :

- Opon splitting, the range filter $[X, Y)$ on the hardware switch is initially unchanged, and the new filters (two new ranges plus exceptions) are installed in the E2D of the server that hosts A;
- As flows in $F_{old}(A)$ gradually terminate, the corresponding exception rules can be removed;
- When the number of exceptions drops below some threshold, the new ranges and remaining exceptions are pushed to the switch, replacing the original rule $[X, Y) \rightarrow A$.

By temporarily leveraging the capabilities of the E2D, migration avoidance achieves load distribution without the complexity of state migration and with efficient use of switch resources. The trade-off is the additional latency to new flows being punted between servers (but this overhead is small and for a short period of time) and some additional switch bandwidth (again, for a short duration) – we quantify these overheads in §7.

6. Prototype Implementation

Our E2 implementation consists of the E2 Manager, the Server Agent, and the E2D. The E2 Manager is implemented in F# and connects to the switch and each server using an out-of-band control network. It interfaces with the switch via an OpenFlow-like API to program the flow table, which is used to load balance traffic and route packets between servers. The E2 Manager runs our placement algorithm (§5) and accordingly allocates a subset of nodes (i.e., NF instances) from the iGraph to each server and instructs the Server Agent to allocate cores for the NFs it has been assigned, to execute the the NFs, to interface with the E2D to allocate ports, create and compose processing modules in SoftNIC, and to set up paths between NFs.

The Server Agent is implemented in Python and runs as a Python daemon on each server in the E2 cluster. The Server Agent acts as a shim layer between the E2 Manager and its local E2D, and it simply executes the instructions passed by the E2 Manager.

The E2D is built on SoftNIC (§4). Our E2D contains several SoftNIC modules which the Server Agent configures for service interconnection and load balancing. Specifically, we

have implemented a match/action module for packet metadata, a module for tagging and untagging packets with tunneling headers to route between servers, and a steering module which implements E2D's part in migration avoidance. The E2D implements the native API discussed in §4.3; for legacy NFs, E2D creates regular Linux network devices.

7. Evaluation

Prototype. Our E2 prototype uses an Intel FM6000 Seacliff Trail Switch with 48 10 Gbps ports and 2,048 flow table entries. We connect four servers to the switch, each with one 10 Gbps link. One server uses the Intel Xeon E5-2680 v2 CPU with 10 cores in each of 2 sockets and the remaining use the Intel Xeon E5-2650 v2 CPU with 8 cores in each of 2 sockets, for a total of 68 cores running at 2.6 GHz. On each server, we dedicate one core to run the E2D layer. The E2 Manager runs on a standalone server that connects to each server and to the management port of the switch on a separate 1 Gbps control network.

We start with microbenchmarks that evaluate E2's data plane (§7.1), then evaluate E2's control plane techniques (§7.2) and finally evaluate overall system performance with E2 (§7.3).

Experimental Setup. We evaluate our design choices using the above E2 prototype. We connect a traffic generator to external ports on our switch with four 10 G links. We use a server with four 10G NICs and two Intel Xeon E5-2680 v2 CPUs as a traffic generator. We implemented the traffic generator to act as the traffic source and sink. Unless stated otherwise, we present results for a traffic workload of all minimum-sized 60B Ethernet packets.

7.1 E2D: Data Plane Performance

We show that E2D introduces little overhead and that its native APIs enable valuable performance improvements.

E2D Overhead. We evaluate the overhead that E2D introduces with a simple forwarding test, where packets are generated by an NF and 'looped back' by the switch to the same NF. In this setup, packets traverse the E2D layer twice (NF $\rightarrow$ switch and switch $\rightarrow$ NF directions). We record an average latency of 4.91 µs.

We compare this result with a scenario where the NF is directly implemented with DPDK (recall that SoftNIC and hence E2D build on top of DPDK), in order to rule out the overhead of E2D. In this case the average latency was 4.61 µs, indicating that E2D incurs 0.3 µs delay (or 0.15 µs for each direction). Given that a typical end-to-end latency requirement within a CO is 1 ms[5], we believe that this latency overhead is insignificant.

In terms of throughput, forwarding through E2D on a single core fully saturates the server's 10 Gbps link as expected [23, 27].

The low latency and high throughput that E2D achieves is thanks to its use of SoftNIC/DPDK. Our results merely show that the *baseline* overhead that E2D/SoftNIC adds to its underlying DPDK is minimal; more complex packet processing at NFs would, of course, result in proportionally higher delays and lower throughput.

E2D Native API. Recall that E2's native API enables performance optimizations through its support for zero-copy vports and rich messages. We use the latter to implement two optimizations: (i) bytestream vports that allow the cost of TCP session reconstruction to be amortized across NFs and, (ii) packet metadata tags that eliminate redundant work by

[5]From discussion with carriers and NF vendors.

Path NF → E2D→ NF	Latency (μs)	Gbps (1500B)	Mpps (64B)
Legacy API	3.2	7.437	0.929
Native Zero-Copy API	1.214	187.515	15.24

Table 1: Latency and throughput between NFs on a single server using E2's legacy vs. native API. Legacy NFs use the Linux raw socket interface.

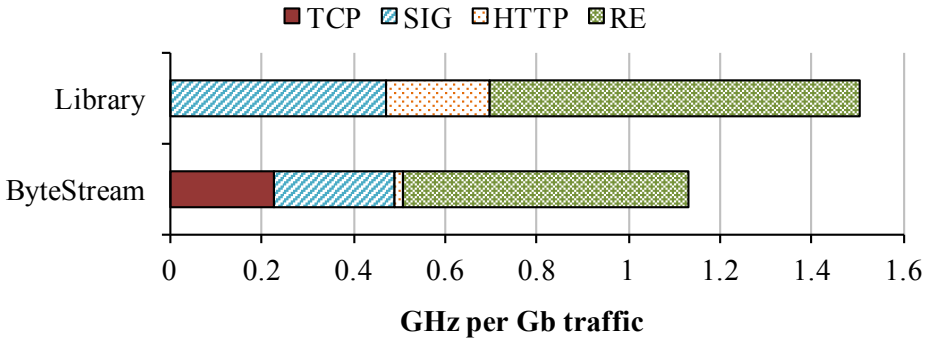


Figure 7: Comparison of CPU cycles for three DPI NFs, without and with bytestream vports. The both cases use the native API.

allowing semantic information computed by one NF to be shared with the E2D or other NFs. We now quantify the performance benefits due to these optimizations: zero-copy vports, bytestream vports, metadata.

Zero-copy vports. We measure the latency and throughput between two NFs on a single server (since the native API does nothing to optimize communication between servers). Table 1 compares the average latency and throughput of the legacy and native APIs along this NF → E2D → NF path. We see that our native API reduces the latency of NF-to-NF communication by over 2.5x on average and increases throughput by over 26x; this improvement is largely due to zero-copy vports (§4) and the fact that legacy NFs incur OS-induced overheads due to packet copies and interrupts. Our native APIs matches the performance of frameworks such as DPDK [27] and netmap [42].

Bytestream vports. TCP session reconstruction, which involves packet parsing, flow state tracking, and TCP segment reassembly, is a common operation required by most DPI-based NFs. Hence, when there are multiple DPI NFs in a pipeline, repeatedly performing TCP reconstruction can waste processing cycles.

We evaluate the performance benefits of bytestream vports using a pipeline of three simple DPI NFs: (i) SIG implements signature matching with the Aho-Corasick algorithm, (ii) HTTP implements an HTTP parser, and (iii) RE implements redundancy elimination using Rabin fingerprinting. These represent an IDS, URL-based filtering, and a WAN optimizer, respectively. The *Library* case in Fig. 7 represents a baseline, where each NF independently performs TCP reconstruction over received packets with our common TCP library. In the ByteStream case, we dedicate a separate NF (TCP) to perform TCP reconstruction and produce metadata (TCP state changes and reassembly anomalies) and reassembled bytestream messages for the three downstream NFs to reuse. E2D guarantees reliable transfer of all messages between NFs that use bytestream vports, with much less overhead than full TCP. The results show that bytestream vports can save 25% of

Path NF → E2D→ NF	Latency (μs)	Gbps (1500B)	Mpps (64B)
Header-Match	1.56	152.515	12.76
Metadata-Match	1.695	145.826	11.96

Table 2: Latency and throughput between NFs on a single server with and without metadata tags.

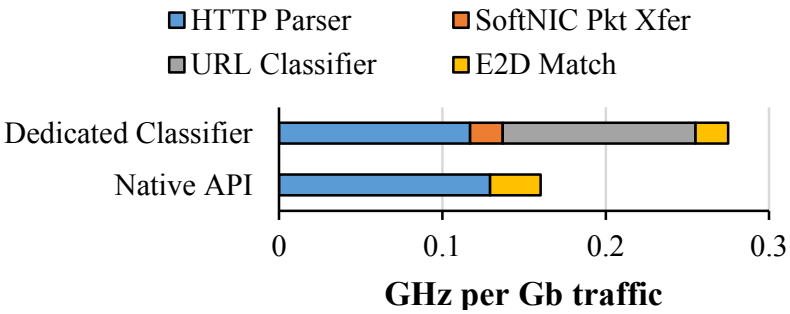


Figure 8: Comparison of CPU cycles between using URL metadata and a dedicated HTTP parser

processing cycles, for the same amount of input traffic.

Metadata Tags. Tags can carry information along with packets and save repeated work in the applications; having the E2D manage tags is both a convenience and potentially also a performance benefit for application writers. The following two experiments quantify the overhead and potential performance benefits due to tagging packets with metadata.

To measure the overhead, we measure the inter-NF throughput using our zero-copy native API under two scenarios. In *Header-Match*, the E2D simply checks a particular header field against a configured value; no metadata tags are attached to packets. In *Metadata-Match*, the source NF creates a metadata tag for each packet which is set to the value of a bit field in the payload; the E2D then checks the tag against a configured value. Table 2 shows our results. We see that *Metadata-Match* achieves a throughput of 11.96 mpps, compared to 12.7 for *Header-Match*. Thus adding metadata lowers throughput by 5.7%.

We demonstrate the performance benefits of metadata tags using a pipeline in which packets leaving an upstream HTTP Logger NF are forwarded to a CDN NF based on the value of the URL associated with their session. Since Logger implements HTTP parsing, a native implementation of the Logger NF can tag packets with their associated URL and the E2D layer will steer packets based on this metadata field. Without native metadata tags, we need to insert a standalone 'URL-Classifier' NF in the pipeline between the Logger and CDN NFs to create equivalent information. In this case, traffic flows as Logger→ E2D → **URL-Classifier** → **E2D**→CDN. As shown in Figure 8, the additional NF and E2D traversal (in bold) increase the processing load by 41% compared to the use of native tags.

7.2 E2 Control Plane Performance

We now evaluate our control plane solutions for NF placement, interconnection, and dynamic scaling, showing that our placement approach achieves better efficiency than two strawmen solutions and that our migration-avoidance design is better than two natural alternatives.

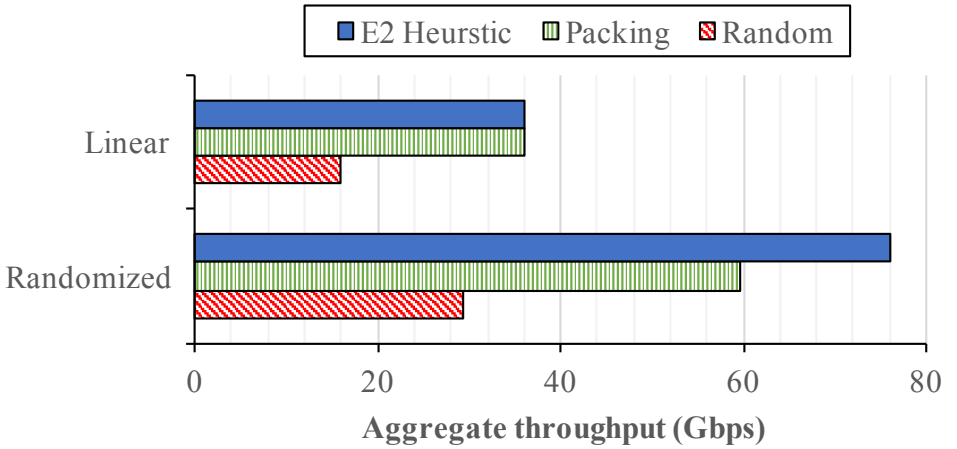


Figure 9: Maximum cluster throughput with different placement solutions, with two different pGraphs.

NF Placement. E2 aims to maximize cluster-wide throughput by placing NFs in a manner that minimizes use of the hardware switch capacity. We evaluate this strategy by simulating the maximum cluster-wide throughput that a rack-scale E2 system (*i.e.*, with 24 servers and 24 external ports) could sustain before *any* component – cores, server links, or switch capacity – of the system is saturated. We compare our solution to two strawmen: "Random" that places nodes on servers at random, and "Packing" that greedily packs nodes onto servers while traversing the iGraph depth-first. We consider two iGraphs: a linear chain with 5 nodes, and a more realistic random graph with 10 nodes.

Figure 9 shows that our approach outperforms the strawmen in all cases. We achieve 2.25-2.59× higher throughput compared to random placement; bin-packing does well on a simple chain but only achieves 0.78× lower throughput for more general graphs. Thus we see that our placement heuristic can improve the overall cluster throughput over the baseline bin-packing algorithm.

Finally, we measure the controller's time to compute placements. Our controller implementation takes 14.6ms to compute an initial placement for a 100-node iGraph and has a response time of 1.76ms when handling 68 split requests per second (which represents the aggressive case of one split request per core per second). We conclude that a centralized controller is unlikely to be a performance bottleneck in the system.

Updating Service Interconnection. We now look at the time the control plane takes to update interconnection paths. In our experiments, the time to update a single rule in the switch varies between 1-8ms with an average of 3ms (the datasheet suggests 1.8ms as the expected time); the switch API only supports one update at a time. In contrast, the per-rule update latency in E2D is only 20 µs, which can be further amortized with a batch of multiple rules. The relatively long time it takes to update the hardware switch (as compared to the software switch) reinforces our conservative use of switch rule entries in migration avoidance. Reconfiguring the E2D after creating a new replica takes roughly 15ms, including the time to coordinate with the E2 Manager and to invoke a new instance.

Dynamic Scaling. We start by evaluating migration avoidance for the simple scenario of a single NF instance that splits in two; the NF requires flow-level affinity. We drive the NF with 1 Gbps of input traffic, with 2,000 new flows arriving each second on average and flow length distributions drawn from published measurement studies [32]. This results in

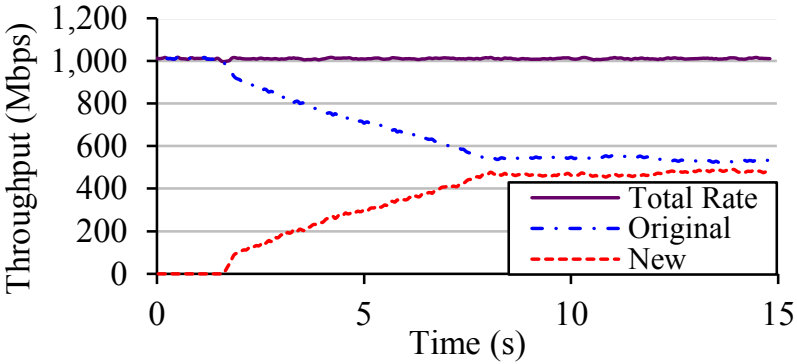


Figure 10: Traffic load at the original and new NF instance with migration avoidance; original NF splits at 2s.

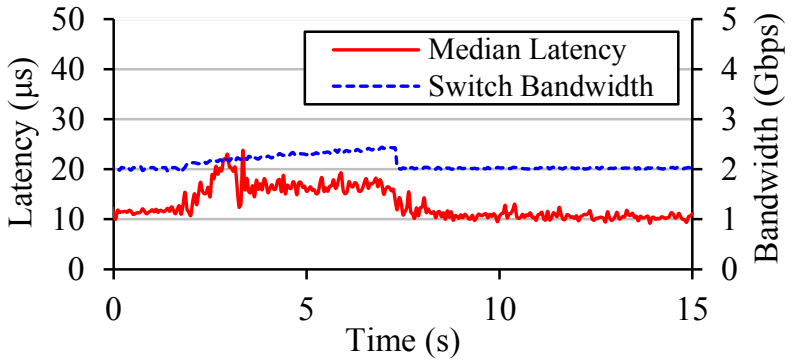


Figure 11: Latency and bandwidth overheads of migration avoidance (the splitting phase is from 1.8s to 7.4s).

a total load of approximately 10,000 active concurrent flows and hence dynamic scaling (effectively) requires 'shifting' load equivalent to 5,000 flows off the original NF.

Fig. 10 shows the traffic load on the original and new NF instances over time; migration avoidance is triggered close to the 2 second mark. We see that our prototype is effective at balancing load: once the system converges, the imbalance in traffic load on the two instances is less than 10%.

We also look at how active flows are impacted *during* the process of splitting. Fig. 11 shows the corresponding packet latency and switch bandwidth consumption over time. We see that packet latency and bandwidth consumption increase during the splitting phase (roughly between the two and eight second markers) as would be expected given we 'detour' traffic through the E2D layer at the original instance. However this degradation is low: in this experiment, latency increases by less than 10μsecs on average, while switch bandwidth increases by 0.5Gbps in the worst case, for a small period of time; the former overhead is a fixed cost, the latter depends on the arrival rate of new flows which is relatively high in our experiment. In summary: migration avoidance balances load evenly (within 10% of ideal) and within a reasonable time frame (shifting load equivalent to roughly 5,000 flows in 5.6 seconds) and does so with minimal impact to active flows (adding less than 10μseconds to packet latencies) and highly scalable use of the switch flow table.

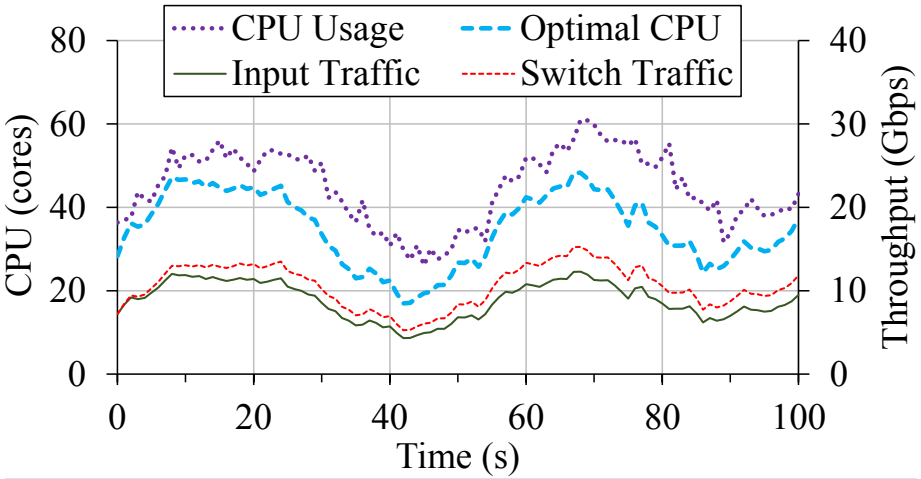


Figure 12: E2 under dynamic workload.

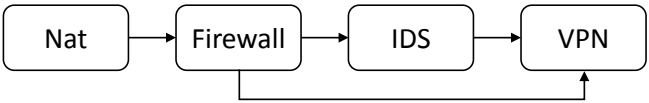


Figure 13: Pipeline used for the evaluation

We briefly compare to two natural strawmen. An "always migrate" approach, as explored in [41] and used in [21], migrates half the active flows to the new NF instance. This approach achieves an ideal balance of load but is complex[6] and requires non-trivial code modifications to support surgical migration of per-flow state. In addition, the disruption due to migration is non-trivial: the authors of [41] report taking 5ms to migrate a single flow during which time traffic must be "paused"; the authors do not report performance when migrating more than a single flow.

A "never migrate" approach that does not leverage software switches avoids migrating flows by pushing exception filters to the hardware switch. This approach is simple and avoids the overhead of detouring traffic that we incur. However, this approach scales poorly; e.g., running the above experiment with never-migrate resulted in a 80% imbalance while consuming all 2,048 rules on the switch.[7] Not only was the asymptotic result poor, but convergence was slow because the switch takes over 1ms to add a single rule and we needed to add close to 2,000 rules.

7.3 E2 Whole-System Performance

To test overall system performance for more realistic NF workloads, we derived a policy graph based on carrier guidelines [7] and BNG router datasheets [4] with 4 NFs: a NAT, a firewall, an IDS and a VPN, as shown in Figure 13. All NFs are implemented in C over our zero-copy native API.

[6]For example, [41] reroutes traffic to an SDN controller while migration is in progress while [21] requires a two-phase commit between the controller and switches; the crux of the problem here is the need for close coordination between traffic and state migration.

[7]The "always migrate" prototype in [41] also uses per-flow rules in switches but this does not appear fundamental to their approach.

We use our prototype with the server and switch configuration described earlier. As in prior work on scaling middleboxes [21], we generate traffic to match the flow-size distribution observed in real-world measurements [13].

We begin the experiment with an input load of 7.2 Gbps and the optimal placement of NFs. Over the course of the experiment, we then vary the input load dynamically up to a maximum of 12.3 Gbps and measure the CPU utilization and switch bandwidth used by E2. Figure 12 plots this measured CPU and switch resource consumption under varying input load. As points of comparison, we also plot the input traffic load (a lower bound on the switch bandwidth usage) and a computed value of the optimal number of cores. We derived the optimal number of cores by summing up the optimal number of NF instances for each NF in the pipeline. To derive the optimal number of NF instances for a NF, we multiply the cycles per unit of traffic that the NF consumes when running in isolation by the total input traffic to the NF, then we divide it by the cycle frequency of a CPU core.

We observe that E2's resource consumption (both CPU and switch bandwidth) scales dynamically to track the trend in input load. At its maximum scaling point, the system consumed up to 60 cores, running an iGraph of 56 vertices (*i.e.*, 56 NF instances) and approximately 600 edges. We also observe the gap between actual *vs.* optimal resource usage in terms of both CPU cores (22.7% on average) and the switch bandwidth (16.4% on average). We note that our measured CPU usage *does* include the cores dedicated to running SoftNIC (which was always one per server in our experiments) while these cores are not included in the optimal core count. Thus the overheads that E2 incurs appear reasonable given that our lower bounds ignore a range of system overheads around forwarding packets between NFs, the NIC and the switch, as also the overheads around running multiple NFs on a shared core or CPU (cache effects, etc.), and the efficiencies that result from avoiding migration and incremental placement as traffic load varies. Finally, we note that our NFs do not use our bytestream and metadata APIs which we expect could yield further improvements.

8. Related Work

We elaborate on related efforts beyond the work mentioned inline throughout this paper.

Industry efforts. Within industry, ETSI operates as the standards body for NFV and OPNFV is a nascent open-source effort with many of the same participants. While "orchestration" figures prominently in their white papers, discussions with participants in these efforts revealed that few demonstrated solutions exist as yet. Components of E2 have been approved as an informative standard at ETSI to fill this role [10].

In terms of academic projects, there are various systems that address individual aspects of E2's functionality, such as load balancing [18, 38], interconnection [17], declarative policies [49], migration [21, 41], but do not provide an overall framework and the system optimizations this enables.

End-to-end NF management. Closest to E2 is Stratos [20], an orchestration layer for NFs deployed in clouds. E2 and Stratos share similar goals but differ significantly in their design details. At a high level, we believe these differences stem from E2's decision to deviate from the canonical SDN architecture. In canonical SDN, the virtual switch at servers offers only the limited processing logic associated with the OpenFlow standard. Instead, E2's control solutions exploit the ability to inject 'non standard' processing logic on the data

path and this allows us to devise simpler and more scalable solutions for tasks such as service interconnection, overload detection, and dynamic scaling.

NF interconnection. Our use of metadata tags (§3) takes inspiration from prior work on FlowTags [17] but with a few simplifying differences: (1) we do not require metadata to accommodate 'mangling' NFs, such as NAT (a key focus in [17]); and (2) metadata tags are only declared and configured at the time of system initialization, and at runtime the E2 datapath does not require reactive involvement of the SDN controller on a per-flow basis, as in FlowTags. Once again, we believe these differences follow from our decision to embed rich programmability in the E2D layer. Our metadata tags are also inspired by the concept of packet annotations in Click [30] and 'network service headers' as defined by the IETF's Service Function Chaining [45].

Hardware platform. E2's platform of choice is a combination of commodity servers and merchant switch silicon. The ServerSwitch system [33] and some commercial "service routers" and appliances [5] also combine x86 and switching hardware; however, in these designs, the two are tightly integrated, locking operators into a fixed ratio of switching to compute capacity. E2 is instead based on loose integration: operators can mix-and-match components from different vendors and can reconfigure (even in the field) the amount of switching and compute capacity based on their evolving needs. Greenhalgh *et al.* [22] describe their vision of a "flowstream" platform that combines switch and x86 hardware in a manner similar to E2 but we are unaware of a detailed design or implementation based on the proposed platform, nor do they articulate the need for a framework of the form E2 aims to provide.

Data plane components. Multiple recent efforts provide specialized platforms in support of efficient software packet processing. Frameworks such as DPDK [27] and netmap [42] are well established tools for high performance packet I/O over commodity NICs. Other systems address efficient packet transfer between VMs in a single server [19, 25, 26, 43], still others explore the trade-offs in hosting NFs in processes, containers, or VMs [29, 34]. All these systems address issues that are complementary but orthogonal to E2: these systems do not address the end-to-end NFV orchestration that E2 does (*e.g.*, placement, scaling), but E2 (and the NFs that E2 supports) can leverage ideas developed in these systems for improved performance.

Dynamic scaling with cross-NF state. Our migration avoidance scheme (§5.4) avoids the complexity of state migration for NFs that operate on state that is easily partitioned or replicated across NF instances; e.g., per-flow counters, per-flow state machines and forwarding tables. However, we do not address the consistency issues that arise when global or aggregate state is spread across multiple NF instances. This is a difficult problem that is addressed in the Split-Merge [41] and OpenNF [21] systems. These systems require that NF vendors adopt a new programming model [41] or add a non-trivial amount of code to existing NF implementations [21]. To support NFs that adopt the Split-Merge or OpenNF architecture, we could extend the E2 controller to implement their corresponding control mechanisms.

9. Conclusion

In this paper we have presented E2, a *framework* for NFV packet processing. It provides the operator with a single coherent system for managing NFs, while relieving developers from having to develop per-NF solutions for placement, scaling, fault-tolerance, and other

functionality. We hope that an open-source framework such as E2 will enable potential NF vendors to focus on implementing interesting new NFs while network operators (and the open-source community) can focus on improving the common management framework.

We verified that E2 did not impose undue overheads, and enabled flexible and efficient interconnection of NFs. We also demonstrated that our placement algorithm performed substantially better than random placement and bin-packing, and our approach to splitting NFs with affinity constraints was superior to the competing approaches.

Acknowledgements

We thank our shepherd, Jeff Mogul, and the anonymous reviewers of the SOSP program committee, for their thoughtful feedback which greatly improved this paper. Ethan Jackson, Kay Ousterhout, and Joshua Reich offered feedback on early drafts of this paper and Maziar Manesh helped us define our lab environment. We are indebted to Tom Anschutz (AT&T), Wenjing Chu (Dell), Yunsong Lu (Huawei), Christian Maciocco (Intel), Pranav Mehta (Intel), Andrew Randall (Metaswitch), Attila Takacs (Ericsson), Percy Tarapore (AT&T), Martin Taylor (Metaswitch) and Venky Venkatesan (Intel) for discussions on NFV efforts in industry.

Appendix: E2 Policy Language

In E2, the operator writes a collection of policy statements, each of which is represented as a directed acyclic graph which we call a 'pipelet'. Nodes in a pipelet correspond to Network Functions (NFs) which receive packets on inbound edges and output packets to outbound edges. Edges are associated with filters which we elaborate on shortly.

NFs are instantiated from specific application types, much like objects are instantiated from classes in object-oriented programming. In addition to user-defined NFs, there are predefined ones such as `Port` which denotes a port on the switch, `Drop`, which discards packets, and `Tee` which creates multiple copies of a packet.

Figure 14 shows an example of a policy with two pipelets, each of which represents a subset of the possible paths for forward and reverse traffic respectively. Figure 15 shows the same example in graph form. The first five lines define the nodes in the graph. Following this are two pipelets, each defining a graph for a specific traffic class. Within the pipelet, we list all the edges forming a policy graph for that traffic class. An edge is described using the simple syntax:

```
src [filter_out] -> (bw) [filter_in] dst;
```

where all three annotations—*filter_out*, *bw*, and *filter_in*—are optional. Filters are boolean expressions computed over packet header fields, physical/virtual ports, or metadata tags, and are written in the libpcap-filter syntax [11]. The *filter_out* annotations specify which packets generated from a NF should enter the edge and implicitly define the traffic class; *filter_in* annotations capture requirements on incoming traffic that are imposed by the downstream NF (example below) and *bw* denotes the expected amount of traffic on the edge, at system bootup.

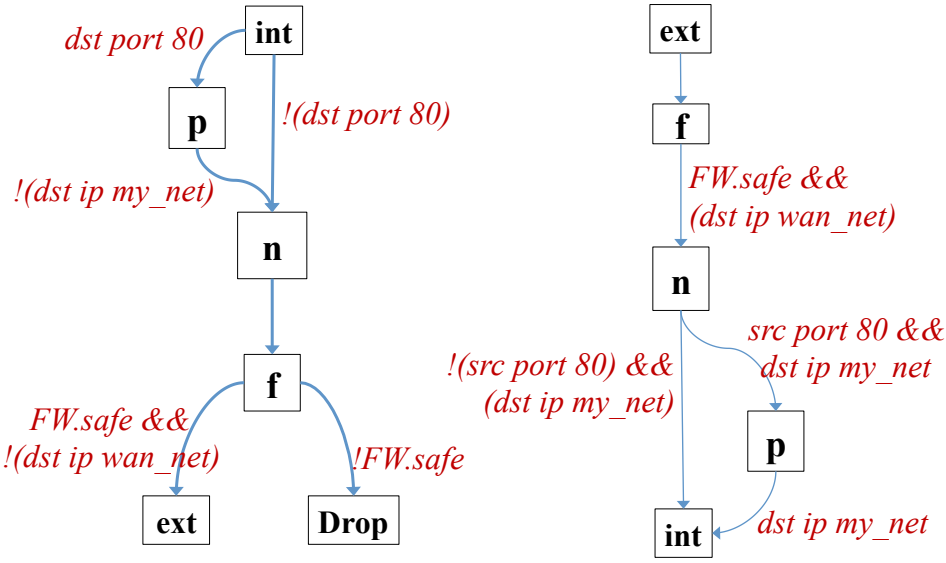
Figure 16 shows a partial example of the use of *filter_in* annotations. Here outbound traffic is passed through a rate-limiter with two input vports; high priority traffic must arrive on one vport and low priority traffic on the other. Thus traffic must be filtered prior to entering the downstream NF; we use *filter_in* annotations to capture such requirements. In this example, prio0 and prio1 are NF-specific metadata that, in this case, are resolved to ports `vp0` and `vp1` at compile time based on information in the rate-limiter's NF description (see §2). In some sense, these are 'placeholder' metadata in that they serve as a level of

```
// First, instantiate NFs from application types.
Proxy p;
NAT   n;
FW    f;
Port<0-7>  int;  // internal customer-facing ports
Port<8-15> ext;  // external WAN-facing ports

// subnet declarations, to simplify filter writing
Address my_net 10.12.30.0/24; // private IP addr
Address wan_net 131.114.88.92/30; // public IP addr

pipelet {  // outbound traffic
  int [dst port 80] -> p;
  int [!(dst port 80)] -> n;
  p [!(dst ip my_net)] -> n;
  n -> f;
  f [FW.safe && !(dst ip wan_net)] -> ext;
  f [!FW.safe] -> Drop;
}

pipelet { // inbound traffic
  ext -> f;
  f [FW.safe && (dst ip wan_net)] -> n;
  n [(src port 80) && (dst ip my_net)] -> p;
  n [!(src port 80) && (dst ip my_net)] -> int;
  p [dst ip my_net] -> int;
}
```

Figure 14: An example specification of a policy. (Certain drop actions are omitted for clarity.)

indirection between policy and mechanism but may not appear at runtime. This allows the operator to be agnostic to how the rate-limiter identifies `prio0` vs. `prio1` traffic. For example, if this is done using metadata (a native rate limiter), E2 will automatically configure the E2D layer to add the appropriate metadata tags; if instead the rate-limiter offers different input ports for each priority class, then the E2D layer will simply steer traffic to the appropriate vport (with no tags, as in our example).

E2 merges all pipelets into a single policy graph, termed a *pGraph*. During this process, E2 checks that each packet coming from a node has exactly one destination. This is verified by checking that the filters on every pair of edges coming out from a NF has an empty intersection, and that the union of all filters attached to a NF's output evaluates to true. If a packet has more than one possible destination, E2 first attempts to remedy this by adding filters to ambiguous edges, specifying the traffic class of the pipelet corresponding to that edge. E.g., simply merging the two pipelets in Figure 15 results in ambiguity for packets leaving the NAT. E2 will thus add a filter that limits the edge for the '`n -> f`' rule to traffic arriving from an internal port. If E2 is unable to automatically resolve ambiguity, it returns an error (this is possible if the operator writes a malformed pipelet); E2 also returns an error if a packet coming from a node has no destination.



(a) pipelet for outbound traffic (b) Pipelet for outbound traffic

Figure 15: Graphical representation of the pipelets in Figure 14.

```
RateLimiter r;   // two input ports, vp0 and vp1
Port<0-7>  int;  // internal customer-facing ports

pipelet {  // outbound traffic
  int [tos 0] -> [prio0] r;
  int [tos 1] -> [prio1] r;
  r -> ...
}
```

Figure 16: An example of a policy that uses filter_in *annotations.*

References

[1] AT&T Domain 2.0 Vision White Paper. https://www.att.com/Common/about_us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf.

[2] Brocade Vyatta 5400 vRouter. http://www.brocade.com/products/all/network-functions-virtualization/product-details/5400-vrouter/index.page.

[3] Ericsson SE Family. http://www.ericsson.com/ourportfolio/products/se-family.

[4] Evolution of the Broadband Network Gateway. http://resources.alcatel-lucent.com/?cid=157553.

[5] Evolved Packet Core Solution. http://lte.alcatel-lucent.com/locale/en_us/downloads/wp_mobile_core_technical_innovation.pdf.

[6] Intel Ethernet Switch FM6000 Series - Software Defined Networking. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[7] Migration to Ethernet-Based Broadband Aggregation. http://www.broadband-forum.org/

technical/download/TR-101_Issue-2.pdf.

[8] Network Functions Virtualisation. http://www.etsi.org/technologies-clusters/technologies/nfv.

[9] NFV Proofs of Concept. http://www.etsi.org/technologies-clusters/technologies/nfv/nfv-poc.

[10] REL002: Scalable Architecture for Reliability (work in progress). http://docbox.etsi.org/ISG/NFV/Open/Drafts/.

[11] *pcap-filter(7) FreeBSD Man Pages*, Jan 2008.

[12] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic Foundations for Networks. In *Proc. ACM POPL* (2014).

[13] BENSON, T., AKELLA, A., AND MALTZ, D. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. Internet Measurement Conference* (2010).

[14] BOSSHART, P., DALY, D., IZZARD, M., MCKEOWN, N., REXFORD, J., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. Programming Protocol-Independent Packet Processors. *CoRR abs/1312.1719* (2013).

[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. In *Proc. ACM SIGCOMM* (2013).

[16] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking Control of the Enterprise. In *Proc. ACM SIGCOMM* (2007).

[17] FAYAZBAKHSH, S., CHIANG, L., SEKAR, V., YU, M., AND MOGUL, J. FlowTags: Enforcing Network-Wide Policies in the Face of Dynamic Middlebox Actions. In *Proc. USENIX NSDI* (2014).

[18] GANDHI, R., LIU, H. H., HU, Y. C., LU, G., PADHYE, J., YUAN, L., AND ZHANG, M. Duet: Cloud Scale Load Balancing with Hardware and Software. In *Proc. ACM SIGCOMM* (2014).

[19] GARZARELLA, S., LETTIERI, G., AND RIZZO, L. Virtual Device Passthrough for High Speed VM Networking. In *Proc. ANCS* (2015).

[20] GEMBER, A., KRISHNAMURTHY, A., JOHN, S. S., GRANDL, R., GAO, X., ANAND, A., BENSON, T., AKELLA, A., AND SEKAR, V. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. *CoRR abs/1305.0209* (2013).

[21] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. OpenNF: Enabling Innovation in Network Function Control. In *Proc. ACM SIGCOMM* (2014).

[22] GREENHALGH, A., HUICI, F., HOERDT, M., PAPADIMITRIOU, P., HANDLEY, M., AND MATHY, L. Flow Processing and the Rise of Commodity Network Hardware. *ACM SIGCOMM Computer Communications Review 39*, 2 (2009), 20–26.

[23] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. SoftNIC: A Software NIC to Augment Hardware. *UCB Technical Report No. UCB/EECS-2015-155* (2015).

[24] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-Accelerated Software Router. In *Proc. ACM SIGCOMM* (2010).

[25] HONDA, M., HUICI, F., LETTIERI, G., AND RIZZO, L. mSwitch: A Highly-Scalable, Modular Software Switch. In *Proc. SOSR* (2015).

[26] HWANG, J., RAMAKRISHNAN, K. K., AND WOOD, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. *IEEE Transactions on Network and Service Management 12*, 1 (2015), 34–47.

[27] Intel Data Plane Development Kit. http://dpdk.org.

[28] KERNIGHAN, B., AND LIN, S. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal 49*, 2 (February 1970).

[29] KIVITY, A., LAOR, D., COSTA, G., ENBERG, P., HAR'EL, N., MARTI, D., AND ZOLOTAROV, V. OSv—Optimizing the Operating System for Virtual Machines. In *Proc. USENIX ATC* (2014).

[30] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Transactions on Computer Systems 18*, 3 (August 2000), 263–297.

[31] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., INGRAM, P., JACKSON, E., LAMBETH, A., LENGLET, R., LI, S.-H., PADMANABHAN, A., PETTIT, J., PFAFF, B., RAMANATHAN, R., SHENKER, S., SHIEH, A., STRIBLING, J., THAKKAR, P., WENDLANDT, D., YIP, A., AND ZHANG, R. Network Virtualization in Multi-tenant Datacenters. In *Proc. USENIX NSDI* (2014).

[32] LEE, D., AND BROWNLEE, N. Passive Measurement of One-way and Two-way Flow Lifetimes. *ACM SIGCOMM Computer Communications Review 37*, 3 (November 2007).

[33] LU, G., GUO, C., LI, Y., ZHOU, Z., YUAN, T., WU, H., XIONG, Y., GAO, R., AND ZHANG, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proc. USENIX NSDI* (2011).

[34] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. ClickOS and the Art of Network Function Virtualization. In *Proc. USENIX NSDI* (2014).

[35] MCCANNE, S., AND JACOBSON, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. USENIX Winter* (1993).

[36] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communications Review 38*, 2 (2008), 69–74.

[37] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software-Defined Networks. In *Proc. USENIX NSDI* (2013).

[38] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud Scale Load Balancing. In *Proc. ACM SIGCOMM* (2013).

[39] PFAFF, B., PETTIT, J., KOPONEN, T., CASADO, M., AND SHENKER, S. Extending Networking into the Virtualization Layer. In *Proc. ACM HotNets* (2009).

[40] QAZI, Z., TU, C., CHIANG, L., MIAO, R., VYAS, S., AND YU, M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. ACM SIGCOMM* (2013).

[41] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. USENIX NSDI* (2013).

[42] RIZZO, L. netmap: A Novel Framework for Fast Packet I/O. In *Proc. USENIX ATC* (2012).

[43] RIZZO, L., AND LETTIERI, G. VALE: A Switched Ethernet for Virtual Machines. In *Proc. ACM CoNEXT* (2012).

[44] SEKAR, V., RATNASAMY, S., REITER, M. K., EGI, N., AND SHI, G. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment. In *Proc. ACM HotNets* (2011).

[45] Network Service Header. `https://tools.ietf.org/html/draft-quinn-nsh-00`.

[46] SHERRY, J., GAO, P., BASU, S., PANDA, A., KRISHNAMURTHY, A., MACCIOCCO, C., MANESH, M., MARTINS, J., RATNASAMY, S., RIZZO, L., AND SHENKER, S. Rollback-Recovery for Middleboxes. In *Proc. ACM SIGCOMM* (2015).

[47] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. ACM SIGCOMM* (2012).

[48] SHINDE, P., KAUFMANN, A., ROSCOE, T., AND KAESTLE, S. We Need to Talk About NICs.

[49] SOULÉ, R., BASU, S., KLEINBERG, R., SIRER, E. G., AND FOSTER, N. Managing the Network with Merlin. In *Proc. ACM HotNets* (2013).