Enforcing Network Policy in Heterogeneous Network Function Box Environment

Lin Cui[a], Fung Po Tso[b,*], Weijia Jia[c]

[a]Department of Computer Science, Jinan University, Guangzhou, China
[b]Department of Computer Science, Loughborough University, LE11 3TU, UK
[c]Department of Computer and Information Science, University of Macau, China

Abstract

Data center operators deploy a variety of both physical and virtual network functions boxes (NFBs) to take advantages of inherent efficiency offered by physical NFBs with the agility and flexibility of virtual ones. However, such heterogeneity faces great challenges in correct, efficient and dynamic network policy implementation because, firstly, existing schemes are limited to exclusively physical or virtual NFBs and not a mix, and secondly, NFBs can co-exist at various locations in the network as a result of emerging technologies such as Software Defined Networking (SDN) and network function virtualization (NFV).

In this paper, we propose a Heterogeneous netwOrk pOlicy enforCement scheme (HOOC) to overcome these challenges. We first formulate and model HOOC, which is shown be to *NP-Hard* by reducing from the Multiple Knapsack Problem (MKP). We then propose an efficient online algorithm that can achieve optimal latency-wise NF service chaining amongst heterogenous NFBs. In addition, we also provide a greedy algorithm when operators prefer smaller run-time than optimality.Our simulation results show that HOOC is efficient and scalable whilst testbed implementation demonstrates that HOOC can be easily deployed in the data center environments.

Keywords: Data Center, Network Policy Management, Middleboxes, Network Function Virtualization (NFV), SDN Switches, Network Service Chaining

*Corresponding author
Email address: p.tso@lboro.ac.uk (Fung Po Tso)

1. Introduction

Data center operators deploy a great variety of network functions (NFs) such as firewall (FWs), content filter, intrusion prevention/detection system (IPS/IDS), deep packet inspection (DPI), network address translation (NAT), HTTP/TCP performance optimizer, load balancer (LB), and etc., at various points in the network topology to safeguard networks and improve application performance [1]. Each network function is responsible for specific treatment of received packets, including forwarding, dropping, rate-limiting, inspecting, and/or modifying packets. In practice, various permutations of or subsets of these functions form an ordered composition (or service chain) – as defined by a network policy [2] – that must be applied to packets in uni-directional or bi-directional manner. This process is also known as network service chaining [3]. Hence, *network policy enforcement implies correct and efficient chaining of network functions.*

Nowadays, network functions are either embedded in purpose-built proprietary hardware, i.e., middleboxes (MBs), or appear as virtual instances running on top of commodity servers through NFV (Network Function Virtualization). We term both hardware middleboxes and NFV servers as *Network Function Boxes* (NFBs). Physical NFBs are more efficient because they are built with dedicate hardware for optimizing the performance of specific functions but are proprietary and hence less extensible. On the other hand, virtualized NFBs have the agility for rapid on-demand deployment and greater degree of programmability for software automation but are less efficient due to virtualization overhead, resource sharing, and general-purpose hardware [4].

In addition to hardware middleboxes and general-purpose NFV servers, with the power of SDN (Software Defined Networking), some simple network function such as firewall and NAT can also be easily and efficiently implemented in SDN switches [5].

Obviously, except purpose-built physical NFBs, a network function can be

30 independently allocated to different servers and SDN switches in the network or collocated with other network functions within a switch or server [3][6].

In fact, today's data center operators adopt mixture of both physical and virtual NFBs to captialize on the efficiency of physical ones and the agility and flexibility of virtual ones [3].

35 Nevertheless, coming with this hybrid heterogeneous paradigm are significant challenges on the correct implementation of network policies in today's data centers: (1) Support for deployment of network policies is limited exclusively to either physical or virtualized NFBs. There is no existing mechanisms for supporting simultaneous use of both form factors [2][7][8][9]; (2) Large va-
40 riety of NFBs at distinct network locations means that the choices for correct service chaining has grown exponentially. We show that large variation in round trip times (RTTs) can be observed for NFBs with different capacity (detailed in Section 2). Given most data center workloads are latency-sensitive and are prone to unpredictable slowdown along the end-to-end links [10][11], *how could*
45 *we ensure that latency for all policy chains is optimal*?

In this paper, we propose a Heterogeneous netwOrk pOlicy enforCement scheme (HOOC), which is an adaptive network policy implementation scheme that will not only support the use of both physical and virtual NFBs but also minimize latency along the policy path (i.e. service chain) such that end-to-end
50 delay will become more predictable. Our experimental evaluation demonstrates that HOOC can achieve optimal placement of network functions amongst heterogeneous NFBs.

The contribution of this paper is fourfold.

1. We experimentally show that performance heterogeneity for running same
55 network functions on different NFBs.

2. We formulate HOOC and prove that it is NP-Hard, by reducing from the Multiple Knapsack Problem (MKP).

3. We model the heterogeneity of NFBs by constructing cost network graphs and propose an efficient online **Shortest Service Chain Path (SSP)**

60 algorithm for finding the shortest path (minimal latency) for any given policy in a cost network graph.

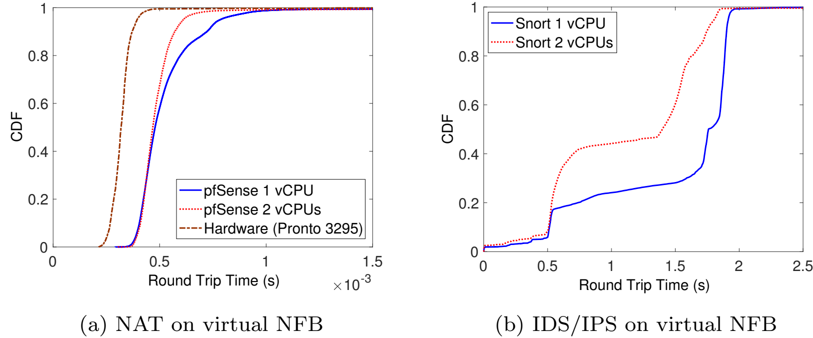
4. Our simulation results show that the HOOC scheme is efficient and scalable. Our testbed results show that the HOOC scheme is practical.

The remainder of this paper is structured as follows. Section 2 presents 65 our simple experiments on revealing performance heterogeneity across the same network function on different NFBs of various capacity. Section 3 describes the problem formulation and the model of HOOC. Efficient schemes for HOOC are proposed in Section 4, followed by testbed implementation the performance evaluation of HOOC in Section 5 and Section 6 respectively. Section 7 out- 70 lines related works, and Section 8 concludes the paper and indicates the future direction of this work.

2. NFB Performance Heterogeneity

In order to understand the extent to which the performance heterogeneity existing amongst the same network functions on different NFB configurations, 75 we have carried out a set of simple experiments using three commodity servers and one Pronto 3295 SDN switch. Each server is configured with an Intel's Xeon E5-1604 4 cores CPU, 16GB RAM and a dual port 1 Gbps NIC (Network Interface Card), and with Ubuntu 14.04 as operating system. One server has been used as virtualised NFB, with KVM (Kernel-based Virtual Machine) as 80 the hypervisor. The other two servers have been used for running *iPerf* [12] client and server respectively. Both the client and server were connected to the NFB directly via 1 Gbp/s links. We have also used a Pronto 3295 SDN switch to emulate a hardware NFB.

We have used two popular open-sourced software – Firewall (pfSense v2.3.1 85 [13]) and IDS/IPS (Snort v2.9.8 [14]) – as our network functions. For firewall experiments, a NAT has been created and used, meaning that the client and server resided in two different networks. For IDS/IPS experiments, both client



(a) NAT on virtual NFB

(b) IDS/IPS on virtual NFB

Figure 1: CDF of RTTs for pfSense and Snort NFBs with different numbers of allocated virtual CPU.

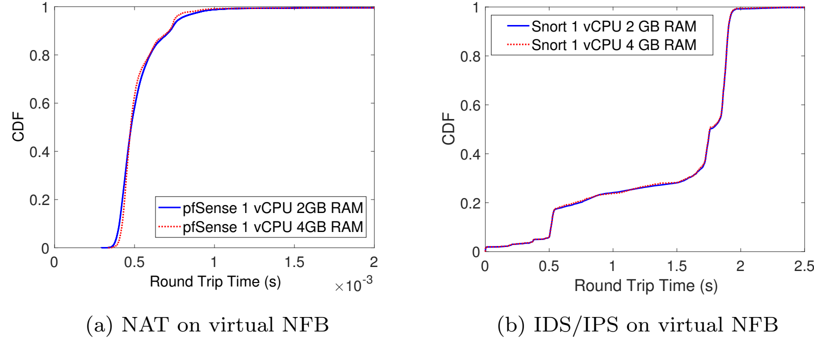
and server were in the same network, meaning that the two physical network ports on the NFB were bridged by software bridge. IDS/IPS rules used were default rules pulled from Snort website. In addition to virtualized firewall, we have also programmed the SDN controller to write some static flow entries to the Pronto switch to make it a simple hardware-based NAT.

In all experiments, we have used *iPerf* to stress the server with TCP requests and record the traffic with *tcpdump* on both client and server. Since we are particularly interested in the end-to-end latency, we have used *Wireshark* (*tshark*) to compute packet round-trip-time (RTT) from recorded traffic streams.

2.1. Correlation with number of CPUs

We first study the correlation of performance heterogeneity of network function with different number of allocated CPUs on NFB. In this set of experiments, we have first allocated only one vCPU (1 vCPUs, 2GB RAM) for both pfSense and Snort servers and then increased the number of vCPUs to two, while keeping the memory (2 vCPUs, 2GB RAM) and other configurations unchanged.

The computed RTT from the recorded traffic has been demonstrated in Figure 1. Since no links in this setup are over-subscribed, the likelihood of traffic congestion is low. Thus, processing delay accounts for significant portion of end-to-end latency. Clearly, Figure 1a shows that having twice as much hardware



(a) NAT on virtual NFB

(b) IDS/IPS on virtual NFB

Figure 2: CDF of RTTs for pfSense and Snort NFBs with different sizes of allocated memory.

resource does not significantly improve RTT as there is only about 5% improvement at the region above 80 percentile. In comparison, the hardware switch implementation has much smaller and predictable RTT, even at 99 percentile. Figure 1b shows more diverse performance results amongst two configurations for Snort IDS/IPS in which 2 vCPUs could give significantly better performance up to as much as 100%. The steps observed in figures are attributed to the different computation demands required by various intrusion detection rules. This means some packets are scrutinized more heavily whereas some are less.

In addition, we have also noticed that the magnitude of RTT for Snort is two orders higher than that of pfSense. This is because the pfSense's workload was mainly on examining the packet header for NAT translation, whereas for IDS/IPS the workload was mainly on deep packet inspection.

2.2. Correlation with size of memory

In this set of experiments, we have only altered the configuration (1 vCPU, 2GB RAM) to increase the size of memory from 1GB to 4GB (1 vCPU, 4GB RAM). The results shown in Figure 2 exhibit only small differences in performance across two configurations. Clearly, this set of experiments has revealed that the performance of network function is largely limited by NFB's processing capacity rather than its amount of memory (as long as it meets minimum requirements).

3. Problem Modeling

Table 1: Notations and Parameters

Symbol	Description
$\mathbb{B}, b_i$	$\mathbb{B}$ is set of all NFBs, $b_i \in \mathbb{B}$
$Cap(b_i), TypeSet(b_i)$	maximum capability and supported NF types of b_i
$\mathbb{N}, n_i$	$\mathbb{N}$ is set of all NFs, $n_i \in \mathbb{N}$
$Type(n_i), Req(n_i), Loc(n_i)$	function type of n_i, processing requirement of n_i and the NFB hosts n_i
$\mathbb{P}, p_i$	$\mathbb{P}$ is set of all network policies, $p_i \in \mathbb{P}$
src_i, dst_i	source and destination of p_i
$Len(p_i)$	number of NFs in p_i
P_i	all possible sequence of p_i with re-ordering
$D(n_i, n_j)$	dealy between n_i and n_j
t_s^i, t_w^i	service time and average waiting time of n_i
λ_i	packet arrival rate of n_i
$t_p(n_i)$	processing delay of n_i
$T(p_i)$	expected delay for the flow constrained by p_i
B_j	nodes in the jth tier of the service chain network

In this section, we will describe the heterogeneous network policy problem. Table 1 summaries notations used in the paper.

130 ### *3.1. Overview*

In this paper, as opposed to existing works which only consider homogeneous NFBs deployment, we consider a heterogeneous environment. Network functions can be implemented at various network locations, either in-network or

at-edge, and on different kinds of NFBs such as hardware middleboxes, commodity servers, and (SDN) switches/routers. These NFBs are distinctively different in the following ways:

- *Hardware middbleboxes* are vendor specific, proprietary boxes for providing specific network functions. Their designs are often optimized for performance and are less extensible. On the contrary,
- *NFV servers* are virtualized that can run multiple, and theoretically, any types of virtual network functions. As they are built on virtualization, better agility can be guaranteed.
- Some simple network functions can also be implemented on *switches* or *routers* such as VPN, simple firewalls which can only perform packet filtering, and load balancers. They are amongst hardware middleboxes. However, SDN can allow us to exploit the OpenFlow switches to increase the performance of service chain by installing some rules (i.e., network function) to their flow tables [5].

Since each types of NFs implementations above have their own advantages, we anticipate that the heterogeneous implementation of network functions will exist for the foreseeable future.

Denote $\mathbb{B} = \{b_1, b_2, \ldots\}$ to be the set of all NFBs in a data center. For a NFB b_i, $Cap(b_i)$ denotes the maximum processing capability of b_i, measuring in number of packets per second (*pps*), e.g., 3800 *pps* [15]. $TypeSet(b_i)$ specifies the set of supported network function types on b_i. NFV servers, theoretically, support all types of network function, while hardware MBs and switches can only support one or few types of network functions. Without loss of generality, we assume that the memory space of NFBs are enough to accommodate states information of all network functions, i.e., bottleneck is the processing capacity as shown in Section 2.

Let $\mathbb{N} = \{n_1, n_2, \ldots\}$ be the set of all network function instances in data center. The $Type(n_i)$ defines the function of n_i, e.g., *IPS/IDS*, *LB*, or *FW*.

$Req(n_i)$ is essentially the requirement of n_i on the processing capacity of NFBs in *pps*. $Loc(n_i)$ is the NFB that currently hosts n_i. One main objective in this paper is to find an appropriated NFB for $Loc(n_i)$.

The set of network functions in $\mathbb{N}$ may belong to different applications, and are deployed and configured by a centralized *Policy Controller* [9]. The centralized *Policy Controller* monitors and controls the liveness of network functions and NFBs, including addition, failure/removal or migration of a network function. Network administrators can specify and update policies through the *Policy Controller*.

The set of network policies is $\mathbb{P}$, which can be defined by users or administrators. In reality, one policy can be applied to multiple flows and a single flow can be subject to the governance of multiple policies. For each $p_i \in \mathbb{P}$, src_i and dst_i specify the source and destination of p_i respectively. All packets matched to them should be constrained by p_i. The ordered list contained in p_i defines the sequence of NFs that all flows matching policy p_i should traverse in order, and $p_i[j]$ refers to the jth NF. For example, $p_i = (n_1, n_2, n_3)$, where $Type(n_1) = FW, Type(n_2) = IPS, Type(n_3) = Proxy$. And $Len(p_i)$ is number of NFs of p_i.

All NFs in p_i must be assigned to appropriate NFBs beforehand, and we assume there are enough NFBs to accommodate all required network functions in data center. Since we consider heterogeneous network functions, there are various possible locations for each network function in p_i. For example, in the above example of p_i, $Loc(n_1)$ could be a core router, $Loc(n_2)$ could be a hardware NFB, and $Loc(n_3)$ could be a NFV server. An example of service chain is given in Figure 3. Next, we will consider the problem of heterogeneous policy placement.

3.2. Delays with network functions

There are many metrics to measure the efficiency of network function placement (service function chaining) for a policy such as communication cost [16][17]. In this paper, we mainly focus on the latency of a policy flow. *However, the*

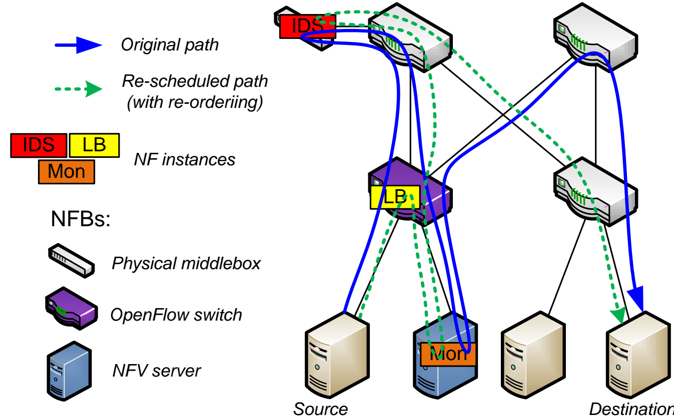


Figure 3: Service chain example for heterogeneous environment: red arrow shows original service chain path: Source→ $LB \to IDS \to Monitor \to$ Destination, green dash arrow shows the optimized path with re-ordering (Mon and IDS)

main idea in this paper can be easily applied to other metrics.

The total delay of a flow includes the transmission delay among adjacent network functions in the service chain and processing delay of network functions.

3.2.1. Transmission delays

In order to steer traffic to the service chain, either Policy Based Routing (PBR) or VLAN stitching can be used in data centers [3]. For either case, *the intended solution in this paper should be unaware of these schemes and is general and applicable to the schemes.* So, we do not consider the detailed routing between two NFBs.

Since, in production data centers, the transmission delay of links in its path are relatively stable and can be easily obtained/estimated through large-scale measurement [18], we assume the transmission delay between two network functions is known and can be obtained through the controller.

The controller will maintain a transmission delay matrix D, $D(n_i, n_j) = D(n_j, n_i)$ is the delay between n_i an n_j. $D(n_i, n_j) = -1$ if the delay is unknown

or they are unreachable. In either cases, paths with $D(n_i, n_j) = -1$ will not be considered for arrangement of service chains.

210 *3.2.2. Processing delays of network functions*

We define service time t_s^i as the time that n_i takes to process a packet. Since that many network functions such as proxies, firewalls and load balancers only process packet headers of which sizes are fixed, ignoring variable length data payloads. Thus, the service time t_s^i is a constant [19]. Specially, considering the 215 processing capacity $Req(n_i)$ of n_i, $t_s^i = 1/Req(n_i)$.

If packets arrival rates is smaller than the processing capacity of network function, the processing delay is equal to the service time. Otherwise, packets will be queued. For simplicity, we consider a M/D/1 queue, and network functions process packets in a First-Come-First-Service (FCFS) discipline. Then, 220 the processing delay is the summation of waiting time and service time. The packet arrival rate for n_i is the total rates of all flows that need to be processed by n_i, which is denoted by λ_i. The utilization $\rho_i = \lambda_i * t_s^i$. The average waiting time t_w^i of n_i is

$$t_w^i = \frac{t_s^i * \rho}{2(1-\rho)} = \frac{\lambda_i * {t_s^i}^2}{2(1-\lambda_i * t_s^i)} \tag{1}$$

Thus, the processing delay of n_i is:

$$t_p(n_i) = \begin{cases} t_s^i & \lambda_i \leq Req(n_i) \\ t_w^i + t_s^i & \lambda_i > Req(n_i) \end{cases} \tag{2}$$

225 *3.3. NF Behavior and Re-ordering of Service Chain*

We have surveyed a wide range of common network functions and service chains to understand their common behaviors and properties. Most of these NFs perform limited types of processing on packets, e.g., watching flows but making no modification, changing packet headers and/or payload. For example, in 230 the simplest case, a flow monitor (FlowMon) obtains operational visibility into the network to characterize network and application performance, and it never

modify packet and flows [3]. Some NFs, e.g., IDS, will check packet headers and payload, and raise alerts to the system administrator. Some NFs (such as firewalls and IPS) do not change packet headers and payload, but they use packet header information to make decision on whether to drop the packet or forward it. Some NFs (such as NAT and LB) may check IP/port fields in packet headers and rewrite these fields [7]. Others (such as traffic shaper) do not modify packet headers and payloads, but may perform traffic shaping tasks such as active queue management or rate limiting [20].

For a service chain, certain ordering requirement of NFs naturally exists due to the nature of the functions applied. For instance, for a service chain applied to North-South traffic in datacenters, a Web Optimization Control (WOC) is not effective on VPN traffic, requiring VPN termination prior to WOC [3]. For other service chain with IDS and FlowMon, since IDS never change the packet content, FlowMon can be applied to the traffic after IDS or placed prior to IDS. If the order of some NFs in a service chain is allowed to be re-organized, there could be more opportunities to improve performance by reducing the length of the service chain path such as the example shown in Figure 3.

In order to model these properties of NFs and leverage these properties, we can classify NFs into several classes according to their behaviors:

- Modifier (M): NFs that may modify the content of a packet (header or payload), e.g., NAT, Proxy;
- Shaper (Sh): NFs that perform traffic shaping tasks such as active queue management or rate limiting, e.g., rate limiter.
- Dropper: NFs that may drop packets of flows, but never modify header of payload of packets, e.g., firewall.
- Static : NFs do not modify the packet or its forwarding path, and in general do not belong to the classes above, e.g., FlowMon, IDS.

Table 2 summarizes the dynamic actions performed by different NFs that are commonly used today.

Table 2: Examples of the dynamic actions performed by different NFs that are commonly used today [7]

Network Functions	Input	Actions	Type
FlowMon	Header	No change	Static
IDS	Header, Payload	No change	Static
Firewall	Header	Drop?	Dropper
IPS	Header, Payload	Drop?	Dropper
NAT	Header	Rewrite header	Modifier
Load balancer	Header	Rewrite header	Modifier
Redundancy eliminator	Payload	Rewrite payload	Modifier

To preserve the correctness of service chain, users can specify constraints on the order of NFs in service chains. For example, we can change the order of static NFs, and move static NFs before Dropper NFs. However, we cannot move static NFs across Modifiers, as this might lead to incorrect operation.

265 In the example shown in Figure 3, the service chain is $LB \rightarrow IDS \rightarrow Monitor$ and the total service chain path from the source to destination has 10 hops. Since that both IDS and $Monitor$ are static NFs and do not modify packets, their orders can be switched. By switching the position of IDS and $Monitor$, the new service chain path (green dashed arrow in the figure) only has 270 8 hops. Furthermore, with heterogeneous NFBs (e.g., hardware or virtualized), there would be more opportunity for improving performance if re-ordering is allowed.

Considering the re-ordering of service chain, we define P_i to be a set of all possible NFs sequence of the service chain, i.e., $P_i = \{l_1, l_2, \ldots\}$. For ex- 275 ample, suppose the service chain of p_i is $Firewall_1 \rightarrow IDS_1 \rightarrow FlowMon_1$, and the position of IDS_1 and $FlowMon_1$ can be swapped. Then, $P_i = \{l_1 = (Firewall_1, IDS_1, FlowMon_1), l_2 = (Firewall_1, FlowMon_1, IDS_1)\}$. NFs of

p_i can be organized according to any sequence defined in P_i.

The policy p_i is called *satisfied* if and only if the following condition holds:

$$p_i[j] == l[j], \forall j = 1, 2 \ldots, Len(p_i), \exists l \in P_i \tag{3}$$

280 The final assigned sequence of p_i must be equal to l, where l can be any accepted list in P_i with re-ordering.

3.4. Heterogeneous network policy enforcement problem

The expected delay for the flow constrained by policy p_i is defined as:

$$\begin{aligned} T(p_i) &= D(src_i, p_i) \\ &+ \sum_{j=1}^{len(p_i)-1} (D(p_i[j], p_i[j+1]) + t_p(p_i[j])) \\ &+ D(p_i[Len(p_i)], dst_i) \end{aligned} \tag{4}$$

We aims to reduce the total delay by efficiently placing network functions 285 onto heterogeneous NFBs while strictly adhering to network policies. Denote $A(n_i)$ to be the NFB which hosts n_i, and $H(b_j)$ is the set of network functions hosted by b_j.

The *Heterogeneous Network Policy Enforcement problem* is defined as follows:

290 **Definition 1.** *Given the set of policies $\mathbb{P}$, NFBs $\mathbb{B}$ and delay matrix D, we need to find an appropriate allocation of network functions, which that minimizes the total expected end-to-end delays of the network:*

$$\begin{aligned} &\min \sum_{p_k \in \mathbb{P}} T(p_k) \\ &s.t. \quad p_k \textit{ is satisfied}, \forall p_k \in \mathbb{P} \\ &\quad A(n_i) \neq \emptyset \ \&\& \ |A(n_i)| = 1, \forall n_i \in p_k, \forall p_k \in \mathbb{P} \\ &\quad \textstyle\sum_{n_i \in H(b_j)} Req(n_i) < Cap(b_j), \forall b_j \in \mathbb{B} \end{aligned} \tag{5}$$

The first constraint ensure that network functions of all service chains are appropriately accommodated by one NFB. The second constraint is the capacity 295 constraint of all NFBs.

The above problem can be easily proven to be *NP-Hard*:

Proof. To show that *Heterogeneous Network Policy Enforcement problem* is NP-Hard, we will show that the Multiple Knapsack Problem (MKP) [21], whose decision version has already been proven to be strongly NP complete, can be reduced to this problem in polynomial time.

Consider a special case of *Heterogeneous Policy Enforcement problem* that the service chain of all policies contain only one network function. Assume that transmission delays between servers and NFBs are the same and there are enough NFBs, meaning that no NFBs are saturated.

Consider each network function n_i to be an item, where its requirement $Req(n_i)$ is item size. Each NFB b_j is a knapsack with limited capacity $Cap(b_j)$. The profit of assigning n_i to each NFB is the negative of the delays. Then the *Heterogeneous Network Policy Enforcement problem* becomes finding an allocation of all network functions to NFBs, maximizing the total profit. Therefore, the MKP problem is reducible to the *Heterogeneous Policy Enforcement problem* in polynomial time, and hence the *Heterogeneous Policy Enforcement problem* is NP-hard. □

4. Heterogeneous Policy Enforcement

In this section, we introduce *HOOC*, a Heterogeneous netwOrk pOlicy enforCement scheme.

4.1. Service chain network

We consider an online solution which process one service chain at a time when a new policy requirement arrives.

For each policy p_i, we need to find appropriate NFBs to accommodate all network functions in p_i with an objective to minimize its total expected delay $T(p_i)$. Considering re-ordering of service chain, for each candidate service chain $l \in P_i$, we construct a graph G^l, which is a m-tier directed graph ($m = Len(p_i)$).

Nodes in the jth tier are NFBs defined by B_j:

$$B_j = \{b_k | l[j] \in TypeSet(b_k) \text{ and } \sum_{n \in A(b_k)} Req(n) + Req(l[j]) \leq Cap(b_k), \forall b_k \in \mathbb{B}\} \tag{6}$$

For a node x in jth ($j \leq m-1$) tier and y in $(j+1)$th tier, there is a directed edges from x to y if y is reachable from x and the weight of the edge is $D(x, y) + t_p(y)$. It is possible that both x and y are the same NFB. In this case, $D(x, y) = 0$.

Then, for each $l \in P_i$, we can construct a graph G^l, and all those graphs can be merged into one single graph G. During the merge operation, for any $l_1 \in P_i$ and $l_2 \in P_i$ ($l_1 \neq l_2$), if $l_1[j] = l_2[j]$, nodes in j-th tier of G^{l_1} can be merged with nodes of j-th tier in G^{l_2} accordingly. If two neighbor nodes x and y in G^{l_1} are merged to neighbor nodes x' and y' in G^{l_2}, the link between them must have the same weight and can be merged too.

Flow originates from the source (src_i) and terminate at the sink (dst_i). For a node x in 1st tier, the weight of the directed edges from src_i to x is $D(src_i, x) + t_p(s)$. For a node y in lth tier, the weight of the directed edges from y to dst_i is $D(y, dst_i)$.

The resulted graph G is called the *Service Chain Network* of p_i. An example of service chain network is given in Figure 4.

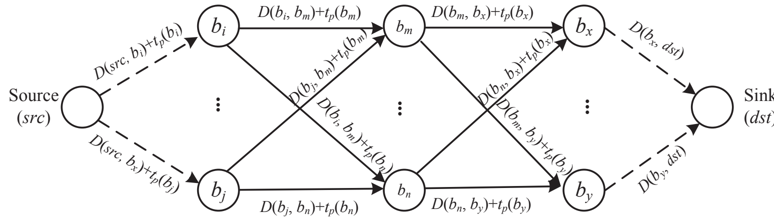


Figure 4: Example of service chain network with length of 3.

4.2. Shortest service chain path

340 According to the construction process of service chain networks, any paths from source to sink need to traverse all tiers, i.e., all NFs in the service chain. Edges among different tiers ensure that all those NFs are in correct order that are acceptable in P_i. And weights of edges are their corresponding delay. Thus, it is clear that the route with the smallest expected latency for a flow is the shortest 345 path from source to sink. We referred this path as *ssp* (Shortest service chain path). However, since nodes in different tiers of the service chain network can be the same NFB with limited capacity, we can not simply re-use traditional shortest first path algorithms, e.g., Dijkstra, Floyd-Warshall.

The difficulty here is that two nodes that belong to different tiers in the 350 service chain network, say x and y, may be in the same NFB and share the same capacity. If we assign p_i to x, it may saturate the NFB such that y can not further accept p_i. In this case, we call them *conflict nodes*. A path from the source will be *blocked* by the latter one of the *conflict nodes*.

Hence, we design the SSP (Shortest Service Chain Path) algorithm to find 355 the shortest path in this situation, as shown in Algorithm 1. The $d(v)$ is used to maintain the distance from source to vertex v. It is initialized to be infinite and will be relaxed during the course of the algorithm. The set S contains all vertices whose final shortest distance from the source have already been determined. Conflict nodes are handled in line 14. The shortest service chain 360 path are maintained in *prev* and can be obtained through $getPath()$.

Obviously, the *shortest service chain path* in Algorithm 1 is a variant of the *Single-Source Shortest Path (SSSP)* problem [22]. We have adapted it to handle the conflict nodes during discovering the optimal path and it can be easily proven to be able to always find the optimal path. And the complexity of the 365 algorithm depends on the way of finding the vertex v with the smallest distance $d(v)$, i.e., the *argmin* operation. Because paths with *conflict nodes* failed to reach the destination, not all vertices and edges are checked in Algorithm 1. Thus, each vertex $v \in V$ is added to set S at most once (line 6 $\sim$ 10), and each edge in E is examined in the for loop of lines 12 $\sim$ 20 at most once

370 during the course of the algorithm. A *priority queue*, which is a data structure consisting of a set of item-key pairs, can be implemented for efficient operation of distanc for each vertex. Operations supported by *priority queue* can be used to implement Algorithm 1: *insert*, e.g., implicit in line 2; *extract-min*, returning the vertex with the minimum distance in line 6, i.e., the *argmin* 375 operation; and *decrease-key*, decreasing the distance of a given vertex in line 15. Furthermore, Fibonacci heaps [23] implement *insert* and *decrease-key* in $O(1)$ amortized time, and *extract-min* (i.e., *argmin*) in $O(\log n)$ amortized time, where n is the number of elements in the priority queue [22]. So, by using Fibonacci heaps, the running time of Algorithm 1 is $O(|E| + |V| \log |V|)$, where 380 $|E|$ is the number of edges and $|V|$ is the number of vertices in the cost network.

Network policy is often stable and is not transient. However, we reckon the fact that traffic demand could change slowly over time and it is necessary to adapt to the changes. This can be easily achieved in HOOC through SDN mechanism: the *Policy Controller* can periodically poll switches for traffic statistics 385 to look for changes in traffic demand in specific part of network topology, and then trigger HOOC to re-optimize the policies that have been affected.

4.3. Greedy Approach

Algorithm 1 ensures the optimality of the service chain path. However, it has one major drawback that its $O(|E| + |V| \log |V|)$ time complexity. Thus, we 390 also propose a greedy approach, which trades off small accuracy for significantly faster speed.

The greedy approach of HOOC is described in Algorithm 3. The main idea of Greedy is that: for each element in the service chain, the algorithm will choose a NFBs with the smallest delay to the source or previous NF in the 395 service chain. If current path is *blocked* by a *conflict node*, the algorithm will fall back to previous NF and choose the NFB with 2nd smallest delay. This process will continue until the destination is reached, or there is no available path. If multiple candidated service chain are available in P_i, the B_j contains acceptable NFBs defined in Equation 6. Specially, for any $l_1 \in P_i$ and $l_2 \in P_i$

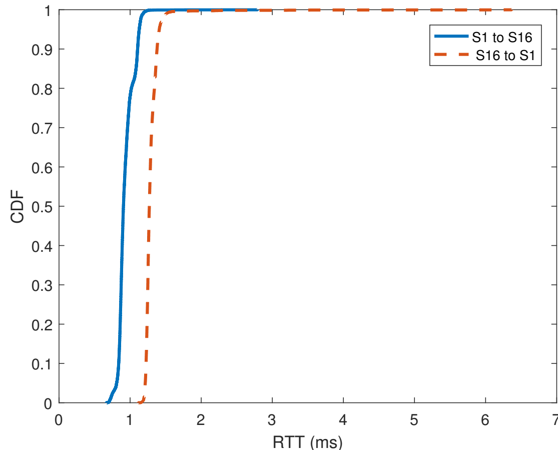


Figure 5: An example RTT for server pair (s1,s16)

400 $(l_1 \neq l_2)$, if $l_1[j] = l_2[j]$, same NFBs obtained for l_1 and l_2 will be merged as a single node, otherwise, they will be treated as different nodes.

5. Implementation

5.1. Testbed

We have implemented on a proof-of-concept testbed consists of 16 Raspberry
405 Pis (Model 2B) [24], two Pronto 3295 SDN (2x48 ports) switches and a Ryu SDN controller running on an Intel's Xeon E5-1604 4 cores CPU and 16GB RAM. We constructed a fat-tree topology ($k = 4$) by logically slicing [25] two pronto switches into 20 4-port SDN switches. As a result of slicing, we had to manually construct the topology graph in the Ryu controller. However, we note
410 that Ryu has a built-in feature that can automatically learn network topologies if regular switches are used. Our example NFs are mainly simple container-based firewalls [26]. We have also attached an IDS/IPS used in Section 2 to one of spare SDN switch ports and is seen as a hardware NFB.

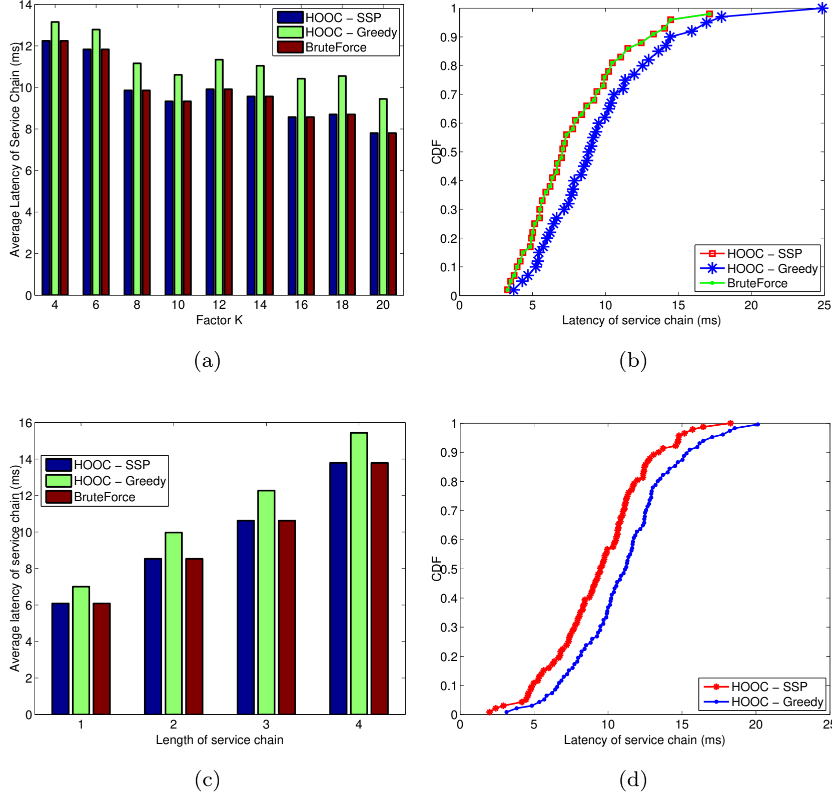


Figure 6: Comparison of latency of service chain: (a) Average latency for various network scale; (b) Latency of service chain for $k = 20$; (c) Average latency for various length of service chain; (d) Latency of service chain for $length = 4$

5.2. Link latency

415 In order to obtain needed link latency we have implemented a reduced version of Pingmesh Agent [18] using C++ for better performance and accuracy. This Pingmesh Agent pings all servers (i.e. Raspberry Pis) using *TCPing*, and measures round-trip-time (RTT) from the TCP-SYN/SYN-ACK intervals. An example server pair $(s1, s16)$ RTT is shown in Figure 5. The average memory 420 footprint is less than 2MB, and the average CPU usage is less than 1%. Ping traffic is very small and ping interval is configurable according to actual needs.

The ping results are uploaded to the controller periodically for constructing

all pairs end-to-end latency table which can be queried using host IP address. This is because we assume that most of deployed NFs will run in commodity servers. There are also some in-network hardware NFBs, as defined in 3.1, that are either SDN switches or attached directly to the switches. Hence the delay from/to these particular devices can be queried through OpenFlow's port statistics APIs or other technique such as OpenNetMon [27].

The processing delay of network functions is obtained from t_s^i, which is inverse proportional to NF's capacity. We did not consider queueing delay in our testbed implementation because our algorithm ensures that NFBs are not overloaded. We also note here that there are also some other techniques that are useful for monitoring processing capacities such sFlow [28].

5.3. Policy controller

The policy controller is implemented as an application module in Ryu. We have chosen Ryu because it has a built in integration for Snort [14] that enables bidirectional communication using unix domain socket. The controller interacts with NFBs that host firewalls using OpenFlow protocol. Although frameworks such as OpenNF [9] can also be added to enrich functionality of the controller, we note that the scope of this paper is to provide a proof-of-concept implementation rather than a full-blown testbed.

In addition to managing NFBs, the controller is also responsible for collecting link latency from Pingmesh Agent and maintaining an in-memory all-pair unidirectional end-to-end latency table which is essential to the HOOC scheme.

6. Evaluation

6.1. Evaluation environment and setup

In order to study the performance of HOOC scheme at scale, we have extensively evaluated it via ns-3 simulations in a fat-tree topology with factor k ranged from 4 to 20 meaning that there are at most 2000 servers and 500 switches in these setups. The same controller which we use for testbed has been used during simulation via ns-3's OpenFlow module.

Each NFB in our simulations is modeled with random residual capacity (number of packets it can process per second) and a set of network function types that it supports. Therefore, a NFB can accept a network function as long as it has sufficient residual capacity and the network function's type is amongst its support list. We also note that NFV servers can support any types of network functions. All NFBs are deployed in the network, including OpenFlow switches, hardware middleboxes and NFV servers.

In all experiments, traffic flows are randomly generated to transmit packets between two servers. Each flow is required to traverse a sequence of various network functions – the service chain – before being forwarded to their destination as specified by policies. In our experiments, the service chains are comprised of 1∼4 network functions (normal distribution) including FW, IPS, RE, LB, IDS and (traffic) Monitor [3]. A centralized controller is implemented to collect all network information that is needed, as defined in Section 3 to perform the HOOC scheme.

Both optimal and greedy approaches for HOOC are implemented. For simplicity, the scheme using SSP to achieve optimal schedule for a service chain is referred as HOOC-SSP, and the greedy approach is referred as HOOC-Greedy. In order to compare and contrast the performance of HOOC, we have also implemented a *Brute-force* approach: By using a DFS (Depth-first search) method, Brute-force approach exhaustively search all NFBs and all possible service chain allocation paths to find the one with smallest latency. Brute-force will give the optimal results but it is not suitable for large-scale network as the cost for searching all permutations will become prohibitively expensive as the search space grows.

6.2. Evaluation results

We first study the performance of HOOC with regard to the latency of service chain as demonstrated in Figure 6. Figure 6a shows the average latency of all service chain under different network scales with the factor k of fat-tree ranging from 4 to 20. It shows that on average HOOC-SSP can always find a service

chain path with the same latency as that of Brute-force, which is optimal. In comparison, the HOOC-Greedy approach cloud fall behind both HOOC-SSP and Brute-force by up to 23%.

We further show a detailed breakdown view in Figure 6b that HOOC-SSP and Brute-force schemes have identical CDF of latency for all policies for a large scale network when $k = 20$. Particularly, they can outperform HOOC-Greedy scheme by 38% at 99 percentile.

Figure 6c reveals that average latency increases linearly with the length of service chain when all NFBs have sufficient capacity for accommodating all network functions. The breakdown of CDF for latency of service chain whose length is comprised of four network functions shown in Figure 6d. It unveils that amongst HOOC's two algorithms, HOOC-SSP can outperform HOOC-Greedy by 21%.

Next we study the performance of different schemes in term of system running time. This is essentially to test the performance of HOOC controller for its efficiency and scalability in cloud data center environment. Figure 7 shows the average total running time to process a policy increases exponentially for all schemes. Nevertheless, as we can see from this figure that HOOC-Greedy is the most efficient methods, consuming only $2.8s$ and $3.7s$ for $k = 18$ and $k = 20$ respectively to complete a cycle. This is because HOOC-Greedy scheme has the smallest search space. On the contrary, HOOC-SSP can complete one NFs placement cycle for $k = 18$ and $k = 20$ at $10s$ and $23s$ respectively, and Brute-force takes up to $149s$ and $232s$ for $k = 18$ and $k = 20$ respectively. The results indicate that HOOC-SSP and HOOC-Greedy can be nearly 9 and 61 times faster than Brute-force. Among HOOC-SSP and HOOC-Greedy, the latter is 5 times more efficient that the former one.

As we have already presented in Section 4 that HOOC-SSP is comprised of constructing a service chain network and finding shortest service chain paths. Figure 7 also demonstrates that 63% of HOOC-SSP time are consumed on constructing the service chain network, whereas finding shortest service chain paths merely accounts for 37% of the time. Clearly, this indicates that in order

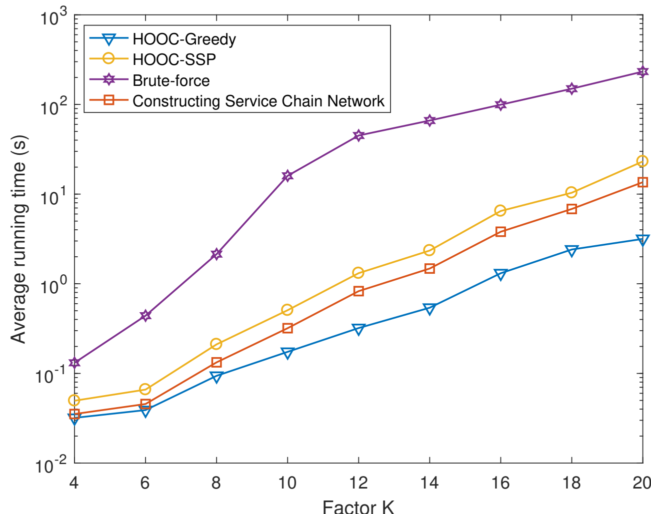


Figure 7: Performance comparison on running time

to further improve the efficiency of HOOC-SSP whilst retaining optimality, we should investigate into optimizing the efficiency of constructing service chain network. We will leave this as part of our future work. However, Figure 7 demonstrates that when efficiency becomes the foremost consideration HOOC-Greedy can strike a good balance between efficiency and its approximation to the optimal.

7. Related Works

The configuration of network connectivity is governed by network policies. When deployed, a policy is translated and implemented as one or more packet processing rules in a diverse range of "middleboxes" (MBs) such as firewalls (e.g. ALLOW TCP 80), load balancers, Intrusion Detection and Prevention Systems (IDS/IPS), and application acceleration boxes [29]. With network programmability enabled by SDN and NFV technologies, such rules can also be implemented outside of traditional "middleboxes" in network switches [30] as well as end-hosts [31]. One of the design requirements for today's cloud data

centres is to support the insertion of new middleboxes [32].

Recent studies have focused primarily on exploiting SDN and NFV to ensure correct policy compositions and enforcement [2][7][8], consolidating policy rules to end hosts [31] and network switches [33], or providing a framework for migrating middleboxes states [9], or policy-aware application placement to incorporate policy requirements [16][29][34].

Nevertheless, this body of work has only partially addressed the problem since, with SDN and NFV, both the number of entities that generate and implement policies independently and dynamically have increased manyfold. Given the large variety of network function entities in terms of both types and locations, inappropriate selections not only eliminate the advantage of SDN and NFV but could also cause severe consequences including data centre outage.

Many data centre applications are sensitive to latencies. One source of latency is network congestion as throughput-intensive applications causes queueing at switches that delays traffic from latency-sensitive applications. Existing techniques to combat queueing are to prioritise flows such that packets from latency-sensitive flows can "jump" the queue [11]; to centrally schedule all flows for every server so no flows will have to queue [35]; or to pace end host packets to achieve guaranteed bandwidth for guaranteed queueing [10].

These techniques assume shortest path forwarding. Today's data centre fabrics have rich path-redundancy in nature, non-shortest paths can be exploited to use path redundancy and spare capacity for mitigating network congestion [36]. As policy rules chaining can effectively shape the network traffic (packets need to follow policy path), they can be chained over non-shortest paths to mitigate congestion-led queueing since propagation delay on physical links are predictable and smaller than queueing delay.

A primary study on heterogeneous network function boxes environment is provided in our previous work [37] and a HOPE scheme is proposed. However, HOOC is different with previous work in the following ways: Firstly, a thorough test-bed experiments are also performed to show the heterogeneity among different NFB implementations; Secondly, the service chain re-ordering is consid-

ered, where NFs can be opportunistically re-ordered for improving performance. Thirdly, The detailed implementation of the Greedy version is introduced in this paper; Finally, a proof-of-concept testbed and some issues of implementations in practice are discussed.

8. Conclusion

Network policies and service chains are important for the security and reliability of data center network today. In practice, network functions of policies can be deployed in different environment, e.g., OpenFlow switches, hardware middleboxes and NFV servers. Such heterogeneous environment for policy allocation remain unexplored in previous research works. In this paper, we study the Heterogeneous Policy Enforcement Problem with a focus on the latency. We first prove that the optimization problem is NP-Hard, then simplified the problem and proposed HOOC, which is proved to be able to find the optimal service chain path for each policy. Extensive simulation results and comparisons with Brute-force approach have demonstrated high effectiveness and optimality of HOOC. The future direction of this work will be to investigate efficient service chain network construction.

Acknowledgements

The work has been partially supported in part by Chinese National Research Fund (NSFC) No. 61772235 and 61402200; the Fundamental Research Funds for the Central Universities (21617409); the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/2 and EP/P004024/1; DCT-MoST Joint-project No. (025/2015/AMJ); University of Macau Funds No. CPG2018-00032-FST & SRG2018-00111-FST; NSFC Key Project No. 61532013; National China 973 Project No. 2015CB352401; Shanghai Scientific Innovation Act of STCSM No.15JC1402400 and 985 Project of Shanghai Jiao Tong University: WF220103001.

References

[1] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan, et al., Steering: A software-defined networking for inline service chaining, in: Network Protocols (ICNP), 2013 21st IEEE International Conference on, IEEE, 2013, pp. 1–10.

[2] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, Y. Zhang, PGA: Using graphs to express and automatically reconcile network policies, in: ACM SIGCOMM, 2015.

[3] Surendra, M. Tufail, S. Majee, C. Captari, S. Homma, Service function chaining use cases in data centers, Tech. Rep. draft-ietf-sfc-dc-use-cases-05, IETF SFC WG (August 2016).

[4] B. Han, V. Gopalakrishnan, L. Ji, S. Lee, Network function virtualization: Challenges and opportunities for innovations, IEEE Communications Magazine 53 (2) (2015) 90–97. doi:10.1109/MCOM.2015.7045396.

[5] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, T. Lakshman, Application-aware data plane processing in sdn, in: HotSDN, ACM, 2014.

[6] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, G. Shi, Design and implementation of a consolidated middlebox architecture., in: NSDI, 2012, pp. 323–336.

[7] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, M. Yu, Simple-fying middlebox policy enforcement using SDN, ACM SIGCOMM Computer Communication Review 43 (4) (2013) 27–38.

[8] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, J. C. Mogul, Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags, in: Proc. USENIX NSDI, 2014.

[9] A. Gember-Jacobson, C. P. Raajay Viswanathan, R. Grandl, J. Khalid, S. Das, A. Akella, OpenNF: enabling innovation in network function control, in: Proc. of ACM SIGCOMM, 2014, pp. 163–174.

615 [10] K. Jang, J. Sherry, H. Ballani, T. Moncaster, Silo: predictable message latency in the cloud, ACM SIGCOMM Computer Communication Review 45 (4) (2015) 435–448.

[11] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, J. Crowcroft, Queues dont matter when you can jump them!, in:
620 NSDI 2015.

[12] iperf.
URL https://iperf.fr

[13] Electric Sheep Fencing LLC, pfsense.
URL https://blog.pfsense.org/

625 [14] Cisco, Snort.
URL https://www.snort.org

[15] Z. Liu, X. Wang, W. Pan, B. Yang, X. Hu, J. Li, Towards efficient load distribution in big data cloud, in: IEEE ICNC, 2015, pp. 117–122.

[16] L. Cui, F. P. Tso, D. P. Pezaros, W. Jia, W. Zho, Policy-aware virtual
630 machine management in data center networks, IEEE ICDCS 2015.

[17] L. Cui, R. Cziva, F. P. Tso, D. P. Pezaros, Synergistic policy and virtual machine consolidation in cloud data centers, IEEE INFOCOM 2016.

[18] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al., Pingmesh: A large-scale system for data cen-
635 ter network latency measurement and analysis, in: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, ACM, 2015, pp. 139–152.

[19] P. Duan, Q. Li, Y. Jiang, S.-T. Xia, Toward latency-aware dynamic middlebox scheduling, in: Computer Communication and Networks (ICCCN), 2015 24th International Conference on, IEEE, 2015, pp. 1–8.

[20] A. Bremler-Barr, Y. Harchol, D. Hay, Openbox: A software-defined framework for developing, deploying, and managing network functions, in: Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference, ACM, 2016, pp. 511–524.

[21] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack problems, Springer Verlag, 2004.

[22] A. Schrijver, Combinatorial optimization: polyhedra and efficiency, Vol. 24, Springer Science & Business Media, 2002.

[23] M. L. Fredman, R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, Journal of the ACM (JACM) 34 (3) (1987) 596–615.

[24] F. P. Tso, D. R. White, S. Jouet, J. Singer, D. P. Pezaros, The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures, in: 2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops, 2013, pp. 108–112. doi:10.1109/ICDCSW.2013.25.

[25] Z. Bozakov, P. Papadimitriou, Autoslice: automated and scalable slicing for software-defined networks, in: Proceedings of the 2012 ACM conference on CoNEXT student workshop, ACM, 2012, pp. 3–4.

[26] R. Cziva, S. Jouet, D. Pezaros, Container-based network function virtualization for software-defined networks, in: Computers and Communications (ISCC), 2015 IEEE 20th International Symposium on, 2015.

[27] N. L. Van Adrichem, C. Doerr, F. A. Kuipers, Opennetmon: Network monitoring in openflow software-defined networks, in: 2014 IEEE Network Operations and Management Symposium (NOMS), IEEE, 2014, pp. 1–8.

665 [28] P. Phaal, S. Panchen, N. McKee, Inmon corporations sflow: A method for monitoring traffic in switched and routed networks, Tech. rep., RFC 3176 (2001).

[29] L. E. Li, V. Liaghat, H. Zhao, M. Hajiaghayi, D. Li, G. Wilfong, Y. R. Yang, C. Guo, PACE: Policy-aware application cloud embedding, in: Proceedings
670 of 32nd IEEE INFOCOM, 2013.

[30] A. Gember, P. Prabhu, Z. Ghadiyali, A. Akella, Toward software-defined middlebox networking, in: Proceedings of the 11th ACM Workshop on Hot Topics in Networks, ACM, 2012, pp. 7–12.

[31] L. Popa, M. Yu, S. Y. Ko, S. Ratnasamy, I. Stoica, CloudPolice: taking
675 access control out of the network, in: ACM SIGCOMM Workshop on Hot Topics in Networks, 2010.

[32] L. Avramov, M. Portolani, The Policy Driven Data Center with ACI: Architecture, Concepts, and Methodology, Cisco Press, 2014.

[33] M. Moshref, M. Yu, A. B. Sharma, R. Govindan, Scalable rule management
680 for data centers., in: USNIX NSDI, 2013.

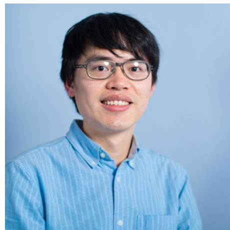
[34] L. Cui, F. P. Tso, D. P. Pezaros, W. Jia, Plan: A policy-aware vm management scheme for cloud data centres, IEEE/ACM Utility & Cloud Computing (UCC) 2015.

[35] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, H. Fugal, Fastpass: A
685 centralized zero-queue datacenter network, in: ACM SIGCOMM 2014.

[36] F. P. Tso, G. Hamilton, R. Weber, C. S. Perkins, D. P. Pezaros, Longer is better: exploiting path diversity in data center networks, IEEE ICDCS 2013.

[37] L. Cui, F. P. Tso, W. Jia, Heterogeneous network policy enforcement in
690 data centers, in: Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on, IEEE, 2017, pp. 552–555.

Lin Cui is currently with the Department of Computer Science at Jinan University, Guangzhou, China. He received the Ph.D. degree from City University of Hong Kong in 2013. He has broad interests in networking systems, with focuses on the following topics: cloud data center resource management, data center networking, software defined networking (SDN), virtualization, distributed systems as well as wireless networking.



Fung Po Tso received his BEng, MPhil and PhD degrees from City University of Hong Kong in 2006, 2007 and 2011 respectively. He is currently lecturer in the Department of Computer Science at the Loughborough University. Prior to joining Loughborough, he worked as SICSA Next Generation Internet Fellow at the School of Computing Science, University of Glasgow during 2011-2014 and lecturer in Liverpool John Moores University during 2014-2017. He has published more than 20 research articles in top venues and outlets. His research interests include: network policy management, network measurement and optimisation, cloud data centre resource management, data centre networking, software defined networking (SDN), distributed systems as well as mobile computing and system.

Weijia Jia is currently a chair Professor at University of Macaua. He is leading currently several large projects on next-generation Internet of Things, environmental sensing, smart cities and cyberspace sensing and associations etc. He received BSc and MSc from Center South University, China in 82 and 84 and PhD from Polytechnic Faculty of Mons, Belgium in 1993 respectively. He worked in German National Research Center for Information Science (GMD) from 93 to 95 as a

research fellow. From 95 to 13, he has worked in City University of Hong Kong as a full professor. From 14 to 17, he has worked as a Chair Professor in Shanghai Jiaotong University. He has published over 400 papers in various IEEE Transactions and prestige international conference proceedings.

Algorithm 1 SSP:Shortest Service Chain Path

Input: Service chain Network $G(V, E)$, p_i, $\mathbb{B}, \mathbb{N}, D$

Output: Shortest service chain path to dst_i

1: $S \leftarrow \emptyset$
2: $d(v) \leftarrow \infty, \forall v \in V$
3: $prev[v] \leftarrow$ undefined, $\forall v \in V$
4: $d(src_i) \leftarrow 0$
5: **while** $S \neq V$ **do**
6: $\quad u \leftarrow argmin_{v \in V \setminus S}\ d(v)$
7: $\quad$ **if** $u = dst_i$ **then**
8: $\quad\quad$ **break**
9: $\quad$ **end if**
10: $\quad S \leftarrow S \cap \{u\}$
11: $\quad n_k \leftarrow$ network function in p_i that will be placed in u
12: $\quad$ **for each** neighbor v of u **do**
13: $\quad\quad$ **if** $d(v) > d(u) + D(u, v)$ **then**
14: $\quad\quad\quad$ **if** $v \notin getPath(prev, u)$ **or** $\sum_{n_j \in H(v)} Req(n_j) + Req(n_k) \leq Cap(v)$ **then**
15: $\quad\quad\quad\quad d(v) \leftarrow d(u) + D(u, v)$
16: $\quad\quad\quad\quad prev[v] \leftarrow u$
17: $\quad\quad\quad\quad prev[v].nf \leftarrow n_k$
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **end if**
20: $\quad$ **end for**
21: **end while**
22: **return** $getPath(prev, dst_i)$

Algorithm 2 *getPath*$(prev, dst)$

```
1: ssp ← ∅
2: l ← ∅
3: u ← dst
4: while prev[u] is defined do
5:     insert u at the beginning of ssp
6:     insert prev[u].nf at the beginning of l
7:     u ← prev[u]
8: end while
9: insert u at the beginning of ssp
10: return ssp and l
```

Algorithm 3 Greedy

Input: $p_i, \mathbb{B}, \mathbb{N}, D$

Output: Service chain path to dst_i

```
1: path ← ∅
2: B'_j ← B_j, ∀j = 1, 2 ..., Len(p_i)
3: j ← 1
4: while j ≤ Len(p_i) do
5:     if B'_j = ∅ then
6:         j ← j − 1
7:         if j < 1 then
8:             path ← ∅                                  ▷ no available path
9:             break
10:        end if
11:        remove last node in path
12:        B'_{j+1} ← B_{j+1}
13:        continue
14:    end if
15:    u ← argmin_{v∈B'_j} D(path[end], v)
16:    B'_j ← B'_j \ {u}
17:    n_k ← network function in p_i that will be placed in u
18:    if u ∉ path or Σ_{n'∈H(u)} Req(n') + Req(n_k) ≤ Cap(u) then
19:        append u at the end of path
20:        if j = Len(p_i) then
21:            break
22:        else
23:            j ← j + 1
24:        end if
25:    end if
26: end while
27: return path
```