CoCo: Compact and Optimized Consolidation of Modularized Service Function Chains in NFV

Zili Meng[1,2], Jun Bi[1,2], Haiping Wang[1,2], Chen Sun[1,2], Hongxin Hu[3]
[1]Institution for Network Sciences and Cyberspace, Tsinghua University
[2]Tsinghua National Laboratory of Information Science and Technology (TNList)
[3]Clemson University
mengzl15@mails.tsinghua.edu.cn, junbi@tsinghua.edu.cn, hpwang@seu.edu.cn, c-sun14@mails.tsinghua.edu.cn, hongxih@clemson.edu

Abstract—**The modularization of Service Function Chains (SFCs) in Network Function Virtualization (NFV) could introduce significant performance overhead and resource efficiency degradation due to introducing frequent packet transfer and consuming much more hardware resources. In response, we exploit the *lightweight* and *individually scalable* features of *elements* in Modularized SFCs (MSFCs) and propose CoCo, a *compact* and *optimized* consolidation framework for MSFC in NFV. CoCo addresses the above problems in two ways. First, CoCo Optimized Placer pays attention to the problem of *which elements to consolidate* and provides a performance-aware placement algorithm to place MSFCs compactly and optimize the global packet transfer cost. Second, CoCo Individual Scaler innovatively introduces a *push-aside scaling up* strategy to avoid degrading performance and taking up new CPU cores. To support MSFC consolidation, CoCo also provides an automatic runtime scheduler to ensure fairness when elements are consolidated on CPU core. Our evaluation results show that CoCo achieves significant performance improvement and efficient resource utilization.**

I. Introduction

Network Function Virtualization (NFV) [12] was recently introduced by replacing traditional hardware-based dedicated middleboxes with virtualized Network Functions (vNFs). Compared to the legacy network, NFV brings benefits of easy development, high elasticity, and dynamic management. Meanwhile, network operators often require traffic to pass through multiple vNFs in a particular sequence (e.g. Firewall⇒NAT⇒Load Balancer), which is commonly referred to as a Service Function Chain (SFC) [16]. To fasten the development of vNFs, many recent research efforts [6], [7], [9], [24] proposed to break traditionally monolithic Network Functions (NFs) into processing *elements*, which could form a Modularized Service Function Chain (MSFC). For example, Intrusion Detection System (IDS) can be broken into a Packet Parser element and a Signature Detector element. In this way, new vNFs could be built based on a library of elements, which could significantly reduce human development hours.

However, introducing modularization into NFV brings two major drawbacks. First, in NFV networks, each vNF is usually deployed in the form of Virtual Machine (VM) with separated CPU cores and isolated memory resource [28]. When traversing a SFC, a packet has to be queued and transferred between VMs, which could introduce communication latency [13]. An MSFC requires more times of packet transmission between elements than its corresponding SFC, which may degrade the chain performance. Second, due to modularization, to deploy an MSFC, we need to consume much more (possible $2\times$ or more) hardware resources to accommodate all processing elements compared with a SFC with monolithic vNFs, which compromises resource efficiency.

Some research efforts have been devoted to addressing the problems above. OpenBox [9] addressed the performance problem by merging the common elements used by different vNFs in an MSFC to decrease the latency. However, OpenBox is constrained for limited cases where an MSFC comprises repeated elements whose internal rules belonging to different vNFs do not conflict with each other. NFVnice [15] addressed the resource efficiency problem by consolidating several NFs onto a CPU core with containers. However, it was designed at NF-level and ignorant of the new problems of modularization such as frequent inter-VM packet transfer. Also, it did not consider the placement problem of *which elements to consolidate*, which is also significant to improve performance and resource efficiency. Inappropriate consolidation may worsen performance by transferring packets repeatedly.

At the same time, a closer look into the modularization technique reveals some features of modularization that could benefit both the performance and resource efficiency of MSFCs. Modularization introduces processing elements that are *lightweight* and *individually scalable* [14], [21]. Therefore, we could consolidate several lightweight elements on the same VM, i.e. CPU core, to reduce hardware resource consumption and improve resource efficiency. Also, by considering which elements to consolidate, performance can be improved by reducing inter-VM packet transfer inside MSFC. Furthermore, in the situation where an element is overloaded, we can only scale out the overloaded lightweight element itself individually instead of its corresponding monolithic NF, which could significantly reduce the scaling cost [11]. The scaled out replica can also be consolidated onto an already working VM without consuming an extra CPU core to further save resource.

Therefore, based on the above observations, we propose CoCo, a compact and optimized element consolidation framework to improve the performance and resource utilization efficiency for MSFC in NFV. To the best of our knowledge, CoCo is the first framework addressing the optimal consolidation for MSFC in NFV. The key idea of CoCo is to reduce inter-VM packet transfer and fully utilize the processing power of

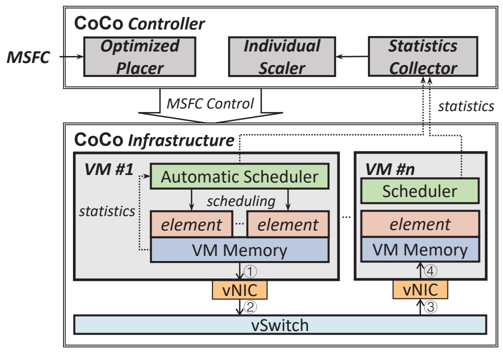


Fig. 1. CoCo Framework Overview

CPU by consolidating appropriate elements on the same VM. However, we encounter three main challenges in our design:

- For SFC placement, to optimize the performance of MSFCs, we are challenged to carefully analyze the cost of inter-VM packet transfer via a virtual switch (vSwitch) [3]. Moreover, we are challenged to design a performance-aware placement algorithm to consolidate appropriate elements together.
- For SFC elasticity control, careless placing the scaled out replica may introduce additional packet transfer between VMs and frequent state synchronization among different replicas of the element, which may degrade the performance significantly (possibly up to tens of *ms* [11]). We are challenged to avoid performance degradation.
- For runtime SFC management, when consolidating multiple elements on the same CPU core, we need to ensure fairness when scheduling CPU resources among elements. However, traditional approach [15] requires *manual* configuration of element priorities, which is time-consuming and lacks scalability. We are challenged to design an automatic scheduler.

To address the above challenges, as shown in Fig. 1, we design the CoCo controller for performance-aware consolidation placement (Optimized Placer) and elasticity control (Individual Scaler), and the CoCo Infrastructure that supports automatic resource scheduling. In *Optimized Placer*, CoCo models the performance cost of inter-VM packet transfer and proposes a 0-1 Quadratic Programming-based placement algorithm. In *Individual Scaler*, CoCo proposes an innovative *push-aside element scaling up* strategy as well as a greedy scaling out method for efficient element scaling. Finally, we design an *Automatic Scheduler* in the CoCo infrastructure that schedules CPU resources based on the processing speed of each element.

In this paper, we make the following contributions:

- We introduce the problem of which elements to consolidate and model the performance cost of inter-VM packet transfer. We then design an Optimized Placer in CoCo controller and propose a performance-aware placement algorithm to achieve optimal performance of MSFCs. (Section II)
- We design an Individual Scaler in CoCo controller for individual scaling of elements. We propose an innovative push-aside scaling up strategy as well as a greedy scaling out method to alleviate the hot spot with little performance and resource overhead. (Section III)
- We design a runtime CPU Automatic scheduler in CoCo infrastructure to automatically ensure fairness between multiple elements on the same CPU core with respect to their different processing speed. (Section IV)
- We evaluate the effectiveness of CoCo framework. Evaluation results show that CoCo could improve both performance and resource efficiency. (Section V)

II. Performance-aware MSFC Placement

In this section, we present the CoCo elements placement algorithm inside Optimized Scaler of CoCo Controller for the initial deployment of an MSFC. We have the following goal in mind in our design: *We prefer to consolidate adjacent elements in an MSFC on the same VM and place the MSFC compactly to reduce inter-VM packet transfer cost.*

In the following, we first analyze the one-hop inter-VM packet transfer cost due to vSwitch-based forwarding. We then find the relationship between CPU utilization and processing speed for an element. These two analyses serve as the fundamentals of placement algorithm of MSFC, which usually contains multiple hops and multiple elements.

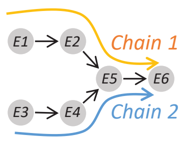
A. Packet Transfer Cost Analysis

In NFV implementation, usually elements are implemented as VMs separately with dedicated CPU cores [28]. To simplify the resource constraint analysis, we assume that each VM is implemented on one CPU core, which could easily be extended to situations where a VM is allocated with multiple CPU cores (Section VII). When packets are consolidated on the same VM with Docker Container [19], intra-VM packet transferring is simple. With shared memory technique provided by Docker, we can directly deliver the pointer on memory of packet from one element to another with negligible latency (about 3 μs under our implementation). However, when packets are transferred between VMs, they must go through four steps, as shown in Fig. 1. First, packets are copied from memory to the virtual NIC (vNIC) on the source VM (Step ①). Next vNIC transfers packets to vSwitch (Step ②). And then packets are delivered reversely from vSwitch to the vNIC of destination VM (Step ③) and finally from vNIC to memory (Step ④). The total transfer delay (about 1 *ms* in our evaluation) degrades the performance of MSFC significantly.

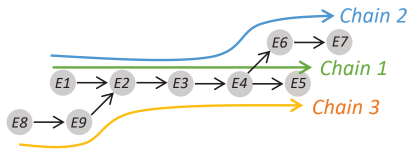
We use *Delayed Bytes (DB)* to represent the packet transfer cost. Theoretically, DB is constrained by the minimum of element throughput, memory copy rate (Step ① and ④), and packet transmission rate (Step ② and ③). However, memory copy rate and packet transmission rate (~10 *GB/s* according to [26]) are much greater than the SFC throughput (99% are <1 *GB/s* in datacenters [22]). Thus DB is directly constrained by the throughput between elements. We denote the throughput as Θ and the additional four-step transfer delay as t_d. Thus

$$DB = \Theta \cdot t_d \tag{1}$$

In the placement analysis, we will use the total sum of DB as our optimization target of performance overhead.



(a) Topology 1



(b) Topology 2

Fig. 2. Two Examples of Processing Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

B. Resource Analysis

Before globally optimizing the total sum of DB, we need to analyze and find the constraints on CPU resource utilization. For a certain element, as the throughput increases, it will consume more CPU resources to process. Thus for each type of element i, we can measure a respective one-to-one mapping function between CPU utilization r and processing speed v:

$$r_i = \phi_i(v_i) \text{ and } v_i = \phi_i^{-1}(r_i) \tag{2}$$

In implementation, network administrators can measure the mapping function for each type of element in advance, which will be introduced in Section V-A.

Docker-based consolidation technique is lightweight and takes few resources [28]. Thus, we can directly add up respective CPU utilizations of elements to estimate the total CPU utilization. Also note that Eq. 2 is an upper bound estimation for CPU utilization given throughput. When several elements are consolidated together, the alternate scheduling mechanism by reusing the idle time caused by interrupts [26] can enable a higher total throughput.

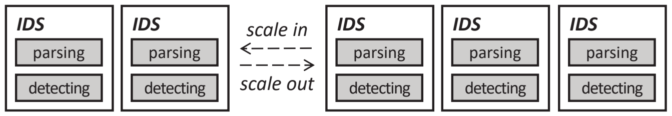
C. MSFC Placement Algorithm

We first abstract the packet processing in MSFCs as a directed acyclic processing graph, denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each node $k \in \mathcal{V}$ represents an element and each edge in $\mathcal{E}$ represents a hop between elements in an MSFC. $k \in \mathcal{I}$ represents the VMs, i.e. CPU cores. Service chains, denoted as $\mathcal{C}$, are defined by tenants. Fig. 2 shows two examples of processing graph. There are two chains and six elements in Fig. 2(a). Chain 1 is E1⇒E2⇒E5⇒E6 and Chain 2 is E3⇒E4⇒E5⇒E6. To consolidate compactly, we assume that the processing speed of each elements on the same chain matches the throughput of the entire chain at initial placement. We denote the throughput of chain j as Θ_j. Conservatively, network administrators can estimate Θ_j with its required bandwidth according to Service Level Agreement [17].

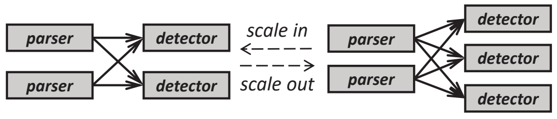
$\alpha_i^j \in \{0, 1\}$ indicates whether element i is on chain j. π_i^j represents the upstream element of element i on chain j, which can be realized with a doubly linked list. To ensure robustness, when i is the first element on chain j, we set $\pi_i^j = i$. When $\alpha_i^j = 0$, we set $\pi_i^j = 0$.

CoCo applies a 0-1 Integer Programming (0-1 IP) algorithm to minimize the inter-VM overhead. $x_{i,k}$ is a binary indicator of whether placing element i onto VM k. For chain j, we analyze the performance overhead between each element i and its upstream element π_i^j on it. From Eq. 1, the hop from π_i^j to i will incur an inter-VM cost of DB_j if and only if they are not placed on the same VM, i.e.

$$x_{i,k} x_{\pi_i^j,k} = 0, \ \forall k \in \mathcal{I}$$



(a) Monolithic scaling of IDS



(b) Individual scaling of modularized IDS

Fig. 3. Individually scalability

Thus we can use $\left(1 - \sum_{k \in \mathcal{I}} x_{i,k} x_{\pi_i^j,k}\right) \in \{0, 1\}$ to indicate whether element π_i^j and i are consolidated together. Then we add up DB of all inter-VM hops in chain j and further add up DB of different chains as our objective function. CoCo aims at minimizing the total inter-VM cost to improve performance:

$$\min \sum_{j \in \mathcal{C}} \sum_{i \in \mathcal{V}} \alpha_i^j DB_j \left(1 - \sum_{k \in \mathcal{I}} x_{i,k} x_{\pi_i^j,k}\right) \tag{3}$$

Meanwhile, the following constraints should be satisfied:
(1) $x_{i,k} \in \{0, 1\}, \ \forall i \in \mathcal{V}, \ k \in \mathcal{I}$
//An element is either consolidated on VM k or not.
(2) $\sum_{k \in \mathcal{I}} x_{i,k} = 1, \ \forall i \in \mathcal{V}$
//An element can only be placed onto one VM.
(3) $\sum_{i \in \mathcal{V}} \left[x_{i,k} \cdot \phi_i \left(\sum_{j \in \mathcal{C}} \alpha_i^j \Theta_j\right)\right] \leqslant 1, \ \forall k \in \mathcal{I}$
//Each CPU core cannot be overloaded at initial placement.

Note that if the estimated through of element i_0 is so high that it cannot be placed on one CPU core, i.e.

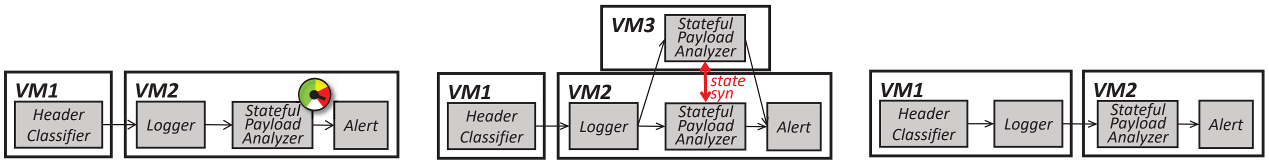
$$\exists i_0 \in \mathcal{V}, \ s.t. \sum_{j \in \mathcal{C}} \alpha_{i_0}^j \Theta_j > \phi_{i_0}^{-1}(1) \tag{4}$$

constraint (2) and (3) may conflict. This is due to the incorrect orchestration between flows and elements. Actually it rarely happens in the real world and never happens in our evaluation. In this situation, scaling out is needed. For overloaded element i_0, if there is only one chain j_0 containing it, CoCo scales out $\lceil \frac{\Theta_{j_0}}{\phi_{i_0}^{-1}(1)} \rceil$ replicas and performs load-balancing among them. If there are several chains containing i_0 (such as E5 in Fig. 2(a)), CoCo scales it out to multiple replicas based on a knapsack algorithm on chain. Then CoCo splits the overlapped chains to different replicas and reconstructs processing graph $\mathcal{G}'$ to ensure that Eq. 4 does not hold for $\forall i \in \mathcal{G}'$.

From the analysis above, we find that Eq. 3 is a 0-1 Quadratic Programming problem and can be solved within limited time and space [10]. By solving the above formulations, we can get the performance-aware optimized placement solution. We evaluate this algorithm in Section V-C.

III. Optimized Individual Scaling

For monolithic NFs, all components must be scaled out at the same time when overloaded, which will take up more resources than needed. Also, continuously synchronizing numerous internal states will introduce significant overhead [11]. After modularization, when traffic increases, only the overloaded elements need to scale out. For example, when the *detector* element of IDS is overloaded, instead of scaling out the whole IDS (Fig. 3(a)), we can only scale out the detector element itself (Fig. 3(b)).



(a) MSFC before scaling (b) Scaling out with traditional method (c) Push-aside scaling up

Fig. 4. Comparison of push-aside scaling up and traditional scaling out

However, deciding where to place the scaled out replica is also important. Careless placement will degrade performance and resource efficiency. In this section, CoCo Individual Scaler provides two innovative scaling strategy including performance-aware push-aside scaling up, and resource-aware greedy scaling out, to efficiently alleviate the overload situation. Push-aside scaling up can avoid the performance degradation caused by additional inter-VM packet transfer. Greedy scaling out can achieve resource efficiency by placing replicas on existing VMs.

A. Push-aside Scaling Up

When scaling out, the traditional NF-level scaling method taken by [11], [27] simply starts a new VM with taking a new CPU core and scales out the overloaded element to the new VM. For example, when the Stateful Payload Analyzer in VM2 is overloaded (Fig. 4(a)), traditional method starts VM3 and copies the element to it (Fig. 4(b)). However, this will introduce additional latency overhead due to inter-VM packet transfer. For example, in Fig. 4(b), a part of packets will suffer 3 inter-VM hops to go through the total MSFC (VM1⇒VM2⇒VM3⇒VM2). Also, frequent state synchronization between replicas will also degrade the performance.

However, when an element in an MSFC is overloaded, the VMs that its upstream or downstream elements placed on may be underloaded. Enabled by the lightweight feature of element, we can re-balance the placement of elements on the two VMs. Thus, the key idea of push-aside scaling up is that the overloaded element can *push* its downstream/upstream element *aside* to its downstream/upstream VM and *scale* itself *up* to alleviate the overload. As for Fig. 4(a), we can *migrate* Logger to VM1 and release its CPU resource (Fig. 4(c)). By allocating the newly released resource to Stateful Payload Analyzer and scaling it up, the overload can be alleviated.

Push-aside scaling up has two advantages. First, compared to the traditional method, it does not create new inter-VM hops, thus there is no additional packet transfer cost. Second, it does not create a new replica but allocates more resources to overloaded element. Thus push-aside scaling up does not suffer the state share and synchronization problems [11].

The algorithm includes the following four steps.

Step 1: Check the practicability. If the estimated throughput $\Theta_{i_0}^{exp}$ for element i_0 is so large that the overload on i_0 cannot be alleviated with one CPU core, i.e. $\Theta_{i_0}^{exp} > \phi_{i_0}^{-1}(1)$, push-aside scaling up will not work. This happens only when an extremely large burst comes. In this situation, algorithm terminates and CoCo goes to greedy scaling out.

Step 2: Find border elements. For element i_0 in chain j_0, CoCo first finds out its upstream and downstream border elements $up\_border_{i_0}^{j_0}$ and $down\_border_{i_0}^{j_0}$. Upstream border element refers to the element that $up\_border_i^j$ and i are placed in the same VM but $up\_border_i^j$ and $\pi^j_{up\_border_i^j}$ are placed separately. With doubly linked list, CoCo goes through elements hop by hop to find out border elements and composes them into a set $\mathcal{B}_{i_0}$. If i_0 is contained by several chains, CoCo checks each chain and composes the results into $\mathcal{B}_{i_0}$.

Step 3: Check whether it can be migrated. After finding out the border elements , CoCo checks whether they can be migrated to the adjacent VM. Suppose both $b_0 \in \mathcal{B}_{i_0}$ and i_0 are on chain j_0. We denote the adjacent VM of b_0 as $k_{b_0}^{adj}$. If

$$\phi_{b_0}(\Theta_{j_0}) + \sum_{i\in\mathcal{V}} x_{i,k_{b_0}^{adj}} \cdot \phi_i \left(\sum_{j\in\mathcal{C}} \alpha_i^j \Theta_j \right) < 1 \qquad (5)$$

which means there is available resource for b_0 on $k_{b_0}^{adj}$, we can migrate b_0 to $k_{b_0}^{adj}$ and release its resource. Similarly, we can check all $b \in \mathcal{B}_{i_0}$ and its respective k_b^{adj}. If none can be migrated, CoCo goes to the greedy scaling out strategy. This happens only when all of the adjacent VMs of i_0 do not have enough resource, which in practice rarely happens. If some of them can be migrated, CoCo composes them into $\mathcal{B}'_{i_0} \subset \mathcal{B}_{i_0}$.

Step 4: Check whether overload can be alleviated. At last, CoCo checks if migrating all elements in $\mathcal{B}'_{i_0}$ will make enough room for i_0 to scale up to alleviate the overload. Otherwise the migration will be useless. CoCo calculates the needed resource $r_{i_0}^* = \phi_{i_0}(\Theta_{i_0}^{exp}) - \phi_{i_0}(\Theta_{i_0}^{cur})$, where $\Theta_{i_0}^{cur}$ is the current processing speed of i_0. The CPU utilization of element $b' \in \mathcal{B}'_{i_0}$ satisfies $r_{b'} = \phi_{b'}\left(\sum_{j\in\mathcal{C}} \alpha_{b'}^j \Theta_j\right)$. If $\sum_{b'\in\mathcal{B}'_{i_0}} r_{b'} < r_{i_0}^*$, which means migration cannot release enough resource, the algorithm terminates and CoCo goes to greedy scaling out. Else, push-aside scaling up can be applied. Also, aware that migrating elements from one VM to another has performance cost and controller overhead [11], CoCo tries to minimizes the number of elements to migrate. CoCo finds a subset $\mathcal{B}''_{i_0} \subset \mathcal{B}'_{i_0}$ with minimal number of elements that satisfies $\sum_{b''\in\mathcal{B}''_{i_0}} r_{b''} \geqslant r^*$.

Moreover, to avoid potential frequently scaling up among elements, network administrators can set a timeout between each time of scaling up. In this way, we can alleviate the overload with minimum elements to migrate.

B. Greedy Scaling Out

If the overloaded element can push none of border elements aside to other VMs, Individual Scaler have to scale it out to somewhere else. In this situation, performance degradation caused by scaling out is unavoidable. Even so, we can still save resource by placing the new replica to an already working VM.

CoCo decides the VM to place the replica based on a greedy algorithm. First, it calculates the remained resource of each

VM and sorts them in increasing order. Next, CoCo greedily compares it with the needed resource $r^*_{i_0}$. When the remained resource of any VM, i.e. CPU core, is larger than $r^*_{i_0}$, CoCo places the replica there. If none of the VMs have available CPU resource, CoCo will call up new VMs and specify new CPU cores, just as the traditional method does.

IV. Automatic Consolidation Scheduling

When consolidating several elements onto one CPU core, CoCo uses one Docker [19] for each element. At this time, we need a scheduling algorithm to enable fair resource allocation. However, as we discussed above, traditional rate-proportional scheduling methods [25] and priority-aware scheduling methods [15] are not scalable due to massive manual configurations. Thus, we design a novel scheduling algorithm to match processing speed of elements with its throughput to automatically achieve both fairness and efficiency. Here, we take CPU resource as the allocation variable since CPU is more likely to become a bottleneck resource than memory [15], especially when elements are densely consolidated in MSFC.

CoCo takes an incrementally adaptive adjustment scheduling algorithm. The algorithm tries to match the processing speed of each element with its packet arrival rate. It incrementally adjusts the CPU utilization of the next scheduling period based on the statistics of current scheduling period. The detailed algorithm is introduced below.

In consolidation, CPU resources are scheduled among elements by periodically allocating *time slices* with CGroup [2]. Suppose scheduling period is T. For element i on a VM, we can get its CPU utilization proportion r_i by counting the number of time slices. Note that r_i is a proportion thus $\sum_i r_i = 1$. Also, we can get current buffer size B_i and last time buffer size B'_i. From Eq. 2, we can know the processing speed v_i satisfies $v_i = \phi_i^{-1}(r_i)$. We denote B^*_i, v^*_i and r^*_i as the *predicted* buffer size, processing speed and CPU utilization at the next scheduling period.

Our scheduling algorithm is based on the matching principle: *For all elements, their buffer variations should be proportional to respective processing speeds*, i.e.

$$\frac{B^*_i - B_i}{v^*_i T} = C,\ \forall i \in \{1, \cdots, n\} \tag{6}$$

By modeling in this way, we try to ensure fairness among elements and effectively allocate resources. The key idea is to match the processing speed with its respective packet arrival rates. In this way, the element with a lower processing speed and smaller flows can also attain an appropriate proportion of CPU. However, when several elements are consolidated together, downstream element may directly read the packet that are already loaded into memory by its upstream element. Thus we cannot get the actual packet arrival rate by simply measuring at the last switch. Naively adding statistics measuring module on the top of Docker will introduce unnecessary overhead. Instead, we can infer the arrival rate v_{ai} from the variation of buffer size, i.e.

$$v_{ai} = \frac{B_i - B'_i}{T} + v_i \tag{7}$$

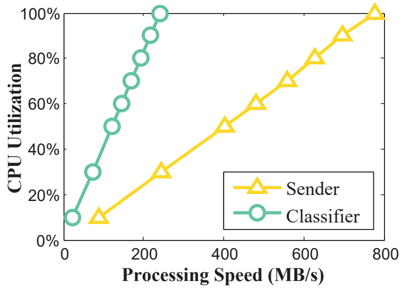


Fig. 5. Throughput-CPU Utilization Mapping

As traffic usually does not vary sharply [8], we can assume that in a scheduling period (usually at millisecond level), the packet arrival rate keeps invariant, i.e. $v_{ai} = v^*_{ai}$. By substituting B^*_i in Eq. 6 with Eq. 7, we can get the CPU proportion for element i in the next scheduling period:

$$r^*_i = \phi_i(v^*_i) = \phi_i\left(\frac{\frac{B_i - B'_i}{T} + v_i}{C+1}\right) \tag{8}$$

The sum of CPU utilization needs to be normalized, thus C subjects to

$$\sum_i \phi_i\left(\frac{\frac{B_i - B'_i}{T} + v_i}{C+1}\right) = 1 \tag{9}$$

Although we cannot get an explicit expression of C, we can first randomly specify an initial value for C and then normalize r^*_i. Comparing to the millisecond level scheduling period, the solving time in this way is negligible.

Another important function of consolidation scheduler is to tell the individual scaler when to scale out. A direct indicator is the buffer size of an element. If buffer is overflowed, packet loss incurs and the element is definitely overloaded. At this time, scheduler can do nothing but execute the scaling up or scaling out methods, as introduced in Section III.

V. Preliminary Evaluation

In this section, we first introduce our methods on measuring throughput-CPU utilization mapping function. Then we build CoCo with Docker [19] to consolidate elements on VMs, and enable inter-VM packet forwarding with Open vSwitch (OVS) [3]. We take the low-level dynamical element migration mechanism from OpenNFand evaluate the effectiveness of high-level *push-aside scaling up* strategy compared to OpenNF [11]. Finally, we evaluate the CoCo performance-aware MSFC placement algorithm based on our simulations.

We evaluate CoCo based on a testbed with one server equipped with two Intel® Xeon® E5-2690 v2 CPUs (3.00GHz, 8 physical cores), 256G RAM, and two 10G NICs. The server runs Linux kernel 4.4.0-31.

A. Throughput-CPU Utilization Mapping

To measure the throughput-CPU utilization mapping, we constrain the available CPU utilization for the element by fixing the `cpu-period` and changing the `cpu-quota` parameter in Docker. We use a packet sender to test the maximum throughput under the current limited CPU proportion. With this method, administrators can get the mapping function $\phi(v)$.

We measure two types of element with different complexity. Packet Sender represents elements with simple processing

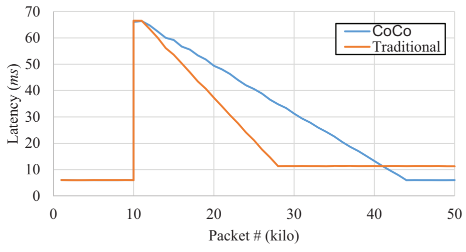


Fig. 6. Real-time Latency per Packet

logic. IP Address-based Header Classifier contains 100 rules and represents relatively complicated elements. The mapping functions are shown in Fig. 5. Surprisingly, a strong linearity correlation can be observed. We present the linear regressions of the two mapping functions:

- **Sender:** $r = -0.022 + 0.0013 \times v,\ R^2 = 0.9997$
- **Classifier:** $r = 0.00048 + 0.0042 \times v,\ R^2 = 0.9999997$

$r \in [0, 1]$ is the CPU utilization and v is the processing speed in *MB/s*. R^2 is a measure of goodness of fit with a value of 1 denoting a perfect fit. Thus in practice, we can further simplify the solving procedure by substituting $\phi(v)$ and $\phi^{-1}(r)$ with their linear approximations.

B. Push-aside Scaling Up

To evaluate the performance of push-aside scaling up, we use the MSFC in Fig. 4(a). At first, the throughput of MSFC is 100 kpps, with each packet 512 B. At the 10k-th packet, we increase the traffic to 150 kpps, which causes the Stateful Payload Analyzer overloaded. The traditional method taken by OpenNF [11] naively scales out by copying Stateful Payload Analyzer to a newly started VM, as shown in Fig. 4(b). In contrast, CoCo migrates Logger to VM1 and allocate the released resource to Stateful Payload Analyzer.

For performance, the comparison of two methods on real-time latency of each packet is shown in Fig. 6. The latency at 10k-th packet increases sharply due to the element migration. Traditional method converges a little faster because it consumes more resources and has a higher processing speed. However, CoCo has a lower converged latency of 6 *ms* compared to 11 *ms* of the traditional method. The improvement is achieved by reducing the additional packet transfer and state synchronization latency. For resource efficiency, with traditional method, the scaled MSFC is allocated with more resources (3 VMs in total). In contrast, 2 VMs are enough with CoCo in this situation (Fig. 4(c)) by push-aside scaling up. CoCo achieves a higher resource efficiency by 1.5×.

C. Performance-aware MSFC Placement

As for evaluating the performance of placement algorithm, we evaluate the total *DB* in the processing graph. We use an Optimization Toolbox [4] to solve the 0-1 Quadratic Programming. For large-scale and complicated topologies, quadratic programming can be efficiently solved with some dedicated commercial solvers such as IBM® CPLEX [5]. We use two topologies shown in Fig. 2 and implement all elements as classifiers described in Fig. 5. We randomly select flows from the LBNL/ICSI enterprise trace [1] to different chains

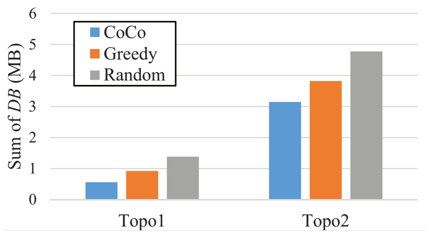


Fig. 7. Sum of Delayed Bytes

TABLE I
PLACEMENT FAILURE RATE (OF 1000 TESTS)

	CoCo	Greedy	Random
Topo1	114 (11.4% fails)	155 (15.5% fails)	214 (21.4% fails)
Topo2	58 (5.8% fails)	65 (6.5% fails)	78 (7.8% fails)

and repeat the experiment for 1000 times to eliminate the randomness. We try to place Topology 1 (Fig. 2(a)) on 2 VMs and Topology 2 (Fig. 2(b)) on 4 VMs.

Since that there is no ready-made solution on *which elements to consolidate*, we compare CoCo with two strawman solutions, including a *greedy* mechanism, which greedily places elements onto VMs chain by chain, and a *random* mechanism, which randomly selects available VMs to place elements.

An important feature of CoCo placement is performance-aware, which is evaluated as the sum of *DB*. The total *DB* in processing graph of successful placements with different strategies is shown in Fig. 7. For Topo1, CoCo reduces the total *DB* by 2.46× compared to random strategy and by 1.64× compared to greedy strategy. For Topo2, even the lengths of chains increase and more inter-VM packet transfers are unavoidable, CoCo still outperforms the random strategy by 1.52× and greedy strategy by 1.21×.

Another feature of CoCo is resource efficient by compact placement, which can be interpreted as: *Given limited CPU cores, i.e. VMs, for different traffic, CoCo has a higher probability to place all of them on successfully.* As shown in Table I, when placing Topo1, CoCo improves the failure rate by 1.88× compared to random strategy and 1.36× compared to greedy strategy of 1000 tests. Note that the placement of Topo1 ($\frac{6 \text{ elements}}{2 \text{ VMs}} = 3$) is tighter than Topo2 ($\frac{9 \text{ elements}}{4 \text{ VMs}} = 2.25$), thus placing Topo1 has a higher failure rate than Topo2. Even so, CoCo improves the failure rate from 7.8% (random) and 6.5% (greedy) to 5.8% for Topo2.

VI. RELATED WORK

In this section, we summarize some related work and compare them with CoCo.

Modularization. Click [20] proposed the idea of modularization and applies it to routers. Recently, Slick [7] and OpenBox [9] were proposed to detailedly discuss modularized NFs and decouple control plane and data plane of modularized NFs for easy management. Besides, OpenBox focused on merging elements to shorten the processing path length. However, above works mainly focus on orchestration-level module management and are orthogonal to our optimizations on performance-aware placement and dynamically scaling.

Consolidation. CoMb [24] designed a detailed mechanism to consolidate middleboxes together to reduce provisioning cost. Furthermore, Flurries [28] and NFVnice [15] were proposed to share CPU cores among different NFs with the technique of Docker Container [19]. By modifying Linux scheduling methods, they achieved almost no loss in NF sharing. However, they operated on monolithic NF level and did not consider the problem of which elements (NFs) to consolidate. However, their development details and infrastructure designs could complement our work as the low-level implementation.

Placement. There are a lot of research on NF placement in NFV, such as [17], [18], [23], which focus on the trade-off between traffic load, QoS, forwarding latencies, and link capacities. However, they focused on the placement at NF-level instead of element-level, and thus did not benefit from the lightweight feature of NF modules. Slick [7] considered placement at element-level. However, all of the work above addressed how to place *middleboxes (vNFs)* onto different *servers* considering complicated and limited physical links. In contrast, CoCo pays attention to the placement problem of *how to consolidate elements* onto different *VMs*, i.e. *cores*.

VII. Discussions

In this section, we discuss how to extend CoCo and highlight several open issues as future directions.

Multi-core Placement Analysis. For simplicity, CoCo assumes that each VM is allocated with one CPU core when optimizing placement to satisfy the general applications. In some cases, when tenants allocate multiple CPU cores to a VM, CoCo can be easily extended by considering the resource constraint of multiple CPU cores instead of a single core.

Intra-core Analysis. CoCo analyzes the inter-core cost caused by vSwitch-based packet transfer. As our future work, by designing cache replacement policies, we may reduce the miss rate of Layer 1 and 2 Cache and further reduce repeatedly packet loading from memory to cache. Moreover, more designs are needed to ensure isolation between consolidated elements. However, those analyses are infrastructure-dependent and differs on various types of CPU, which is beyond our scope. CoCo can be easily extended to analyze intra-core situations on a certain type of CPU.

VIII. Conclusion

This paper presents CoCo, a high performance and efficient resource management framework, for providing compact and optimized element consolidation in MSFC. CoCo addresses the problem of which elements to consolidate in the first place and provides a performance-aware placement algorithm based on 0-1 Quadratic Programming. CoCo also innovatively proposes a push-aside scaling up strategy to avoid performance degradation in scaling. CoCo further designs an automatic CPU scheduler aware of the difference of processing speed between elements. Our preliminary evaluation results show that CoCo could reduce packet transfer cost by up to 2.46× and improve performance at scaling by 45.5% with more efficient resource utilization.

IX. Acknowledgement

We thank anonymous ICC reviewers for their valuable comments. We also thank Zhilong Zheng and Heng Yu from Tsinghua University for providing suggestions on system design and implementation. This work is supported by National Key Research and Development Plan of China (2017YFB0801701), the National Natural Science Foundation of China (No. 61472213) and National Research Foundation of Korea (NRF-2014K1A1A2064649). Jun Bi is the corresponding author.

References

[1] Lbnl/icsi enterprise tracing project. http://www.icir.org/enterprise-tracing, 2005.
[2] Linux container: https://linuxcontainers.org/, 2007.
[3] Open vswitch: http://openvswitch.org/, 2009.
[4] Matlab optimization toolbox: https://mathworks.com/products/ optimization.html, 2015.
[5] Cplex optimizer: https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer, 2018.
[6] James W Anderson, Ryan Braud, et al. xomb: extensible open middleboxes with commodity servers. In *ANCS*. ACM, 2012.
[7] Bilal Anwer, Theophilus Benson, et al. Programming slick network functions. In *SOSR*. ACM, 2015.
[8] Theophilus Benson, Ashok Anand, et al. Microte: Fine grained traffic engineering for data centers. In *CoNEXT*. ACM, 2011.
[9] Anat Bremler-Barr, Yotam Harchol, et al. Openbox: a software-defined framework for developing, deploying, and managing network functions. In *SIGCOMM*. ACM, 2016.
[10] Alberto Caprara. Constrained 01 quadratic programming: Basic approaches and extensions. *European Journal of Operational Research*, 187(3):1494 – 1503, 2008.
[11] Aaron Gember-Jacobson, Raajay Viswanathan, et al. Opennf: Enabling innovation in network function control. In *SIGCOMM*. ACM, 2014.
[12] R Guerzoni et al. Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper. In *SDN and OpenFlow World Congress*, 2012.
[13] Jinho Hwang, KK Ramakrishnan, et al. Netvm: high performance and flexible networking using virtualization on commodity platforms. *Transactions on Network and Service Management*, 12(1):34–47, 2015.
[14] Hamzeh Khazaei, Cornel Barna, et al. Efficiency analysis of provisioning microservices. In *CloudCom*. IEEE, 2016.
[15] Sameer G Kulkarni, Wei Zhang, et al. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *SIGCOMM*. ACM, 2017.
[16] S Kumar, M Tufail, S Majee, et al. Service function chaining use cases in data centers. *IETF SFC WG*, 2015.
[17] Marcelo C Luizelli, Leonardo R Bays, et al. Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions. In *IM*. IEEE, 2015.
[18] Sevil Mehraghdam, Matthias Keller, et al. Specifying and placing chains of virtual network functions. In *CloudNet*. IEEE, 2014.
[19] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.
[20] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *SOSP*, 1999.
[21] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015.
[22] Arjun Roy, Hongyi Zeng, et al. Inside the social network's (datacenter) network. In *SIGCOMM*. ACM, 2015.
[23] Marco Savi, Massimo Tornatore, et al. Impact of processing costs on service chain placement in network functions virtualization. In *NFV-SDN*. IEEE, 2015.
[24] Vyas Sekar, Norbert Egi, et al. Design and implementation of a consolidated middlebox architecture. In *NSDI*. USENIX, 2012.
[25] Dimitrios Stiliadis and Anujan Varma. Rate-proportional servers: a design methodology for fair queueing algorithms. *IEEE/ACM Transactions on networking*, 6(2):164–174, 1998.
[26] Andrew S Tanenbaum. *Modern operating system*. Pearson Education, Inc, 2009.
[27] Yang Wang, Gaogang Xie, et al. Transparent flow migration for nfv. In *ICNP*. IEEE, 2016.
[28] Wei Zhang, Jinho Hwang, et al. Flurries: Countless fine-grained nfs for flexible per-flow customization. In *CoNEXT*. ACM, 2016.