Adapt: Adaptive Service Chain Scheduling with Stateless Migration and NF Consolidation

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract. This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract. This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract. This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract. This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

The trend of Network Function Virtualization (NFV) [] aims to replace hardware network functions (NFs) with virtualized NF (VNF) instances, that run as virtual machines [] or containers [] inside commodity servers. To deploy a service chain consisting of multiple consecutive NFs for complex network service, tenants usually deploy it on to existing NFV platforms managing multiple servers. The predominant way is to deploy a service chain inside a same server, and chaining the deployed VNF instances with high-performance software SDN switches.

Nowadays, servers in cloud environment are provisioned with redundant CPU cores. To make better use of the server resources, differnt service chains ordered by different cloud tenants are usually packed into the same server []. (Probably emphasize what VNF instaces will experience inside the same server). Therefore, timely and correct scheduling of different VNF instances occupied by different service chains is critical to the overall performance of the entire NFV system. However, the dominant scheduling strategies all possess their own weaknesses when being used to schedule VNF instances inside the same server.

One of the most common scheduling strategy can be refered to as share-nothing strategy (SN strategy). The SN strategy mandates that a single VNF instance is bond to a fixed number of cores and share no CPU resources with other VNF instances. Since the VNF instance owns all the available CPU resources, its packet processing will not be interrupted by OS-level scheduling, resulting in stable packet processing delay and high single-instance processing throughput. The downside of this strategy is that it may waste important CPU resources when multiple VNF instances are composed into a service chain. Since the input traffic flows through the instances in a sequential order, the overall throughput of the chain will be limited by the bottleneck, i.e. the VNF instance that uses up all the available CPU resources. The rest of the VNF instances on the chain still have spare CPU resources, but they can't process additional input traffic due to the presence of the bottleneck.

Another important scheduling strategy is refered to as consolidated strategy (C strategy). The C strategy may lauch multiple VNF instances on one CPU core. Since a single VNF instance shares the same CPU core with other VNF instances, its processing will be constantly interrupted by the OS scheduler, resutling in prolonged and undeterministic packet processing delay. On contrary to the previous strategy, this consolidated scheduling enjoys a better CPU utilization, as the scheduler may launch new VNF instances on under-utilized CPU corese. In the long run, this strategy may enjoys a better system-level throughput.

Neither of the two strategies are optimal for today's NFV system design. On one hand, while SN strategy ensures stable and low packet processing latency, it's ineffectiveness in handling under-utilized cpu cores may lead to decreased system throughput. On the other hand, while the C strategy enjoys better CPU utilization and has better system throughput, its packet processing latency is prolonged and unstable, and has a negative impact on the QoS of many latency-sensitive applications. Apparently, a good scheduling strategy should take the best from both strategies, achiving stable and low packet processing latencies while ensuring good CPU utilization and better system throughput.

In this paper, we propose a new NFV scheudling system, called AdaptNF. What distinguish AdaptNF from exisitng scheduling systems are two folds.

First, AdaptNF uses a hybrid scheduling strategy that adapts to the current available CPU core resources. AdaptNF relies on SN strategy to allocate VNF instances under moderate workload for better packet processing latencies, and intelligently switch to the consolidated strategy under high workload for better system throughput. In this paper, we focus on designing the scheudling strategy for running NFV service chains on a single server with redundant CPU cores. The overall scheduling strategy can be described as an adaptive

strategy, which is combine SN strategy with C strategy, and switched between the two strategy when necessary. The overall goal is to achieve good packet processing latency, while retaining maximum CPU utilization.

Second, AdaptNF utilizes a special migration approach called stateless migration to balance the workload among different VNF instances of the same type. The stateless migration is fully compatible with existing SDN-NFV platform without adding any additional patches to the existing SDN-NFV architecture. It achieves migration of a service chain from one instance to another by rule-table updating. We update the rule-table by dividing the original service chain rule into a series of sub-rules, that collectively forwarding existing flows to the original instance, and using a new rule to direct new flows to the old instance. We have designed a K-prefix covering algorithm to generate a set of sub-rules that have minimum number of covered flows. In this way, we strike a balance between the number of flow rules generated and the overall flow migration time.

We have implemented AdaptNF on a server system equipped with OpenNetVM and OpenVSwitch. We leverage OVS as the SDN dataplane and uses the containers manaaged by OpenNetVM to run different VNF instances. Our evaluation results show that (suplement evaluation result)

In summary, we have made the following contributions in this paper:

- A new stateless migration approach that integrates well with existing SDN-NFV architecture and strikes a balance between number of rules generated and migration completion time.
- A new adaptive scheduling method that combines service chain consolidation inside the same instance with NF consolidation inside the same core.
- A two-stage heuristic algorithm that balance workload among different VNF instances.
- Testbed implementation and extensive evaluation.

II. Backgroudn and Motivation

In this section, we highlight the recent technical solutions in the NFV that motivated the design of ElasticNF. Through summarized the defects of the current technology and described our motivations.

Stateless migration: An important benefit of the NFV vision is elastic scaling the ability to increase or decrease the number of NF instances currently devoted to a particular NF, in response to changes in offered load. However, the difficulty of elastic scaling arises in how to handle NF instances state appropriately. In typical case, NFV controller manages the NF states when initiating migration, placement, etc. When flows need to migrate from old NF to a new one, the NFV controller coordinates the migration of relevant state from the old to new NF instance; once migration completes, the flows will be route to the new NF. This approach can lead to long pause times during which both old and new NF instances stop processing packets while state is migrated, for example, the test [24] incurs a pause time of 490 ms when migrating only 1,500 flows despite extensive optimization to the process. Such a result is unacceptable for some delay-sensitive flows. If the delay overhead of the state migration is so large, is it possible to prevent the state of the flow make a migration and still let the flow stay in the original NF up to end? And waited for the end of the flow can also be routed to a new NF to achieve the goal of migrating traffic eventually. Which proved to be a feasible solution, we call that stateless migration.

Hybrid scheduling: The NFs deployed in the industry vary are diverse in application, complexity, and processing requirements. The ETSI standard indicates that NF has different processing and performance requirements. Therefore, most types of NF show that the CPU demand and the processing delay of each packet are different. How to balance the delay and throughput so as to obtain the best NFV system performance? We conducted some practical experiments to explore the key factors that affect the performance of the NFV platform.

In Figure 1, there are two service chains. Service chain 1 passes through three NFs. The conditions are Firewall (NF 1), Payload Scan (NF 2), and Getaway (NF 4). Service chain 2 replace Payload Scan (NF 2) with Encrypt (NF 3). Therefore, NF 1 and NF 4 are simultaneously part of service chain 1 and service chain 2. In the first stage, we test the default initial scheduler, 4 cores used in total. Pktgen generates 64-byte packets at line rate, equally splitting them between two flows that are assigned to chain-1 and chain-2. Table 1 shows the results of the operation. NF1 allocates the same resources to the two service chains, each chain occupies 50%. However, for service chain 2, NF3 discards a majority of the packets processed. In this way, the packets processed by NF1 are waiting in the queue for NF3 processing, but the speed of the packet reaches faster than processing rate of NF3, causing serious packet loss. NF 3 has become the bottleneck of the chain. Comparing the throughput of the two service chains and the NFs CPU allocation ratio, it can be seen that the throughput of the low-cost service chain 1 is 80% higher than the high-cost service chain 2, whereas the core dedicated by NF2 is only between 50% and 55%, 95%-99% of the core which NF 3 is located (the remaining small part of the resources are used for internal processing of the core). The difference in throughput between the two service chains is closely related to NF 2 and NF 3. The high-cost service chain 2 lacks resources, while the low-cost service chain 1 resources are wasted. The resource allocation strategy between the two service chains leads to low throughput performance.

In order to further explored how to allocate resources to achieve the optimal performance of the platform, or gave us some inspirations. We continued to explore the collected mainstream solutions: consolidated strategy and dedicated strategy.

Dedicated strategy: Earlier works, openNF [**?**], CoMb [**?**] etc, used the schedule type that one NF runs on dedicated core as a unit. It is simple and efficient with dedicated core, and provide stable processing delay of each packet.

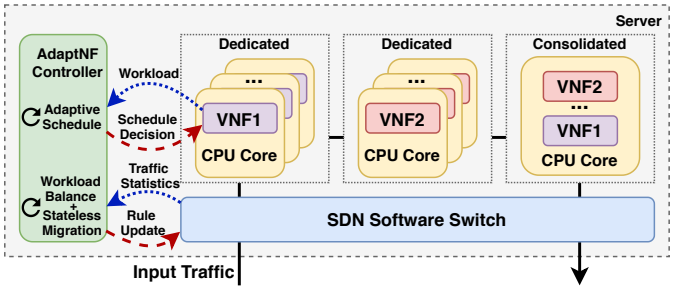

Figure 1. System overview of Adapt.

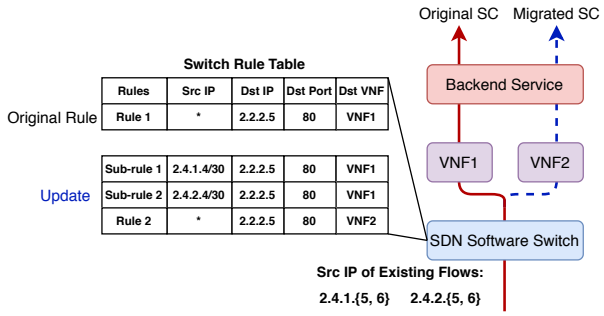

Figure 2. An Example Rule Update Using the Proposed Stateless Migration Approach.

III. Overview

In this section, we give an overview of AdaptNF. Fig. 1 illustrates the architecture of AdaptNF, which largely follows the typical architecture of a NFV management system. AdaptNF concentrates on scheduling service chains within a single server. The primary component of AdaptNF is a controller, which manages the lifecycles of service chains and the associated VNF instances.

AdaptNF controller has two basic functionalities:

A. Lifecycle Management for Service Chains

Overall, the AdpatNF controller serves as a SDN controller which closely manages how service chain traffic are routed inside the server. It also serves as a local NFV agents that is responsible for launching and removing new VNF instances to deploy the service chains.

The controller takes service chain creation command from a high-level NFV management and deploys service chains inside the server. Depending on the current workload condition, the controller selects instances with the smallest workload and installs service chain rules into the SDN software switch to direct the input traffic matching the rule across different instances along the service chain. When NFV management system issues a service chain deletion request, the controller deletes the service chain from the server by removing the rules.

B. Adaptive Schedule of Exisitng VNF Instances

The AdaptNF controller is also responsible for balancing the workload of different VNF instances and dynamically scale-out/in by adding/removing necessary VNF instances. To achieve this, the AdaptNF constantly monitors the CPU usage of each VNF instance, as well as the input traffic statistics of each VNF instance collected from the SDN software switch.

The AdaptNF makes scheduling decision periodically. At the start of a decision making interval to generate new service chain scheduling decision, the AdaptNF uses the adaptive scheduling method discussed in Sec. VI to generate a detailed scheduling decision. In particular, the AdaptNF determines how many new instances need to be launched for scaling the current system out. Adaptavie schedule method generates two kinds of possible VNF instances, which are referred to as share-nothing instances and consolidated instances. When the number of available CPU cores are redundant, adaptive schedule prefers using share-nothing instances, for improved packet processing latency. When the available CPU resources becomes scarce, adaptive schedule preferse consolidating multiple VNF instance on to the same core, for improved overall throughput.

After generating the scheduling decision, the AdaptNF launches the corresponding VNF instances inside the server. All the VNF instances are launched as Docker containers with pre-built NF images, so that the launching only takes a minimal amount of time. Then the AdaptNF executes a workload balance algorithm discussed in Sec. V to decide how to balance the service chain workload across different number of VNF instances. Using the algorithm output, the AdaptNF then uses stateless migration (Sec. IV) to rebalance the workload on different instances by updating the rule table and constantly monitoring the expiration of generated sub-rules.

When all the stateless migration completes, AdaptNF enters the next scheduling interval.

IV. Stateless Migration

Stateless migration in ElasticNF consists of the following steps.

First, the IP source addresses of all the existing flows matched by the service chain rule is collected as input. The service chain rule is then divided into sub-rules with distinct IP source headers, while the rest of the sub-rule headers remain identical as the original rule.

Second, the original rule is divided into several sub-rules forwarding matched flows to the original instance. We must ensure that all the existing flows are matched by the sub-rules for processing consistency.

If the number of distinct source IP addresses does not exceed a pre-defined value K, the original rule is divided into several exact-match sub-rules, each matching one of the input IP source address. Otherwise, the original rule is divided into K sub-rules (*e.g.* $K = 64$) with distinct IP source prefixes, which are calculated according to three constraints discussed in the following section and match all the input IP addresses.

Third, a new rule is installed with identical match headers as the original rule, forwards flows to the new instance and has the lowest priority among all the sub-rules.

Finally, The migration expires when all the generated sub-rules expire and only the new rule remains on the table to direct subsequent flows to the new instance. If some sub-rules persist over a threshold time, those rules are divided again with the second step to accelarate expiration.

Following this approach, we generate Update 2 in Fig. 2 to migrate the service chain containing 4 existing flows from VNF1 to VNF2. The original Rule 1 is divided into 2 sub-rules with distinct IP source headers. Sub-rule 1 with prefix 2.4.1.4/30 matches existing flows 2.4.1.{5,6} and two new flows 2.4.1.{4, 7}. Sub-rule 2 with prefix 2.4.2.4/30 matches the rest of the existing flows and two new flows 2.4.2.{4, 7}. The migration completes when the four existing flows terminate and the four new flows stop arriving [1].

A. Explanation of Migration Steps

We only use IP source address to divide a service chain rule during migration. A standard SDN rule has at most 13 headers to match [], each may contain prefix or wildcard mask. If we do not limit the header used for dividing a rule, it would be difficult to search for correct sub-rules within a reasonable time. IP source header is selected because IP header contains a large address space (2^{32} for IPv4 and 2^{64} for IPv6), which can match enough number of flows to enable fine-grained migration control and improve the workload-balancing performance in Sec. V. Also, IP source address can uniquely characterize an input traffic flow with high probability.

The number of sub-rules generated for a single migration has an fixed upper bound K. In this way, we guarantee that each migration only expands the size of the rule table to an acceptable range, and has limited performance impact on the underlying SDN software switch [].

To match excessive existing flows with limited sub-rules, we add IP prefix to the IP source header, As a consequence, new flows may be matched and directed to the original instance. While this can be tolerated, the migration now completes when existing flows terminate and new flows matched by the sub-rules stop arriving. If new flows keep arriving, the migration completion time may be prolonged to an unacceptable range. We define the IP addresses covered by an IP prefix as the set of IP addresses matched by this prefix. For instance, the prefix 192.168.128.0/24 covers IP addresses 192.168.128.{0, 1, ..., 255}. To control the migration completion time, we add three constaints to the generated IP prefixes.

First, each prefix covers at least one input IP address, which ensures the necessity for every sub-rule. *Second*, the IP addresses covered by different prefixes should be mutually disjoint, ensuring that each sub-rule can expire independently without affecting others. *Finally*, the total number of IP addresses covered by these prefixes is minimized. With this one, the amount of new flows covered by the sub-flows is expected to account for a small portion of the entire new flows, so we can expect a reasonable time before new flows covered by the sub-rules stop arriving.

In summary, this migration method strikes a balance between the total number of generated sub-rules and the overall migration completion time, and is suitable for practical deployment. The only remaining question is how to generate K prefixes to cover excessive input IP addresses under the contraints.

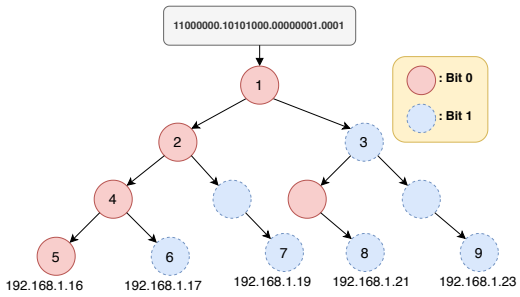

Figure 3. A constructed trie with five IPs, including 192.168.1.{16, 17, 19, 21, 23}. Because the most significant 28 bits are the same, we merge these nodes into one node for simplicity.

B. K-prefix Covering Algorithm

We refer the problem as K-prefix covering problem and a formal definition is given below:

Problem 1. *Given N IP addresses, find out $K(1 \leq K < N)$ optimal IP prefixes to cover the N input IP addresses, under the three constraints discussed in Sec. IV-A.*

This problem can be solved with dynamic programming on a trie constructed from the N input IP addresses.

1) Trie Construction: The first step is to construct an uni-bit trie t from the N input IP addresses with p_{root} as root. Every node in t represents an IP prefix, whose prefix length equals the depth of the node from the root. Every leaf node of t represents one of the N IP addresses used for constructing the trie. Fig. 3 illustrates an example trie constructed with five IP addresses.

Let p be a trie node of t, then the IP prefix of p covers all the input IP addresses represented by p's descendent leaf nodes. And the total number of IP addresses covered by p is 2^{32-d_p}, where d_p is the prefix length of node p. For instance, the IP prefix of node 2 in Fig. 3 is 192.168.1.16/30, which covers the three input IP addresses of the descendent leaf nodes {5,6,7}. The prefix length of node 2 is 30, and the total number of IP addresses covered is 4.

We have proved that the optimal solution of problem 1 is a subset of the trie nodes of t and presented the detailed proof in Appendix A. Therefore, solving K-prefix covering problem on the N IP addresses is equivalent to finding K optimal nodes from the tree rooted at p_{root} to cover all the descendent leaf nodes of p_{root}.

2) DP Algorithm: We investigate the problem of i-prefix covering ($1 \leq i \leq K$) on p's descendent leaf nodes, which is to find i optimal nodes from the sub-tree rooted at p to cover the descendent leaf nodes of p. This problem exihibits optimal substructure.

Let $p.l[i], 1 \leq i \leq K$ records the total number of covered IP addresses of an optimal solution to the i-prefix covering

problem on p'. Depending on the type of p, $p.l[i]$ can be computed from the optimal sub-problems of p's child nodes.

First, If p is a none-leaf node of depth d_p with left child l and right child r, $p.l[i]$ can be computed with Eq. 1:

$$p.l[i] = \begin{cases} 2^{(32-d_p)}, & \text{if } i = 1 \\ \min_{x+y=i}(l.l[x] + r.l[y]), & \text{if } 1 < i <= K \end{cases} \quad (1)$$

When $i = 1$, the solution of the 1-prefix covering on p contains a single node, which is p itself. According to Sec. IV-B1, $p.l[1]$ is $2^{(32-d_p)}$. When $i > 1$, $p.l[i]$ is computed using the optimal solutions of the sub-problems solved on the left and right-subtree. Suppose that x optimal nodes are selected from the left-subtree, and $y = i - x$ optimal nodes are selected from the right-subtree, then $p.l[i]$ will be the minimum number of covered IP addresses among all the (x, y) pairs.

Second: If p is a none-leaf node with only one child, then $p.l[i]$ can be computed with Eq. 2:

$$p.l[i] = \begin{cases} l.l[i] & \text{if p only has left child } l \\ r.l[i] & \text{if p only has right child } r \end{cases}, 1 <= i <= K \quad (2)$$

The solution of i-prefix covering on p always equals that on p's child node, as p and its child share the same descendent leaf nodes.

Finally: If p is a leaf node, $p.l[i]$ can be computed with Eq. 3:

$$p.l[i] = \begin{cases} 1, & \text{if } i = 1 \\ +\infty, & \text{if } 1 < i \leq K \end{cases} \quad (3)$$

When $i > 1$, it is not possible to find a valid solution for a leaf node, so we set $p.l[i]$ to $+\infty$ to invalidate the result.

3) Back-tracking for K Optimal Prefixes: We can do an inorder traversial on t and compute $p.l[i], 1 \leq i \leq K$ for every trie node using Eq. 1-3. When computing $p.l[i]$, we further records the optimal sub-problem on p's child nodes that are used to compute $p.l[i]$. For instance, if $l.l[2] + r.l[3]$ is the minimum value for $p.l[5]$, then we record that the solutions to the 2-prefix covering on l and 3-prefix covering on r constitute the solution to 5-prefix covering on p.

To generate the resulting K prefixes, we do a recursive back-tracking from $p_{root}.l[K]$ using the recorded optimal sub-problems. Everytime we encounter a 1-prefix covering on some node p', we incorporate the IP prefix of p' into the final solution. Eventually, we get the K optimal prefixes of problem 1 on the N input IP addresses.

C. Implementation

We discuss how to implement stateless migration on existing SDN software switches like OVS [].

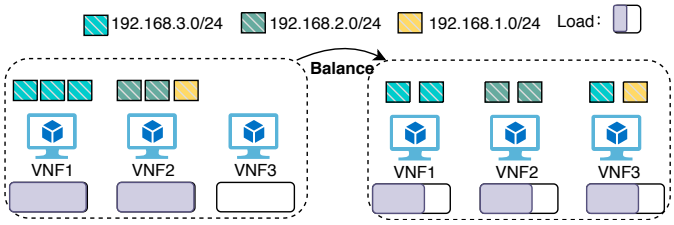

Figure 4. An example of workload balance with migration of a partial service chain.

1) Accelaration: The uni-bit trie constructed using the input IP addresses contain many single-child nodes, which can be merged to form a compressed trie. When implementing the K-prefix covering algorithm in ElasticNF controller, we first sort the N input IP addresses in ascending order, and then build a compressed trie in bottom-up order by gradually merging adjacent trie nodes. The i-prefix covering problem of each trie node can be computed along the trie construction. Since the compress trie contains only $2 \times N - 1$ nodes, the overall time complexity of the K-prefix covering algorithm can be accelarated to $O(N \times log(N) + N)$.

2) Choosing K: The value of K regulates the migration completion time and the number of generated sub-rules. In practice, we set K to 128, and set the threshold of sub-rule expiration time to 1 minute. We prove in the evaluation that these parameter settings lead to a good migration performance.

3) Obtain Input IP Addresses: The K-prefix covering algorithm requires distinct IP source addresses of existing flows as input. To acquire the IP source addresses, we use two match tables to forward input flows to destination VNF instances. To deploy a service chain rule with index i that forwards flows to VNF instance j, the ElasticNF controller only installs rule i in the first table. Flows matched by rule i is resubmitted to inport i on the second table. The second table is initialized with no active rules, and generates PACKET_IN event for each new flow received from inport i. The controller then retrieves the source IP address sip from the event, and installs a new rule in the second table that forwards packets received from inport i matching sip to VNF instance j.

When doing stateless migration, the ElasticNF controller queries the second table and obtain the IP source addresses of existing flows matched by rule i. Since software swith also records the traffic volume experienced by a rule, the controller also acquires the traffic volume sent from different source IP addresses.

V. Workload Balance with Stateless Migration

A motivating example of workload balance is shown in Fig. 4. VNF1 and VNF2 process three service chains and are both overloaded. The system scales out by adding a new instance VNF3. Now we must balance the workload across the three instances to eliminate the two hotspots.

With stateless migration, we can migrate service chain 3 from VNF2 to VNF3 and bring VNF2 back to normal. However, we run out of available migration options as VNF1 is only processing a single service chain. The only way to further

improve the balance is to split service chain 1 and partially migrate one third of the service chain traffic to VNF3. The statless migration approach discussed in Sec. IV allows us to partially migrate a service chain, making it possible to create a more balanced workload.

A. Migration of Partial Service Chain

Each IP prefix on the constructed trie from Sec. IV-B1 can be treated as a partial service chain, which is responsible for processing the covered flows of the IP prefix. By generating new rules with higher priorities than the original rule, we can migrate the partial service chain to another instance. When migration completes, the original rule is splited in two, each forwarding covered flows to different VNF instances.

Migrating a partial service chain represents a fine-grained approach to control the migrated traffic volume. Ideally, a service chain can be splitted into multiple parts and migrated to different instances for the most balanced workload. However, this may generate excessive sub-rules during migration, and slow down the SDN software switch.

To strike a balance, a service chain, together with its partial service chains, are only granted with a single migration opportunity when executing the workload balance algorithm. Depending on the type of the NF, we further set a threshold x on service chain throughput, and only migrate service chains whose throughput is larger than x.

B. Workload Balance Algorithm

The proposed workload balance algorithm is a greedy heuristic that migrates service chain traffic from over-packed instances to under-packed instances, with the goal to balance the workload among different instances using the smallest number of migrations.

1) Collecting Input Traffic: The algorithm first estimates the traffic throughput for each service chain on VNF instances that need balancing, by querying the second rule table as discussed in Sec. IV-C. The throughput of each (partial) service chain is estimated by summing up the collected traffic throughput for each covered flows.

2) Grouping the Instances: For each instance j, we calculate the overall throughput s_j by summing up the throughput of all service chains, and the target throughput t_j using method discussed in Sec. VI. If $s_j > t_j$, we categorize instance j as over-packed, and the over-packed capacity $s_j - t_j$ can be migrated out. Otherwise, instance j is categorized as under-packed, and the remaining capacity $t_j - s_j$ can be filled with new service chains. The set of over-packed instances is denoted as $\mathbb{O}$, while the set of under-packed instances is denoted as $\mathbb{U}$.

The algorithm operates by migrating the (partial) service chains from instances in $\mathbb{O}$ to instances in $\mathbb{U}$. When the algorithm ends, we expect the total throughput s_j of an instance to be close to its target throughput t_j. The algorithm has two stages:

3) Stage 1: The first stage is a best effort one. That is, it will try to migrate a (partial) service chain with the largest throughput from an over-packed instance to an under-packet instance.

For each instance j_o in $\mathbb{O}$, we first check whether the $s_{j_o} - t_{j_o} < x$. Due to the migration restriction on service chains whose throughput are smaller than x, we can no longer migrate a service chain out to further decrease the over-packed capacity. So we remove j_o from $\mathbb{O}$. Otherwise, we greedily select an instance $j_u = \arg\max_{j \in \mathbb{U}}(t_j - s_j)$ from $\mathbb{U}$ with the largest remaining capacity. Then, we search from the (partial) service chains on instance j_o, for one with the largest largest throughput between x and $\min(s_{j_o} - t_{j_o}, t_{j_u} - s_{j_u})$. Then we migrate that chain to instance j_u. If we can not find such a (partial) service chain, we also remove j_o from $\mathbb{O}$ and put j_o into a new set $\mathbb{L}$. Instances in $\mathbb{L}$ are still over-packed with too much service chain traffic, we use another strategy to process $\mathbb{L}$ in the second stage.

The stage 1 algorithm keeps looping through instances in $\mathbb{O}$ until $\mathbb{O}$ becomes empty.

4) Stage 2: For each instance j_l in $\mathbb{L}$, the second stage algorithm tries to decrease the throughput of j_l below the target throughput t_j. To do so, we find an instance j_u in $\mathbb{U}$ with the largest remaining capacity, search for a (partial) service chain in j_l with the smallest throughput bewteen $s_{j_l} - t_{j_l}$ and $s_{j_l} - t_{j_l} + s_{j_u} - t_{j_u}$, and migrate that (partial) service chain to j_u. The retriction on the throughput of the migrated service chain guarantees that every migration improves the balance. The workload balance algorithm terminates when $\mathbb{L}$ becomes empty.

VI. Adaptive Schedule

Parameter Selection For each type of NF, we set c to be the maximum throughput of an overloaded instance. c can be measured by monitoring of the overload status of an instance. We set $m = 0.5 \times c$ to be the medium throughput of an instance. We further set $x = 0.1 \times c$ to be the minimum throughput requirement of a migrated service chain. Note that these parameters are only used to algorithm execution, they can be constantly calibrated in practice.

Scale-out With Dedicated Instances The controller constantly monitors the throughput on each instance. If some instance j is overloaded ($s_j >= c$), scale-out is triggered by adding $\lceil \frac{\sum_{j \in \mathbb{J}} s_j}{m} \rceil$ instances. If the number of available cores is redundant, the controller scales out by launching dedicated instances, and run workload balance algorithm by setting t_j to m.

Scale-out with Consolidated Instances If the total number of instances needed for all the NFs exceeds the remaining number of available cores, we switch to use consolidated instances and consolidate several VNF instances on the same core. The CPU usage allocated to each type of consolidated is propotional to the overall throughput of the dedicatd NF instances. The t_j for dedicated instances is set to m, and the t_j for consolidated instances is set to the maximum capacity of the consolidated instance.

VII. RELATED WORK

In this section, we briefly review existing work on NF management and scheduling and state management, as well as load balancing problem in the context of NFV resources allocation.

NF Management and Scheduling. In recent years, several NFV platforms have been developed to accelerate packet processing on commodity servers [4, 21, 23, 32, 43]. Many works focus on managing and scheduling NFs for performance target or efficient resource usage [16, 25, 30, 39, 41, 46]. In motivation, we have discussed OpenNF[], Flurries[] and other relate works. We do not revisit them here. Our work focuses on a different situation: We integrate the allocation strategy into a hybrid approach, exploit strengths and avoiding weaknesses to achieve the best performance.

State Management. State management for elastic scaling of stateful NFs always been a hot issue in the NFV field, such as how to handle NF state appropriately and when to update the overall NF states. They all affect the overall NF performance, in terms of throughput and latency. Some [33] assume that all state is local, but neither shared or migrated, caledl this the Local-Only approach. Others [27] assume that all state is remote, stored in a centralized store, called this the Remote-Only approach. Neither Local-Only or Remote-Only could avoids efficient or high performance overheads when migration. According to the Stateless Migration through the prefixes cover, the original traffic is processed by the original NF, so that the state is not involved, thus it can achieve efficient and high performance.

Load balance. Load balance problem in the resource setting has been studied in task scheduling, cache system and cloud system [21], [23], [37]. William et al. define average load of all resources as the server's load [23], while in [21], a server's load is defined as the weighted average of different resources where the weights are based on the runtime loads. Those works are inherently a variant of bin packing problem [13], where the task (job or request) is atomic and unsplittable. Howeve, flow is splittable among all service chains, the load balance could more fine-grained in the same types NFs, so our objective is to minimize the variance of load in the platform, numbers of flows with the least migration during execution

REFERENCES

[1] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.

APPENDIX A

Theorem 1. *Let $S_K = \{p_1, ..., p_K\}$ be a solution to the K-prefix covering problem. Let t be a trie with L nodes, which is constructed using the N IP addresses of the K-prefix covering problem. Let $S_t = \{p_1^t, ..., p_L^t\}$ be the set of the IP prefixes represented by the L nodes of t. Then $S_K \subset S_t$.*

Proof. First, We prove that there does not exist an IP prefix p_j, such that $p_j \subset S_K$ and $p_j \not\subset S_t$. Assume that p_j is such an IP prefix. Then we can insert p_j into the trie t. After insertion of p_j, there are two situations. First, the last bit of prefix p_j is one of the trie node, then $p_j \subset S_t$. We get a contradiction. Second, the last bit of p_j is inserted as a new node. At this time, the descendent leaf nodes of p_j is an empty set, covering none of the N IP addresses. Therefore, p_j violates the constraint that each prefix should cover at least one input IP address, and S_K is not a valid optimal solution. We get another contradiction.

Second, we have $|S_K| < |S_t|$ because $|S_K| = K$, $K < N$ and $N < |S_t|$.

Finally, since $\forall p_j \subset S_K, p_j \subset S_t$ and $|S_K| < |S_t|$, we can conclude that $S_K \subset S_t$. □

APPENDIX B

To get a deeper insight into workload balance problem, we formally model the problem as an integer programming model.

Let $\mathbb{I} = \{1, \ldots, I\}$ define the set of service chains, and $\mathbb{J} = \{1, \ldots, J\}$ the set of VNF instances. Let $t_{i,j}$ define the original traffic throughput of service chain i on instance j, for any $i \in \mathbb{I}$ and $j \in \mathbb{J}$. Note that each service chain $i \in \mathbb{I}$ is deployed in a single instance denoted by $a_i \in \mathbb{J}$, that is, $t_{i,a_i} > 0$ and $t_{i,j'} = 0$ for all $j' \in \mathbb{J} \setminus \{a_i\}$.

Each service chain can be untouched, entirely migrated, or partially migrated. We assume that the migrated traffic throughput from any service chain should be relocated to a single destination instance. Let $\mathbb{S}_i$ be the set of possible amounts of traffic throughput that can be migrated from any given service chain $i \in \mathbb{I}$.

The decision variables are defined as follows. Let $x_{i,j}$ be the new traffic throughput of service chain $i \in \mathbb{I}$ on instance $j \in \mathbb{J}$ after migration. Define y as the maximum total traffic throughput among all instances. Define binary variable $z_{i,j} = 1$ if $x_{i,j} > 0$, and $z_{i,j} = 0$ if $x_{i,j} = 0$, for any $i \in \mathbb{I}$ and $j \in \mathbb{J}$.

$$\min \quad \alpha \cdot y + \beta \cdot \sum_{i \in \mathbb{I}} \sum_{j \in \mathbb{J}} z_{i,j} \tag{4}$$

$$\text{s.t.} \quad \sum_{i \in \mathbb{I}} x_{i,j} \le y, \qquad \forall j \in \mathbb{J} \tag{5}$$

$$x_{i,j'} \in \mathbb{S}_i, \qquad \forall i \in \mathbb{I},\ \forall j' \in \mathbb{J} \setminus \{a_i\} \tag{6}$$

$$\sum_{j \in \mathbb{J}} x_{i,j} = t_{i,a_i}, \qquad \forall i \in \mathbb{I} \tag{7}$$

$$0 \le x_{i,j} \le M z_{i,j}, \qquad \forall i \in \mathbb{I},\ \forall j \in \mathbb{J} \tag{8}$$

$$\sum_{j' \in \mathbb{J} \setminus \{a_i\}} z_{i,j'} \le 1, \qquad \forall i \in \mathbb{I} \tag{9}$$

$$z_{i,j} \in \{0, 1\}, \qquad \forall i \in \mathbb{I},\ \forall j \in \mathbb{J} \tag{10}$$

The objective function (4) is a weighted sum of two important performance indicators. The first term y is the maximum total traffic throughput among all instances. The second term $\sum_{i \in \mathbb{I}} \sum_{j \in \mathbb{J}} z_{ij}$ is the total number of service chains after migration, which also equals to I plus the total number of service chain splits.

Constraint (5) ensures that y is the maximum total traffic throughput among all instances. Constraints (6) ensures that

the amount of migrated traffic throughput are valid. Constraint (7) ensures that the total traffic throughput of each service chain should be the same before and after the migration. Constraint (8) enforces the relationships between variables $x_{i,j}$ and $z_{i,j}$. Constraint (9) ensures the assumption that the migrated traffic throughput from any service chain should be relocated to a single destination instance. Constraint (10) ensures that the variables $z_{i,j}$ are binary.

In the model above, the second term of the objective function is concerned with the total number of service chain splits. If you care about the total number of migrations, the objective function should be:

$$\alpha \cdot y + \beta \cdot \sum_{i \in \mathbb{I}} \sum_{j' \in \mathbb{J} \setminus \{a_i\}} z_{i,j'}$$

This integer programming model is hard to solve efficiently in practice. The Adapt controller has a tight time budget for generating the migration decisions. Therefore, we seek to design an efficient heuristic algorithm, taking into account the practical system situations.