Self-Adaptive SRv6-INT-Driven System Adjustment in Runtime for Reliable Service Function Chaining

Xuefeng Yan, Nelson L. S. da Fonseca, and Zuqing Zhu, *Fellow, IEEE*

***Abstract*—Self-adaptation of service function chains (SFCs) has been considered as an important attribute to ensure the resource-efficiency and reliability of network function virtualization (NFV) systems. In this work, we leverage the idea of integrating segment routing over IPv6 (SRv6) and in-band network telemetry (INT) seamlessly to realize SRv6-INT and explore the mutual benefits of SRv6 and INT for achieving self-adaptive SFC deployment. Specifically, we design and experimentally demonstrate a self-adaptive SRv6-INT-driven SFC deployment system that orchestrates network and IT resources timely to adapt to bursty traffic and network changes. We first enhance our previous design of SRv6-INT to better use it for self-adaptive SFC deployment, and then propose an IT resource management technique for Kubernetes (K8s) to accomplish resource allocation and contention resolution without offline virtual network function (vNF) profiling. Next, a closed-loop system is designed to manage SFCs in both the local and global ways. As for the local way, we let servers make local decisions based on the INT data encoded in packets to scale the vNFs running on them vertically. The global way involves the control plane, which oversees the SFC deployment in the whole network to change the number and placement of vNFs and the traffic routing through them. Finally, we prototype our proposal with commodity servers and hardware PDP switches based on Tofino ASICs, and experimentally demonstrate its effectiveness.**

***Index Terms*—IPv6 segment routing (SRv6), In-band network telemetry (INT), Programmable data plane (PDP), Service function chain (SFC), Resource management, Closed-loop control.**

I. Introduction

OVER past decades, networking technologies have been developing rapidly to adapt to the fast-growing traffic and network services with stringent quality-of-service (QoS) demands [1]. Specifically, the advances on physical-layer innovations [2–6], software-defined networking (SDN) [7, 8], network function virtualization (NFV) [9–11], and machine learning and artificial intelligence (AI) [12] have enhanced the performance of Internet to achieve seamless global coverage. With SDN and NFV, a service provider (SP) can decompose a network service into a series of virtual network functions (vNFs), instantiate the vNFs on general-purpose network elements (*i.e.*, commodity servers, switches and storages), and route the network service's traffic through them in sequence, deploying the network service with service function chains (SFCs) [13].

Despite the advantages, it is still challenging to realize cost-efficient and reliable SFC due to the difficulty of orchestrating network and IT resources timely to address bursty traffic and dynamic network status. Therefore, segment routing (SR) [14] has been proposed to facilitate adaptive traffic steering for SFC. For a packet flow, the ingress switch of its SR domain plans its routing path and operations along the path, encodes the results as an ordered list of path segments, and encapsulates the segment list as a stack of labels in each packet of the flow. Then, the subsequent switches along the routing path process each packet according to the segment list.

There are two typical implementations of SR, *i.e.*, the SR over multi-protocol label switching (SR-MPLS) [15] and SR over IPv6 (SRv6) [16], to encode SR labels based on MPLS labels and IPv6 addresses, respectively. Here, SRv6 can better explore the advantages of SR [14], and thus it has attracted much more research and development efforts. SRv6 defines a *Segment Routing Header* field in the extension header of IPv6, and encodes a stack of *Segment Lists* in it to indicate the segments that assemble a flow's routing path and how the flow's packets should be processed at the end of each segment. Therefore, an SP can customize the *Segment Lists* of a flow to steer it through an SFC and update its routing path in runtime.

In addition to SRv6, fine-grained network monitoring is also indispensable for adjusting the configuration of an SFC timely. Recently, programmable data plane (PDP) have led to novel network monitoring techniques that can satisfy this requirement, such as the in-band network telemetry (INT) [17]. INT realizes fine-grained and realtime network monitoring by letting packets carry the telemetry instructions that will be executed by the switches along the packets' routing paths to collect the corresponding monitoring results and insert them as INT fields in the packets. Although INT is promising, it may be incompatible with SRv6, because both techniques add fields in packets and thus can generate packets whose lengths exceed the maximum transmission unit (MTU) of a network. To solve this problem, we designed SR-INT in [18], which makes SR and INT share the fields in a packet in the time-division multiplexing (TDM) manner such that the two techniques are integrated seamlessly without resulting in excessively long packets, accomplishing highly-efficient network monitoring and adaptive traffic engineering simultaneously. However, the SR-INT in [18] was not developed based on SRv6 and its applications for SFC has not been explored yet.

On the other hand, the provisioning of SFC also involves instantiating vNFs on servers, using the runtime environments such as virtual machines (VMs) and containers. Compared with VMs, containers usually incur less overheads and have shorter deployment and migration time, offering a lightweight and agile way for instantiating vNFs. Kubernetes (K8s) [19] is widely-recognized as the *de facto* standard platform for

X. Yan and Z. Zhu are with the School of Information Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, P. R. China (email: zqzhu@ieee.org).

N. Fonseca is with the Institute of Computing, State University of Campinas, Campinas, SP 13083-852, Brazil.

Manuscript received on November 17, 2023.

managing containerized vNFs. To maintain the QoS, cost-effectiveness, and reliability of SFC, K8s needs to re-optimize provisioning scheme in runtime [10], where both IT resource allocation and contention resolution needs to be addressed. As for resource allocation, K8s adjusts resource quotas of an SFC vertically or horizontally to adapt to traffic fluctuation. The vertical scaling changes the IT resources (*e.g.*, CPU cycles and memory) allocated to a single vNF, while horizontal scaling adjusts the number of a vNF's replicas to accommodate time-varying traffic. As for contention resolution, the IT resources allocated to multiple vNFs is changed dynamically to enable highly-efficient resource sharing without starving any vNFs.

In this work, we expand our preliminary proposal of SRv6-INT in [20] to design and experimentally demonstrate a self-adaptive SRv6-INT-driven SFC deployment system that can orchestrate network and IT resources timely to adapt to bursty traffic and dynamic network changes. Specifically, the major improvement is to facilitate "zero-touch" network control and management (NC&M), *i.e.*, leveraging closed-loop and automatic system adjustment to optimize the provisioning of SFC in both the IT and network aspects in runtime. We first enhance the system design in [20] to better utilize SRv6-INT for self-adaptive SFC deployment. Then, we propose an IT resource management technique for K8s to realize resource allocation and contention resolution without offline vNF profiling.

Next, a closed-loop NC&M system is designed to manage SFC deployment in both the local and global ways. As for the local way, we let servers make local decisions based on the INT data encoded in packets to scale the vNFs running on them vertically. The global way involves the control plane, which oversees the SFC deployment in the whole network to change the number and placement of vNFs (*i.e.*, horizontal scaling) and the traffic routing through them. To make our system compatible with the standardized NFV frameworks, we implement the control plane based on the management and orchestration (MANO) framework [21] of European telecommunications standards institute (ETSI). We also leverage the idea of zero touch network and service management (ZSM) [22] provided by ETSI to separate the management and automation of SFC into IT and network domains and coordinate them with a global domain-wide orchestrator. Finally, we prototype our proposal with commodity servers and hardware PDP switches based on Tofino ASICs, and experimentally demonstrate its capability of adjusting the provisioning schemes of SFCs dynamically in runtime, to avoid QoS violations and precisely balance the tradeoff between QoS and resource consumption.

The rest of the paper is organized as follows. Section II briefly discusses the related work. We introduce the system design of our proposal in Section III. The implementation of our design is described in Section IV. In Section V, we present the experimental demonstrations for performance evaluation. Finally, Section VI summarizes the paper.

II. Related Work

Since its inception, SR has been applied for various purposes, including traffic engineering [23], network monitoring [24], SFC [25], *etc*. Due to its advantages, SRv6 has become the mainstream of SR. For a comprehensive survey on SRv6, one can refer to [14], and standardization work on SRv6 can be found in [16, 26], where Ref. [16] defines the programmability of SRv6 and how to support SRv6 in 5G networks has been discussed by 3GPP in [26]. As for the implementation of SRv6, open-source projects like ROSE [27] have been developed. Specifically, ROSE is dedicated to building an open-source SRv6 system based on Linux kernel.

As it can achieve real-time, fine-grained, and flow-oriented network monitoring, INT [17] has gained intensive research interests recently. The combination of INT and SR was initially considered in [28], where SR was leveraged to plan the routing paths of INT probes. However, this work did not try to reduce the increased overheads of running INT and SR together. We proposed SR-INT in [18] to explore the mutual benefit of SR and INT for adaptive network monitoring, without increasing the overall overheads. Nevertheless, SR-INT was not designed based on SRv6, and we did not consider to utilize SR-INT for optimizing SFC provisioning. In addition to SR-INT, the studies in [24, 29] also leveraged SR for network monitoring, but they invoked INT to send probes to traverse the concerned switches and links, which would induce much more overheads than SR-INT. In [20], we extended SR-INT to realize SRv6-INT, made it comply with SRv6, and prototyped a preliminary system based on it for accomplishing timely SFC re-optimization. To the best of our knowledge, this is the first work that utilized INT for orchestrating IT and network resources adaptively for SFC deployment. Compared with the IT monitoring approaches that run monitoring tasks in in-service tasks (*e.g.*, Bubble-flux [20]), our approach could monitor vNFs in the IT domain without inducing any adverse effects on in-service vNFs and yield more precise QoS measurements on throughput and latency. Nevertheless, closed-loop NC&M mechanism was not developed in [20].

The lightweight and agile features of containers allow SPs to deploy vNFs flexibly and cost-efficiently. vNFs at Layers 2 and 3 usually focus on packet forwarding, and thus they run in the user space of operating systems and are supported by kernel bypass techniques, such as the Data Plane Development Kit (DPDK) [30]. On the other hand, vNFs for Layers 4-7 require to handle the entire network protocol stack. Hence, flexibility and functionality take precedence in this category of vNFs, and a robust, kernel-based protocol stack is often required [31]. vNFs of different types can be formed into an SFC, which can be deployed by using SRv6 as the network engine. Therefore, previous studies [32–34] utilized SRv6 to deploy SFC and extended the implementation of SRv6 in Linux kernel for this purpose. However, they did not consider the performance monitoring or dynamic runtime re-optimization of SFC.

Runtime re-optimization of SFC needs "zero-touch" NC&M [22], which can realize self-configure, self-monitor, self-repair, and self-optimize based on service-level policies and rules, with minimum human intervention. More specifically, vNFs in an SFC should be managed adaptively with vertical and horizontal scaling. In [35], the authors explored data-based modeling methods for vertical and horizontal scaling of container-based jobs, but their proposal overlooked the issue of resource contention, which can happen when multiple jobs are co-

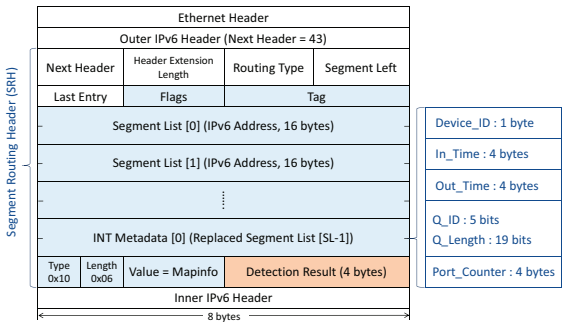


Fig. 1. Packet format designed for SRv6-INT (adapted from [20]).

located on a same server and lead to significant performance degradation [36, 37]. In order to address resource contention, existing approaches studied contention-aware SFC deployment [38–40], but the cost-efficiency of their solutions can be further improved by adjusting the resource sharing among vNFs in runtime, which often needs offline vNF profiling [39, 41]. Specifically, the dynamic resource quota and contention mitigation has to understand the performance models of vNFs in advance and collect necessary prior knowledge for resource allocation. Nevertheless, offline vNF profiling can be difficult or even infeasible, considering the variety of applications and complexity of network environments used for service provisioning [42]. Therefore, a model-free black-box approach would be required to accomplish dynamic resource allocation and adjustment without the need of offline vNF profiling.

III. System Design

In this section, we first review the protocol design of SRv6-INT in [20], and then describe our design of the self-adaptive SRv6-INT-driven SFC deployment system.

A. Protocol Design of SRv6-INT

Fig. 1 shows the packet format used for SRv6-INT [20], which was designed based on the standard of SRv6. Specifically, we modified the *Segment Routing Header* (*SRH*) in IPv6 extension header, and the major modifications are marked in blue in Fig. 1. We use the fields of *Flags* and *Tag* to indicate the SRv6-INT behaviors that should be taken on a packet. To reduce the length of each SRv6-INT packet, we replace a *Segment List* with an *INT Metadata* after a packet has traversed a segment on its routing path. Initially, the *SRH* of a packet contains a stack of *SL* IPv6 addresses, each of which denotes the last switch of a segment (*i.e.*, an endpoint), and thus there are *SL* segments on the packet's routing path. *INT Metadata* has the same length as *Segment ID* (*i.e.*, 16 bytes), and it includes the INT fields (five at most) that contains the telemetry data collected on an endpoint, recording the status of the last hop of an experienced segment.

The INT fields in each *INT Metadata* are illustrated in the right subplot of Fig. 1. The type-length-value (TLV) tuple in the modified *SRH* denotes the *MapInfo* that defines the INT data collection scheme, where each of the last 5 bits in the *Value* field tells whether a corresponding INT field should be filed with telemetry data. In *INT Metadata*, *Device_ID* stores the unique ID of the switch that performs the *End.T.INT* action in the SRv6-INT protocol, *i.e.*, the device ID of the last switch in a segment, *In_Time* is for the time-stamp when the packet arrives at the switch. *vNF_Latency* stores the processing latency of the vNFs experienced by the packet on the switch's local server. *Q_ID* and *Q_Length* are for the ID and length of the queue that stores the packet, respectively, and *Port_Counter* stores the packet counts of the flow. Different from the packet format in [20], we add a new field, *Detection Result*, which uses the last 4 bytes of *SRH* to store the vNF processing latency of the previous packet.

In order to facilitate SRv6-INT, we followed the principle of SRv6 to design three actions in [20]. Specifically, *H.Encaps.INT* is used to encapsulate *SRH* at athe ingress switch of the packet, *End.DT.INT* is to decapsulate *SRH* at the egress switch, and *End.T.INT* is designed for the operation on each endpoint, which performs normal INT operations with two mechanisms, *i.e.*, the "plain SRv6-INT" and "SRv6-INT for SFC", and replaces a *Segment List* with an *INT Metadata*. We store the processing latency of each vNF on the server in an internal register of the PDP switch that attaches to the server, and convey it to the local resource closed-loop controller on the server by leveraging the *Detection Result* in packets. Other than the modifications above, we change the definition of a subfield in *INT Metadata*. Specifically, the *Out_Time* defined in [20] is replaced with *vNF_Latency*, as shown in Fig. 1, which stores the processing latency of all the vNFs experienced by the packet on the server attached to the last switch of a path segment, for global closed-loop control.

B. System Architecture

Fig. 2 shows the architecture of our proposed self-adaptive SRv6-INT-driven SFC deployment system. Specifically, SRv6-INT is used to route SFC traffic, monitor SFC performance in runtime, and enable closed-loop resource management for SFC. Specifically, the network system consider both the IT and network domains to orchestrate resources for SFC deployment and readjustment. The network domain (*i.e.*, the SRv6 domain) consists of the PDP switches that equip Tofino ASICs and can be programmed with the P4 language [43]. We program the PDP switches to implement the packet processing pipelines for SRv6-INT in them, and manage them in runtime with an ONOS-based SDN controller through P4Runtime. As for the IT domain, we attach one or a cluster of Linux servers to some of the PDP switches directly, which will be used to instantiate vNFs. To ensure high resource efficiency, we choose to deploy containerized vNFs and develop our IT resource management system based on the container orchestrator of K8s [19]. Specifically, we deploy K8s master/workers in the servers, and modify the management component in the K8s master to manage the servers to instantiate, configure and remove vNFs.

With SRv6-INT, the working status of each vNF can be monitored in realtime (*i.e.*, the throughput and processing latency of the vNF can be precisely obtained by the PDP switch that connects directly to the vNF's server, and then encoded as INT fields in packets), and the packets that carry INT fields will be mirrored to the domain-wide orchestrator

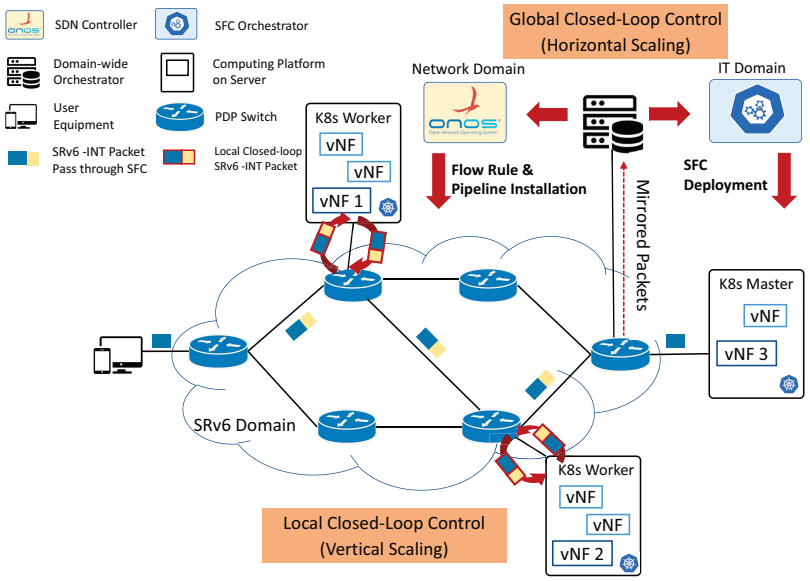


Fig. 2. Architecture of self-adaptive SRv6-INT-driven SFC deployment system.

before leaving the SRv6 domain. Then, the domain-wide orchestrator will process the data in the INT fields and provide suggestions to the SDN controller and SFC orchestrator to re-optimize SFC provisioning schemes adaptively, realizing the global closed-loop NC&M. To coordinate management of the network and IT domains, we modify the ONOS-based SDN controller and the K8s-based SFC orchestrator to make sure that they can orchestrate IT and network resources for SFC deployment and readjustment based on the real-time network status analysis done by the domain-wide orchestrator.

C. Global and Local Closed-loop SFC Management

In order to realize self-adaptive SRv6-INT-driven SFC deployment, we design two levels of closed-loop SFC management (*i.e.*, in the global and local manner). The local closed-loop SFC management runs on the per-server basis and is responsible for the vertical scaling of vNFs and mitigation of resource contention on a single server. Specifically, each PDP switch that directly connects to one or more servers will collect the processing latencies of the vNFs running on the server(s) and convey them to its local vNF controller by leveraging the *Detection Result* field in packets. Then, the local vNF controller will make timely decisions on vertical scaling of the vNFs and mitigation of resource contention accordingly.

The global closed-loop SFC management runs on the centralized domain-wide orchestrator to perform horizontal scaling of vNFs running on any servers in the network and rerouting of traffic through any SFC. Specifically, the horizontal vNF scaling can increase or decrease the replicas of a vNF to adapt to the traffic variation of an SFC. The domain-wide orchestrator conducts the global closed-loop SFC management based on the telemetry data encoded in SRv6-INT packets. To coordinate the two levels of closed-loop SFC management, we make them collaborate in the best-effort manner. Specifically, only when the local closed-loop SFC management cannot adapt to the traffic variation of an SFC, the telemetry data encoded in SRv6-INT packets will report abnormal work status continuously, which will trigger the global closed-loop SFC management to kick in. The detailed procedure of the collaboration will be explained in Section IV.

IV. System Implementation and Operation Procedure

In this section, we explain the implementation of our proposed system and the details of its operation procedure.

A. Implementation and Functional Modules

Fig. 3 shows the implementation and functional modules of our proposal. The control plane includes a global orchestrator and a local vNF controller. The local vNF controller runs on a server and checks the telemetry data encoded in SRv6-INT packets to perform vertical scaling of vNFs and mitigation of resource contention locally. The global orchestrator consists of an ONOS-based SDN controller for global NC&M, a K8s-based SFC orchestrator for global INT resource management, and a domain-wide orchestrator that coordinates the management of IT and network domains. The SDN controller manages the PDP switches to steer traffic through SFCs, while the SFC orchestrator locates on the K8s master for SFC management. Here, each vNF is instantiated in the form of pod. As shown in Fig. 3, the traffic forwarding between a server's network interface card (NIC) and its vNFs is implemented with Open-vSwitch (OVS), which is configured by K8s network plugin.

B. Procedure of Local Closed-loop Control

As illustrated in Fig. 4, we abstract the IT resources relevant to vNF deployment with the context of the non-uniform memory access (NUMA) [44], which is widely-used to manage multi-core computing platforms. Specifically, the NUMA architecture groups CPU cores into nodes, and the cores across nodes are interconnected through high-speed connections such

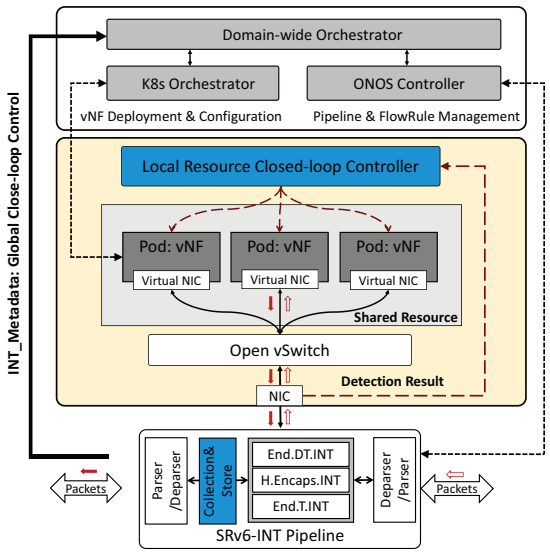


Fig. 3. Implementation and functional modules of our proposed system.

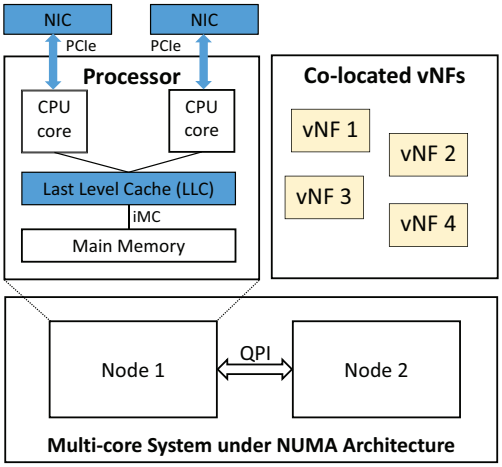


Fig. 4. vNF deployment based on NUMA architecture.

as the quick path interconnect (QPI). In addition to CPU cores, each node also contains local memory. The CPU cores of a same node share an integrated memory controller (iMC) and last level cache (LLC). To minimize the latency and bandwidth overhead associated with memory access, we implement our proposal to always place vNFs on the same NUMA node of the server's NIC. For the local closed-loop control, we implement a local vNF controller on each K8s worker, which operates based on the processing latencies of local vNFs (encoded in *Detection Result* of SRv6-INT packets). The processing latency of a vNF can be obtained by the PDP switch that directly connects to the vNF's server, by comparing the time when each packet leaves the PDP switch for the first time with the time when it re-enters the PDP switch after being processed by its vNF. The local vNF controller checks this performance metric and dynamically adjusts the IT resources (*e.g.*, CPU cycles and memory) assigned to in-service vNFs, to mitigate resource contention among the vNFs in each pod.

Specifically, the local vNF controller parses SRv6-INT packets to get the current processing latencies of all the co-located vNFs, and computes the 95% tail latency[1] of each vNF. When it finds that a vNF's 95% tail latency violates the vNF's QoS requirement, it will readjust the IT resources allocated to the vNF until the tail latency is pushed back to normal. Our design of the algorithm that tackles the problem of how to readjust the IT resources allocated to vNFs adaptively is based on the principle of not relying on any prior knowledge about the performance models of vNFs (*i.e.*, offline vNF profiling is not needed). Specifically, we leverage the concept of resource equivalence [42] to allocate IT resources to co-located vNFs. For a specific vNF, there usually exist multiple resource allocation schemes to satisfy its QoS demand on latency, if we consider the four resource types (*i.e.*, CPU frequency, CPU quota, LLC, and memory bandwidth). Then, we can design an algorithm based on a state machine to choose a type of resource to adjust in each iteration and find a feasible resource allocation scheme eventually. Table I lists the notations used in the following algorithm design, where the tail latency and processing rate (throughput) are in μs and Gbps, respectively.

TABLE I
NOTATIONS USED IN ALGORITHM DESIGN

Notations	Descriptions
$T[i]$	Type of resource required by vNF i
$L_{\max}[i]$	Longest tail latency tolerated by vNF i
$L[i]$	Current tail latency of vNF i
$\tau[i]$	Current tail latency slack of vNF i
$R[i]$	Current processing rate of vNF i
C_i	Current processing capacity of vNF i

The overall procedure of the local closed-loop SFC management is shown in *Algorithm* 1, which optimizes the resource allocations of vNFs running on a server based on their tail processing latencies, in the way that prior knowledge about the mapping between tail processing latency and resource utilization of each vNF is not required. We make a PDP switch continuously monitor and record the processing latencies of all the local vNFs and feed the latencies to the local vNF controller by leveraging SRv6-INT packets. *Line* 1 is for the initialization, where we select a resource type empirically and set it as the type of resource that needs to be adjust for all the local vNFs. Then, the while-loop of *Lines* 2-28 explains how the local closed-loop control works when the system is operational. Specifically, each in-service local vNF i is checked during operation (*Lines* 3-27). *Line* 4 makes the local vNF controller get the latest processing latency of vNF i by parsing each SRv6-INT packet to the vNF. Then, after processing N packets, the 95% tail latency of vNF i is obtained as L_{95} (*Lines* 5-6). *Line* 7 gets the gap between L_{95} and the longest latency that can be tolerated by vNF i ($L_{\max}[i]$) as $\tau[i]$, and *Line* 8 finds the vNF $\hat{i}$ whose gap $\tau[\hat{i}]$ is the largest. Next, if L_{95} already exceeds $L_{\max}[i]$, we first use *Lines* 10-12 to find the right type of resource ($T[i]$) to add to vNF i, and then try to allocate more type-$T[i]$ resource to vNF i in the best-effort manner (*Lines* 13-22). Otherwise, we just free redundant type-$T[\hat{i}]$ resource from vNF $\hat{i}$ (*Line* 24).

[1]In this work, we choose 95% tail latency because it is a reasonably strict performance indicator that can reveal the effect of "long tail" accurately.

Algorithm 1: Local Closed-loop SFC Management

```
1  select a resource type j empirically and set it as the type of resource to
   be adjusted for each local vNF i (T[i] = j);
2  while the system is operational do
3      for each local vNF i do
4          parse each packet to vNF i to get the latest processing latency
           in its Detection Result field;
5          if N packets have been processed for vNF i then
6              calculate 95% tail latency as L_95 based on recorded
               latencies, and reset the packet counter of vNF i;
7              τ[i] = L_max[i] − L_95;
8              î = argmax_{∀i}(τ[i]);
9              if L_95 > L_max[i] then
10                 if L_95 does not decrease since last calculation then
11                     run Algorithm 3 to get resource type T[i];
12                 end
13                 if there is enough type-T[i] resource to assign then
14                     add more type-T[i] resource to vNF i;
15                 else
16                     run Algorithm 2 to free type-T[i] resource;
17                     if there is enough type-T[i] resource to assign
                       then
18                         add more type-T[i] resource to vNF i;
19                     else
20                         continue;
21                     end
22                 end
23             else
24                 run Algorithm 2 to free type-T[i] resource from
                   vNF î;
25             end
26         end
27     end
28 end
```

Algorithm 2: Free a Type of Resource from a vNF

```
   Input: resource type j, vNF i.
1  deallocate a fixed amount of type-j resource from vNF i;
2  parse each packet to vNF i to get the latest processing latency in its
   Detection Result field;
3  if N packets have been processed for vNF i then
4      calculate 95% tail latency as L_95 based on recorded latencies, and
       reset the packet counter of vNF i;
5      if L_95 > L_max[i] then
6          revert the resource deallocation of vNF i;
7          run Algorithm 3 to update the type of resource T[i] to be
           adjusted for vNF i;
8      end
9  end
```

Algorithm 2 is a sub-procedure of *Algorithm* 1, and it tries to free type-j resource from a vNF i, which has the largest performance margin on tail processing latency. *Line* 1 just deallocates a fixed amount of type-j resource from vNF i. Then, *Line* 2 monitors the processing latency of vNF i after the resource deallocation. After N packets have been checked, we calculate the 95% tail latency as L_{95} based on the recorded latencies (*Lines* 3-4). Note that, we need to balance the tradeoff between accuracy and timeliness when choosing N. Specifically, a larger N makes the computation of 95% tail latency more accurate, but as it lets *Algorithm* 2 accumulate more packets before making a decision, the timeliness of resource adjustment becomes worse, and *vice versa*. However, the restriction on timeliness can be ignored when the throughput or packets per second (pps) is reasonably high (*e.g.*, higher than 1 Gbps or $100,000$ pps). Hence, when choosing N in our experiments, we only consider the statistical accuracy of 95% tail latency and set $N = 1,500$. Then, if we find that L_{95} exceeds $L_{\max}[i]$, the resource deallocation is reverted and we run *Algorithm* 3 to update the type of resource $T[i]$ to be adjusted for vNF i in the next round (*Lines* 5-8).

Algorithm 3: State Machine to Determine Resource Type

```
   Input: vNF i with resource type T[i].
1  map T[i] = {1, 2, 3, 4} to {CPU quota, LLC, memory bandwidth,
   CPU frequency}, respectively;
2  T[i] = (T[i] + 1)  mod 4;
3  return(T[i]);
```

The state machine that finds the type of resource $T[i]$ to be adjusted for vNF i in the next round is described in *Algorithm* 3, which works as the sub-procedure of *Algorithm* 2 based on the idea of resource equivalence (*i.e.*, different types of resources can be interchanged to achieve the same QoS target). We map $T[i] = \{1, 2, 3, 4\}$ to CPU quota, LLC, memory bandwidth, and CPU frequency, respectively (*Line* 1). *Line* 2 makes the state machine iterate the resource types in sequence, and the new resource type is returned in *Line* 3.

C. Procedure of Global Closed-loop Control

The primary goal of the global closed-loop SFC management is to perform horizontal scaling of vNFs running on any servers in the network and rerouting the traffic of any in-service SFCs. Meanwhile, the global closed-loop control should coordinate with the local one to explore their mutual benefits. Specifically, whenever the local closed-loop control cannot resolve the resource contention on a server, the SRv6-INT packets that traverse the vNFs on the server will convey the abnormal network status to the domain-wide orchestrator, which will then invoke the global closed-loop control. In this way, we do not need to invoke network-wide re-optimization frequently, making SFC readjustment more timely and relieving the complexity of network reconfiguration.

Algorithm 4: Global Closed-loop SFC Management

```
1  while the system is operational do
2      for each in-service vNF i do
3          parse each packet that has traversed vNF i to get the latest
           telemetry data (in its INT Metadatas) regarding the vNF;
4          get current throughput and processing latency of vNF i as R_i
           and L_i, respectively;
5          update the maximum throughput and longest processing
           latency of vNF i as R*[i] and L*[i], respectively;
6          set the maximum capacity of vNF i as C*_i = R*[i] · L*[i];
7          if (L[i] > L_max[i]) AND (this exception of vNF i has been
           persisted for more than t̂ seconds) then
8              get current capacity of vNF i as C_i = R_i · L_i;
9              set the new number of vNF replicas as n = ⌈C_i / C*_i⌉ ;
10             instruct SFC orchestrator to deploy n replicas of vNF i;
11             tell SDN controller to reroute traffic over vNF replicas;
12         end
13     end
14 end
```

Algorithm 4 shows the detailed procedure of the global closed-loop SFC management, which runs on the domain-

wide orchestrator. For each in-service vNF i, *Lines* 3-6 parse each SRv6-INT packet that has traversed it to get the latest telemetry data regarding it, and update the maximum throughput, longest processing latency, maximum capacity of vNF i accordingly. Then, if the current processing latency of vNF i exceeds the longest latency that can be tolerated by vNF i and this exception has lasted for a period that is longer than the preset threshold $\hat{t}$, the global SFC management will kick in to scale vNF i horizontally and reroute the SFC traffic of vNF i through the new replicas (*Lines* 7-12). Specifically, in *Line* 10, when the SFC orchestrator needs to deploy new replicas of vNF i, it selects the server(s) with the most available IT resources, while the rerouting of the SFC's traffic in *Line* 11 is accomplished with multi-path routing to distribute the traffic evenly to all the replicas of vNF i. Specifically, we leverage the longest common subsequence based algorithm in [13] to place the vNF replicas and reroute traffic through them.

V. Experimental Demonstrations

In this section, we conduct experiments to demonstrate the effectiveness of our proposed system.

A. Experimental Setup

We set up a network testbed as shown in Fig. 5 to evaluate the performance of our self-adaptive SRv6-INT-driven SFC deployment system and demonstrate its effectiveness experimentally. The testbed's data plane consists of four 32-port PDP switches based on Tofino ASICs, which are interconnected in an SRv6 domain using 40-Gbps links, and three commodity servers that are used for vNF deployment (*i.e.*, two servers work as K8s workers and one is configured as the K8s master). Each local connection between a PDP switch and the server attached to it operates at 10 Gbps. Each server contains two NUMA nodes, each of which is equipped with an Intel 10-core Xeon Silver-4210 2.20 GHz CPU, 32 GB of DDR4 memory, and a 4-port Intel 10GbE NIC. The control plane includes an ONOS-based SDN controller, an SFC orchestrator and a domain-wide orchestrator, which are all running on an independent Linux server. The user equipment is also emulated with a Linux server for traffic generation and result analysis.

TABLE II
Parameters of Experimental Demonstrations

Parameters	Settings
Port-count of PDP switch	32
Capacity of inter-switch link	40 Gbps
Capacity of server-switch link	10 Gbps
NUMA nodes in a server	2
CPU cores in an NUMA node	10
Memory in an NUMA node	32 GB
vNF types	Snort, Fwd, ntopng, and nDPI
Packet sizes	$\{200, 256, 512, 1024\}$ Bytes

Our IT resource management system is developed based on K8s v1.23.17 using containerd v1.7.2 as the container engine and CAT [45] to allocate/adjust LLC resources to vNFs. Table II summarizes the key parameters and their settings used in the experiments. The experiments use four types of vNFs: packet forwarding based on OVS [46] (Fwd), intrusion detection with Snort [47] (Snort), deep packet inspection via nDPI [48] (nDPI), and network monitoring using ntopng [49] (ntopng).

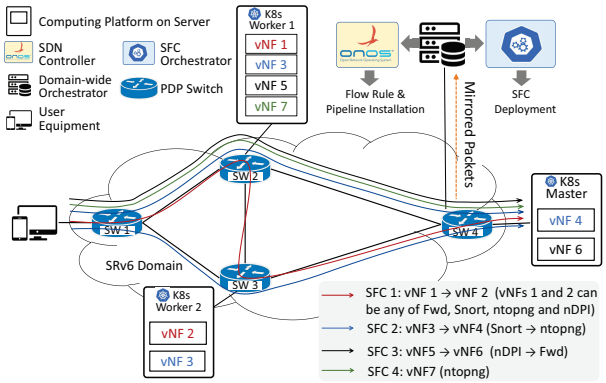


Fig. 5. Experimental setup.

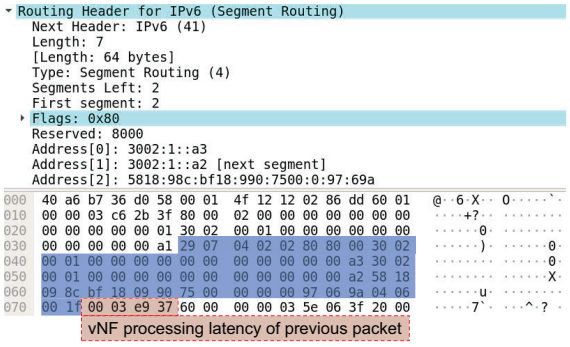
B. Functional Verification

We first deploy *SFC* 1 in the testbed as shown in Fig. 5, and set its routing path as *Switch* (*SW*) 1→*SW* 2→*SW* 3→*SW* 4, where its traffic experiences *vNFs* 1 and 2 on the servers attached to *SWs* 2 and 3, respectively. According to our design, the processing latency of each vNF in *SFC* 1 is measured by its local PDP switch. For instance, when a packet arrives at *SW* 2 for the first time, it time-stamps the packet as its ingress time, and after the packet has been processed by *vNF* 1 and reentered *SW* 2, it time-stamps the packet again as its egress time. Then, by comparing the two time-stamps, *SW* 2 obtains the processing latency of *vNF* 1, stores it locally, and encodes the latency in the *Detection Result* field of the next packet that is for *SFC* 1 to convey to *K8s Worker* 1. Fig. 6 shows the Wireshark captures of a packet, when it is sent out from *SW* 2 for the first and second time. In Fig. 6(a), when the packet is sent out from *SW* 2 to *vNF* 1, its *Detection Result* field is encoded with the vNF processing latency of its previous packet in *vNF* 1, which will be collected by *K8s Worker* 1. Next, in Fig. 6(b), when the packet reenters *SW* 2, the *vNF_Latency* field in the corresponding *INT Metadata* is encoded with the processing latency of *vNF* 1, which can be collected by the domain-wide orchestrator for the global closed-loop control.

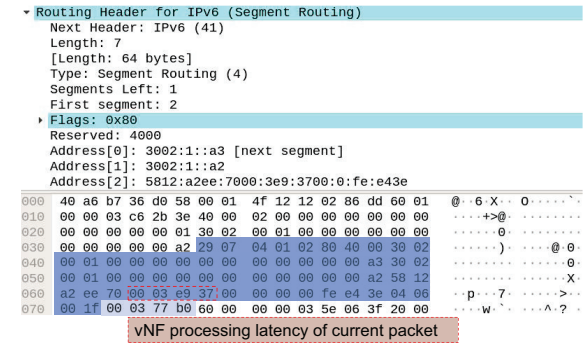
Next, we benchmark the largest-achievable packet processing throughput of the four vNFs, *i.e.*, the packet forwarding (Fwd), Snort, nDPI and ntopng. Here, the largest-achievable throughput of a vNF is obtained when we assigning the most computing resources to it (*i.e.*, letting it use all the 10 CPU cores in the NUMA node that the network interface card is assigned to). The results in Fig. 7 show that the throughput of each vNF increases with the packet size.

C. Self-Adaptive SFC Adjustment

Then, to verify that our proposal can achieve self-adaptive SFC adjustment, we conduct experiments on SRv6-INT-driven horizontal scaling of vNFs first. As explained in Section IV-C, the horizontal scaling of a vNF is invoked by the global closed-loop control, which happens when the local closed-loop



(a) Packet from *SW* 2 to *vNF* 1



(b) Packet from *SW* 2 to *SW* 3

Fig. 6. Wireshark captures to verify vNF processing latency collection.

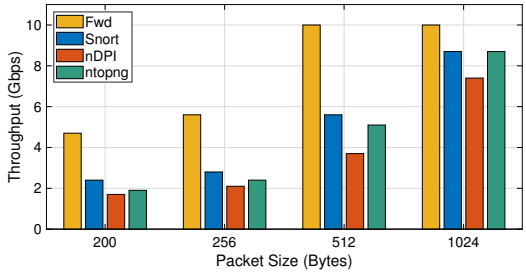
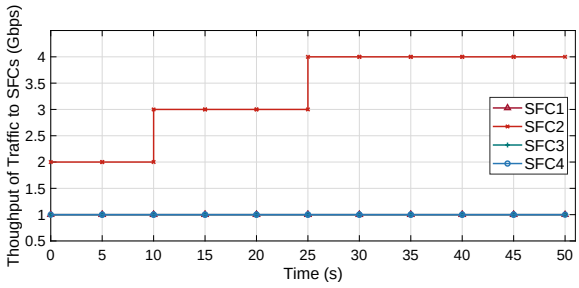


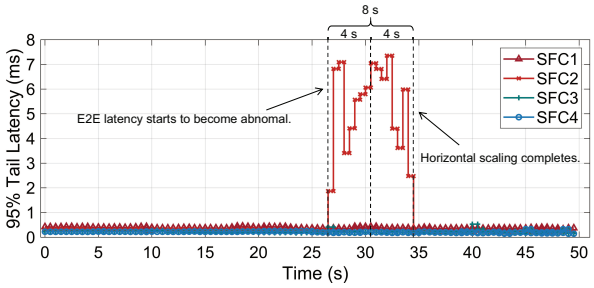
Fig. 7. Largest-achievable throughput of vNFs running on an NUMA node.

control on a server cannot resolve the abnormal processing latency increase of a vNF within a period of $\hat{t}$ (*i.e.*, we set $\hat{t} = 4$ seconds in the experiments). This time, we deploy all *SFCs* 1-4 in the testbed with the schemes shown in Fig. 5, where the routing path and vNFs of each SFC are plotted with a unique color. The vNFs deployed on *K8s Worker* 1 are Fwd, Snort, nDPI and ntopng, for *vNFs* 1, 3, 5, and 7, respectively. And the *vNF* 2 deployed on *K8s Worker* 2 is nDPI.

In this experiment, we increase the throughput of the traffic to the SFCs as that in Fig. 8(a), where the traffic to *SFCs* 1, 3 and 4 stay at 1 Gbps for the whole experiment, but that to *SFC* 2 is increased by 1 Gbps at $t = 10$ and 25 seconds, respectively. In Fig. 8(b), it can be seen that when the traffic to *SFC* 2 is increased for the first time, its end-to-end (E2E) latency does not change, attributing to the fact that the impact of the traffic increase is successfully resolved by the local closed-loop control on *K8s Worker* 1 to adjust the resources allocated to the *vNF* 3 (Snort) in *SFC* 2. However, when the traffic is increased again at $t = 25$ seconds, the local closed-loop control cannot mitigate the impact anymore and thus the E2E latency of *SFC* 2 surges from several hundred μs to above 7 ms and stays there, as shown in Fig. 8(b). Hence, the global closed-loop control kicks in to duplicate a *vNF* 3 on *K8s Worker* 2 and reroute half of the traffic to *SFC* 2 to it by letting the SDN controller modify the corresponding SRv6-INT flow entries on *SWs* 1 and 2. Specifically, Fig. 8(b) indicates that the abnormal E2E latency of *SFC* 2 lasts for $\sim$8 seconds, in which 4 seconds are used to satisfy the condition to invoke the global closed-loop control, and thus it takes the global closed-loop control $\sim$4 seconds to scale *vNF* 3 horizontally and bring the E2E latency of *SFC* 2 back to normal.



(a) Throughput of traffic to *SFCs* 1-4

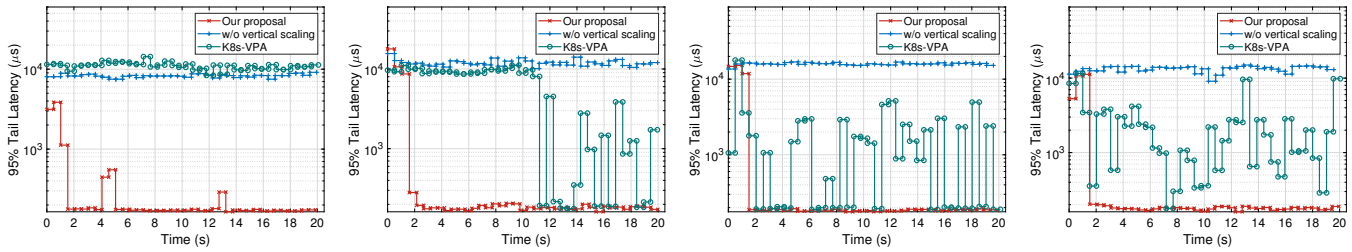


(b) Tail E2E latencies of SFCs

Fig. 8. Experimental results of horizontal scaling of vNF.

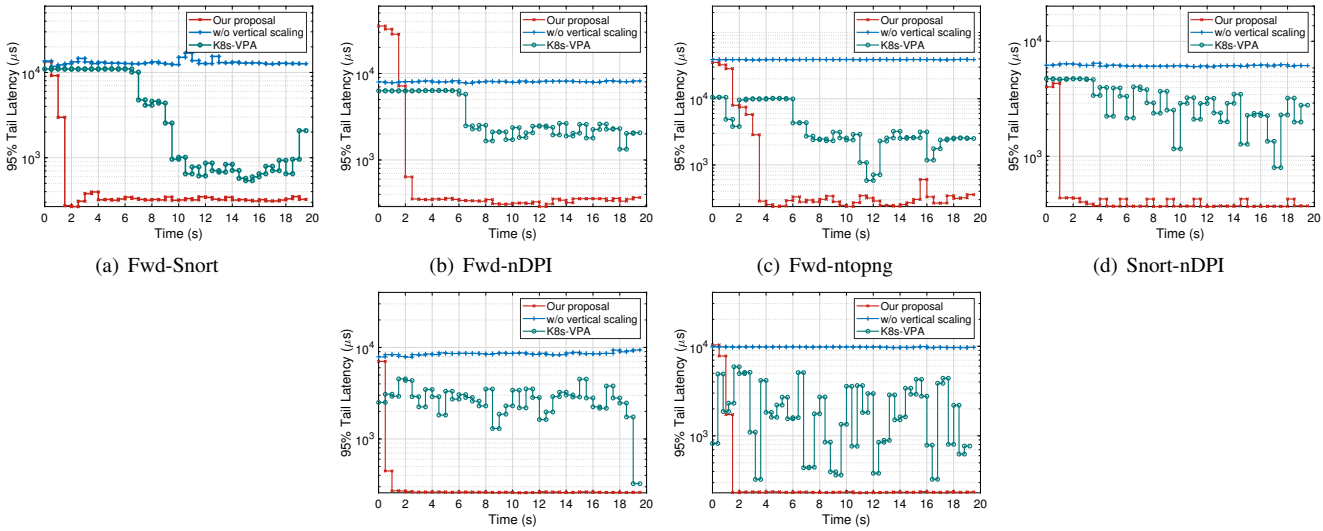
Next, we conduct experiments to demonstrate SRv6-INT-driven vertical scaling of vNFs, which is accomplished by the local closed-loop control on each server. Moreover, to better evaluate the performance of our proposal, we consider two benchmarks, *i.e.*, the one without vertical vNF scaling and the one that realizes vertical scaling of vNFs with the native vertical pod autoscaler in K8s (K8s-VPA).

We first consider the simple scenario that there is only one vNF running on a server, and plot the 95% tail latency of each vNF in Fig. 9. Here, we design the experiments to make all the vNFs experience abnormally long processing latency at the beginning. It can be seen that if vertical vNF scaling is not included, the 95% tail latency of each vNF just stays abnormally long for the whole experiment. As for K8s-VPA, it cannot stably reduce the tail latency of each vNF either, due to the fact that it cannot adjust all the four types of resources adaptively as our proposal does. Our proposal performs the best in Fig. 9. Specifically, it leverages the local closed-loop control to reduce the 95% tail latency of each vNF below 1 ms within 2 seconds and maintain the latency at the low level thereafter. Hence, the results in Fig. 9 verify that our proposal can adjust the 4-dimensional resources allocated to each vNF



(a) Fwd (b) Snort (c) nDPI (d) ntopng

Fig. 9. Results of managing a single vNF on sever with local closed-loop control.



(a) Fwd-Snort (b) Fwd-nDPI (c) Fwd-ntopng (d) Snort-nDPI

(e) Snort-ntopng (f) nDPI-ntopng

Fig. 10. Results of managing an SFC that contains two vNFs with local closed-loop control.

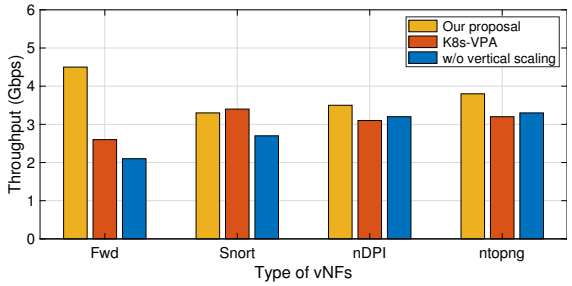
adaptively to maintain its packet processing performance.

Then, we repeat the experiments to consider an SFC that contains two vNFs, each of which runs on an independent server. Specifically, the experiments consider the *SFC* 1 in Fig. 5 with its two vNFs deployed on *K8s Workers* 1 and 2, respectively, where the two vNFs can be any of the six possible combinations of Fwd, Snort, nDPI, and ntopng, and we still make each SFC experience abnormally long E2E latency at the beginning. The results on the tail E2E latency of each SFC are plotted in Fig. 10, and similar trends as those in Fig. 9 can be observed. Specifically, our proposal can reduce the 95% tail E2E latencies of all the SFCs below 1 ms within 4 seconds and maintain the latency at the low level thereafter, outperforming the two benchmarks significantly.

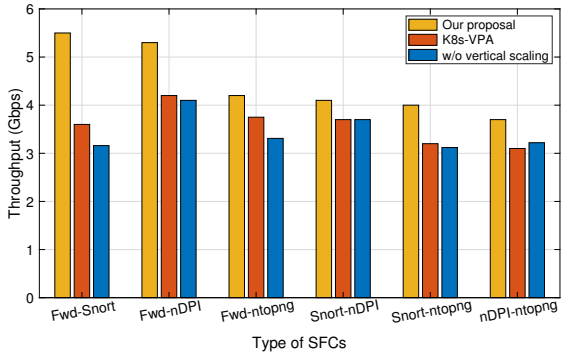
Next, we consider the situation where the four vNFs are co-located on a single server, to check the capabilities of the three approaches on mitigating the resource contention among the vNFs. Specifically, each experiment measures the largest-achievable throughput of a vNF under the condition that the throughput of each of the three remaining vNFs is fixed at 1 Gbps using the packet size of $1,024$ bytes. The results in Fig. 11(a) indicate that our proposal can achieve the largest throughput except for the case of Snort, where its throughput is slightly smaller than that of K8s-VPA. This is because K8s-VPA can allocate CPU quota aggressively to a vNF, while the performance of Snort is very sensitive to this type of resources. We also repeat the experiments to measure the E2E throughput of *SFC* 1 (with different vNF configurations) when it is colocating with *SFCs* 2-4 in the network as shown in Fig. 5. The results are shown in Fig. 11(b), indicating that the E2E throughput achieved by our proposal is always higher than those by the benchmarks. Fig. 11 confirms that our proposal can effectively mitigate the resource contention among co-located vNFs to improve their packet processing performance.

In all, the superiority of our proposal over the two benchmarks in Figs. 9-11 is attributed to three reasons. First, we optimize resource allocations of vNFs in a more comprehensive way (*i.e.*, *Algorithm* 1 adjusts the 4-dimensional resources used by each vNF adaptively, while K8s-VPA only considers the usages of CPU and memory of each vNF). Second, the benchmarks either do not adjust the resources allocated to vNFs or only adjust the resources based on historical information, while *Algorithm* 1 realizes local closed-loop SFC management in runtime by monitoring the QoS metrics of each vNF proactively and making adaptive changes to avoid QoS violations. Lastly, when seeing a QoS violation, K8s-VPA only tries to assign unused resources to the related vNF, while *Algorithm* 1 can take resources from the vNFs that still have performance margin to allocate to the concerned vNF.

Finally, we conduct experiments to verify that our proposal can achieve self-adaptive resource adjustment according to dynamic traffic changes. We still have the four vNFs co-

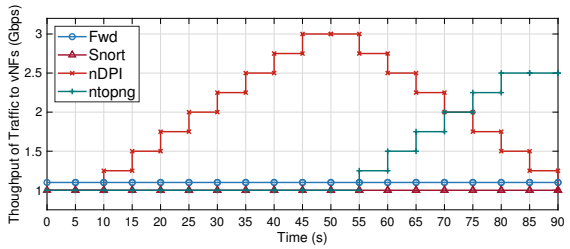


(a) Throughput of a vNF when colocating with three other vNFs.

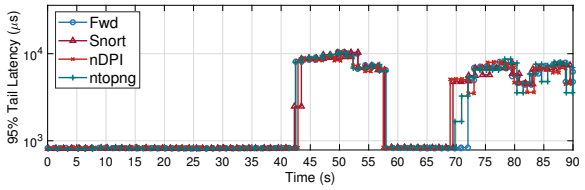


(b) E2E Throughput of an SFC when colocating with other SFCs.

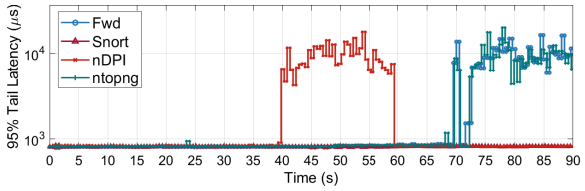
Fig. 11. Throughput in colocating scenarios.



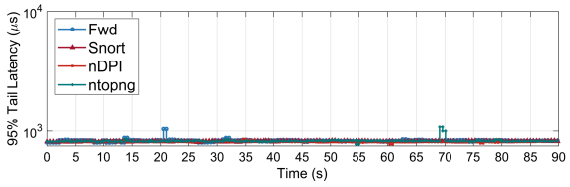
(a) Dynamic traffic to vNFs



(b) 95% tail latency of vNFs without vertical scaling



(c) 95% tail latency of vNFs with K8s-VPA



(d) 95% tail latency of vNFs with our proposal

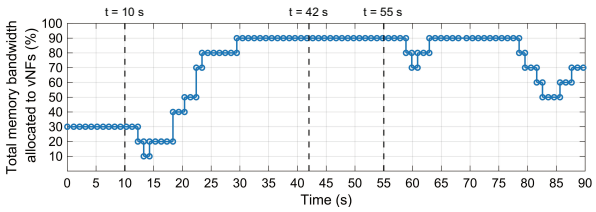
Fig. 12. Experimental results of dynamic traffic scenario.

located on a single server, but make the traffic throughput to them change according to the curves plotted in Fig. 12(a). Figs. 12(b)-12(d) shows the results on 95% tail latency of without vertical scaling, K8s-VPA and our proposal, respectively. In Fig. 12(b), as there is no vertical scaling to adjust the resources allocated to the vNFs according to their traffic volumes, resource contentions happen during the periods of $[42, 57]$ seconds and $[70, 90]$ seconds, where the 95% tail latencies of all the vNFs increase dramatically. The situation of using K8s-VPA is better in Fig. 12(c), as the aforementioned resource contentions only increase the 95% latency of nDPI and that of Fwd during the two periods, respectively, while the packet processing performance of Snort and ntopng can always be maintained well. The 95% tail latencies in Fig. 12(d) verify the effectiveness of our proposal when there are dynamic traffic changes, as the 95% tail latencies of all the vNFs are maintained at the low level all the time, no matter how the traffic volumes to the vNFs change.

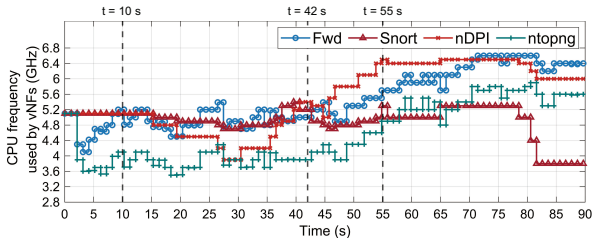
Furthermore, Fig. 13 shows how our proposal adjusts the CPU quota, memory bandwidth, CPU frequency and LLC allocated to the vNFs to adapt to their dynamic traffic changes. Initially, as the traffic volumes to the vNFs are all low, our proposal allocates a small amount of resources in each type to all the vNFs. Then, at $t = 10$ seconds, the traffic to nDPI starts to increase, thus the local closed-loop control tries to consolidate the CPU quota and CPU frequency allocated to nDPI and ntopng, and allocates more resources to Fwd to cope with the resource contention caused by the increase of the traffic to nDPI. At $t = 42$ seconds, our proposal finds that the performance of ntopng can no longer be maintained if the CPU frequency and CPU quota allocated to them are still kept at a low level, and thus the trends of the resource allocations to it gets reverted. When the time reaches $t = 55$ seconds, the input traffic of ntopng starts to increase. The local closed-loop control quickly detects this and starts to allocate more CPU quota and LLC to ntopng. By repeating such adaptive resource adjustment, our proposal maintains the 95% tail latency of all the vNFs at the low level throughout the experiment. Similarly, in Fig. 10, we observe that for the E2E processing tail latency metric, only our proposed solution addresses the issue of high tail latency under the initial configuration.

VI. Conclusion

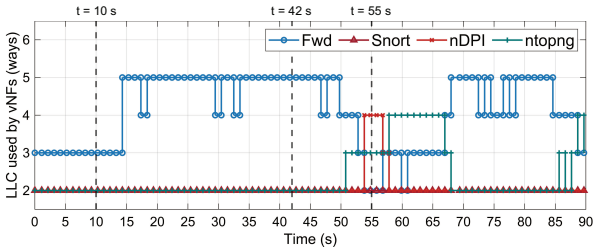
In this work, we designed and experimentally demonstrated a self-adaptive SRv6-INT-driven SFC deployment system that can orchestrate network and IT resources timely to adapt to bursty traffic and dynamic network changes. Specifically, our proposal leveraged closed-loop and automatic system adjustment to optimize the provisioning of SFC in both the IT and network aspects in runtime. We first proposed an IT resource management technique for K8s to realize resource allocation and contention resolution without offline vNF profiling. Then, based on the technique, a closed-loop NC&M system was designed to manage SFC deployment in both the local and global ways. We prototyped our proposal with commodity



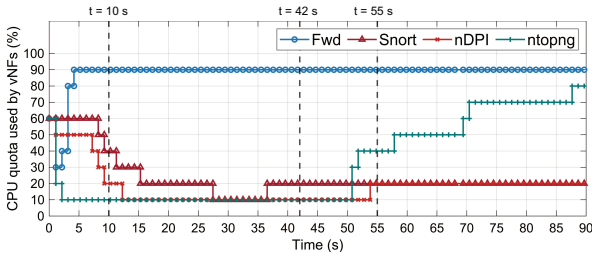
(a) Memory bandwidth allocated to *vNFs*



(b) CPU frequency allocated to *vNFs*



(c) LLC allocated to *vNFs*



(d) CPU quota allocated to *vNFs*

Fig. 13. Local closed-loop control for vNFs with dynamic traffic changes.

servers and hardware PDP switches based on Tofino ASICs, and experimentally demonstrated its capability of adjusting the provisioning schemes of SFCs dynamically in runtime, to avoid QoS violations and precisely balance the tradeoff between QoS and resource consumption.

ACKNOWLEDGMENTS

This work was supported by the NSFC project 62371432.

REFERENCES

[1] Cisco Annual Internet Report (2018-2023). [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[2] Z. Zhu, W. Lu, L. Zhang, and N. Ansari, "Dynamic service provisioning in elastic optical networks with hybrid single-/multi-path routing," *J. Lightw. Technol.*, vol. 31, pp. 15–22, Jan. 2013.

[3] L. Gong *et al.*, "Efficient resource allocation for all-optical multicasting over spectrum-sliced elastic optical networks," *J. Opt. Commun. Netw.*, vol. 5, pp. 836–847, Aug. 2013.

[4] P. Lu *et al.*, "Highly-efficient data migration and backup for Big Data applications in elastic optical inter-datacenter networks," *IEEE Netw.*, vol. 29, pp. 36–42, Sept./Oct. 2015.

[5] P. Marsch *et al.*, "5G radio access network architecture: Design guidelines and key considerations," *IEEE Commun. Mag.*, vol. 54, pp. 24–32, Nov. 2016.

[6] Z. Zhu *et al.*, "Impairment- and splitting-aware cloud-ready multicast provisioning in elastic optical networks," *IEEE/ACM Trans. Netw.*, vol. 25, pp. 1220–1234, Apr. 2017.

[7] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: an intellectual history of programmable networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–98, Apr. 2014.

[8] S. Li *et al.*, "Protocol oblivious forwarding (POF): Software-defined networking with enhanced programmability," *IEEE Netw.*, vol. 31, pp. 12–20, Mar./Apr. 2017.

[9] L. Gong and Z. Zhu, "Virtual optical network embedding (VONE) over elastic optical networks," *J. Lightw. Technol.*, vol. 32, pp. 450–460, Feb. 2014.

[10] J. Liu *et al.*, "On dynamic service function chain deployment and readjustment," *IEEE Trans. Netw. Serv. Manag.*, vol. 14, pp. 543–553, Sept. 2017.

[11] L. Gong, Y. Wen, Z. Zhu, and T. Lee, "Toward profit-seeking virtual network embedding algorithm via global resource capacity," in *Proc. of INFOCOM 2014*, pp. 1–9, Apr. 2014.

[12] W. Lu *et al.*, "AI-assisted knowledge-defined network orchestration for energy-efficient data center networks," *IEEE Commun. Mag.*, vol. 58, pp. 86–92, Jan. 2020.

[13] W. Fang *et al.*, "Joint spectrum and IT resource allocation for efficient vNF service chaining in inter-datacenter elastic optical networks," *IEEE Commun. Lett.*, vol. 20, pp. 1539–1542, Aug. 2016.

[14] P. Ventre *et al.*, "Segment Routing: A comprehensive survey of research activities, standardization efforts, and implementation results," *IEEE Commun. Surveys Tuts.*, vol. 23, pp. 182–221, First Quarter 2021.

[15] X. Xu *et al.*, "MPLS segment routing over IP," *RFC 8663*, Dec. 2019. [Online]. Available: https://datatracker.ietf.org/doc/rfc8663/.

[16] C. Filsfils *et al.*, "Segment routing over IPv6 (SRv6) network programming," *RFC 8986*, Feb. 2021. [Online]. Available: https://datatracker.ietf.org/doc/rfc8986/.

[17] C. Kim *et al.*, "In-band network telemetry (INT)," *Tech. Spec.*, Jun. 2016. [Online]. Available: https://p4.org/assets/INT-current-spec.pdf.

[18] Q. Zheng, S. Tang, B. Chen, and Z. Zhu, "Highly-efficient and adaptive network monitoring: When INT meets segment routing," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, pp. 2587–2597, Sept. 2021.

[19] Kubernetes. [Online]. Available: https://kubernetes.io/.

[20] X. Yan, Z. Xu, B. Chen, and Z. Zhu, "SRv6-INT: Runtime monitoring for green service function chaining in B5G-MEC," in *Proc. of ICC 2023*, pp. 3145–3150, May 2023.

[21] ETSI GR NFV-MAN 001 (v1.2.1). [Online]. Available: https://www.etsi.org/deliver/etsi_gr/NFV-MAN/001_099/001/01.02.01_60/gr_NFV-MAN001v010201p.pdf.

[22] M. Liyanage *et al.*, "A survey on zero touch network and service management (ZSM) for 5G and beyond networks," *J. Netw. Comput. Appl.*, vol. 203, pp. 68–81, Jul. 2022.

[23] S. Previdi *et al.*, "Source packet routing in networking SPRING problem statement and requirements," *RFC 7855*, May 2016. [Online]. Available: https://datatracker.ietf.org/doc/rfc7855/.

[24] P. Loreti *et al.*, "SRv6-PM: A cloud-native architecture for performance monitoring of SRv6 networks," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, pp. 611–626, Mar. 2021.

[25] A. Abdelsalam *et al.*, "SERA: Segment routing aware firewall for service function chaining scenarios," in *Proc. of IFIP Netw. Workshops 2018*, pp. 46–54, May 2018.

[26] D. Chandramouli and T. Sun, "System architecture for the 5G system," *3GPP Specification #: 23.501 (v17.4.0)*, Mar. 2022.

[27] ROSE Project. [Online]. Available: https://netgroup.github.io/rose/.

[28] T. Pan *et al.*, "INT-path: Towards optimal path planning for in-band network-wide telemetry," in *Proc. of IEEE INFOCOM 2019*, pp. 487–495, Apr. 2019.

[29] F. Aubry *et al.*, "SCMon: Leveraging segment routing to improve network monitoring," in *Proc. of IEEE INFOCOM 2016*, pp. 1–9, Apr. 2016.

[30] DPDK. [Online]. Available: https://www.dpdk.org/.

[31] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "StackMap: Low-latency networking with the OS stack and dedicated NICs," in *Proc. of USENIX ATC 2016*, pp. 43–56, Jun. 2016.

[32] A. Mayer *et al.*, "An efficient Linux kernel implementation of service function chaining for legacy VNFs based on IPv6 segment routing," in *Proc. of NetSoft 2019*, pp. 1–9, Jun. 2019.

[33] A. Abdelsalam *et al.*, "Implementation of virtual network function chaining through segment routing in a linux-based NFV infrastructure," in *Proc. of NetSoft 2017*, pp. 1–5, Jul. 2017.

[34] F. Duchene, D. Lebrun, and O. Bonaventure, "SRv6Pipes: enabling in-network bytestream functions," in *Proc. of IFIP Netw. 2018*, pp. 1–9, May 2018.

[35] K. Rzadca *et al.*, "Autopilot: workload autoscaling at Google," in *Proc. of ACM EuroSys 2020*, pp. 1–16, Apr. 2020.

[36] C. Sieber *et al.*, "Towards optimal adaptation of NFV packet processing to modern CPU memory architectures," in *Proc. of ACM CAN 2017*, pp. 7–12, Dec. 2017.

[37] Q. Cai *et al.*, "Understanding host network stack overheads," in *Proc. of ACM SIGCOMM 2021*, pp. 65–77, Aug. 2021.

[38] R. Rahman and P. Graham, "Compatibility-based static VM placement minimizing interference," *J. Netw. Comput. Appl.*, vol. 84, pp. 68–81, Apr. 2017.

[39] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *Proc. of ACM SIGCOMM 2020*, pp. 270–282, Jul. 2020.

[40] H. Yu *et al.*, "Octans: Optimal placement of service function chains in many-core systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, pp. 2202–2215, Sept. 2021.

[41] V. Chintapalli, M. Adeppady, and B. Tamma, "RESTRAIN: A dynamic and cost-efficient resource management scheme for addressing performance interference in NFV-based systems," *J. Netw. Comput. Appl.*, vol. 201, p. 103312, May 2022.

[42] S. Chen, C. Delimitrou, and J. Martinez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in *Proc. of ACM ASPLOS 2019*, pp. 107–120, Apr. 2019.

[43] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.

[44] Non-uniform memory access. [Online]. Available: https://en.wikipedia.org/wiki/Non-uniform_memory_access.

[45] CAT. [Online]. Available: https://github.com/intel/intel-cmt-cat/.

[46] Open vSwitch. [Online]. Available: https://www.openvswitch.org/.

[47] Snort. [Online]. Available: https://www.snort.org/.

[48] nDPI. [Online]. Available: https://github.com/ntop/nDPI.

[49] ntopng. [Online]. Available: https://github.com/ntop/ntopng/.