2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)

# RTNF: Predictable Latency for Network Function Virtualization

Saeed Abedi[†] Neeraj Gandhi[†] Henri Maxime Demoulin[†] Yang Li[‡] Yang Wu[‡] Linh Thi Xuan Phan[†]
[†]*University of Pennsylvania* [‡]*Facebook*

***Abstract*—A key challenge with network function virtualization is to provide stable latencies, so that the network functions can be treated simply as "bumps in the wire." In this paper, we present RTNF, a scalable framework for the online resource allocation and scheduling of NFV applications that provides predictable end-to-end latency guarantees. RTNF is based on a novel time-aware abstraction algorithm that transforms complex NFV graphs and their performance requirements into sets of scheduling interfaces; these can then be used by the resource manager and the scheduler on each node to efficiently allocate resources and to schedule NFV requests at runtime. We provide a complexity analysis of our algorithm and the design of a concrete implementation of our framework. Our evaluation, based on simulations and an experimental prototype, shows that RTNF can schedule DAG-based NFV applications with solid timing guarantees while incurring only a small overhead, and that it substantially outperforms existing techniques.**

## I. Introduction

Modern network functions no longer restrict themselves to forwarding packets; they also perform a variety of other functions, such as firewalling, intrusion detection, proxying, load balancing, network address translation, and WAN optimization. Traditionally, these functions have been implemented as middleboxes on dedicated hardware. But increasingly this infrastructure is being virtualized, and the physical middleboxes are being replaced by containers or virtual machines that run on a shared platform, such as the cloud [34]. This trend towards *network function virtualization (NFV)* offers a variety of potential benefits that resemble those of cloud computing—including consolidation, easier management, higher efficiency, and better scalability.

Ideally, the virtualized network functions would offer the same properties as the middleboxes they are replacing. In particular, they would offer low, predictable latencies. This is necessary for the network functions to remain transparent to the rest of the network: they are expected to behave as "bumps in the wire" that do not have any effect on the traffic that passes through them (other than the effects they were designed for). However, current cloud technology can only support these properties to a very limited extent. The reasons are partly historical: most existing platforms were developed for cloud *computing*, where worst-case latency guarantees are rarely needed. Of course, there are scenarios in which latency variations can cause problems even for cloud workloads – such as performance "cross-talk" between VMs [35] – and a number of mitigation techniques have been developed (e.g., [14, 26, 35]). However, most existing solutions take a best-effort approach and cannot provide predictable latencies.

More recently, a number of high-performance platforms have been built specifically for NFV: systems such as ClickOS [23], E2 [27], and NetVM [16] enable flexible and efficient creation, chaining and processing of virtualized network functions. In addition, resource management and scheduling techniques for NFV have also been developed—including, e.g., PLayer [19], SIMPLE [30], FlowTags [10], connection acrobatics [25], and OpenNF [13]. However, these platforms and techniques typically focus on improving the throughput and/or average latencies using best-effort approaches; as a result, they cannot provide any guarantees in terms of worst-case or tail latency. In our evaluation, we found that the 99th percentile latency of the packets when processed by a chain of network functions using E2 is an order of magnitude larger than its average latency!

In this paper, we aim to provide the missing link that could enable the design of *scalable* NFV platforms with latency guarantees. We focus on the *resource allocation and scheduling* of complex graph-based NFV applications in a cloud setting to meet their end-to-end latency constraints, thus bounding tail latency as a result. Our main contribution is a novel *abstraction algorithm* that transforms a complex DAG-based NFV application and its performance requirement into sets of resource-aware interfaces, which are much simpler than the application itself. These interfaces can then be used by the resource manager and the node-local schedulers to efficiently deploy the network functions, and to adapt the scheduling parameters based on the packet arrival rate to achieve the overall latency goals. Moreover, since the interfaces abstract away many details of the original applications, the difficult task of finding a good assignment of NFs to nodes becomes substantially easier.

We also present the design of a concrete NFV platform called RTNF. Our evaluation, based on a prototype implementation of RTNF as well as real-world network traces and network functions, shows that RTNF can schedule DAG-based NFV applications with solid timing guarantees, and that it substantially outperforms existing algorithms in terms of worst-case, tail and average latencies. Our results also show that RTNF incurs only a small overhead and utilizes resources efficiently, even when the workload is bursty. In summary, we make the following contributions:

- An efficient abstraction algorithm for transforming complex DAG-based NFV applications into chain-based interfaces, which enable efficient resource allocation and adaptive selection of the scheduling parameters to the actual packet arrival rate at run-time.
- An ILP formulation of the online placement of the NFs for customer requests based on the interfaces so as to meet their deadlines.
- Correctness and complexity analyses of our algorithm.
- A simulator and prototype implementation for RTNF.
- Extensive evaluation using real-world network functions and data center traffic traces.

978-1-7281-0678-6/19/$31.00 ©2019 IEEE
DOI 10.1109/RTAS.2019.00038

## II. Related Work

Several high-performance NFV platforms are already available, including e.g., ClickOS [23], NetVM [16], OpenBox [7], Dysco [44], NFP [38], and E2 [27]. These platforms provide a flexible, efficient and modular framework for programming, management, and deployment of NFs. A number of other efforts have looked at resource management and scheduling for NFV, including PLayer [19], SIMPLE [30], FlowTags [10], connection acrobatics [25], OpenNF [13], NFVnice [20], and E2 [27], to name a few. However, none of the existing solutions supports resource allocation with predictable end-to-end latency, which is a focus of RTNF.

Task graph scheduling in distributed systems is an established area of research (see [41] for a survey). However, prior work typically optimizes other metrics such as cost, energy, and resource utilization instead of providing timing guarantees. Solutions that do consider deadlines only support a single application [2, 8, 9, 42, 43], single-job DAG tasks (i.e., aperiodic tasks) [22, 33], or offline settings [11, 32]; hence, they are not suitable for our setting.

Recent networking research has also developed methods to achieve better predictability for network latencies [3, 29] and network function processing [17, 28, 39]. Unlike ours, these results are limited to either the network layer or individual network functions, and they cannot provide *end-to-end* latency predictability for DAG-based workloads.

The only existing work we are aware of that considers end-to-end latency guarantees for NFV is NFV-RT [21]. However, NFV-RT does not support general DAGs – it is restricted to NFV service *chains*. NFV-RT also has a number of limitations. For instance, when provisioning the resources for a service chain, it assumes a fixed maximum traffic rate. In contrast, RTNF is able to adapt its resource provisioning to the actual arrival rates of requests at run time, and thus utilizes resources more effectively. Furthermore, the analysis in NFV-RT assumes that the period of consolidation of two traffic flows (requests) is the inverse of their total rate, which is not true [1]. Although this assumption might not be an issue when throughput or best-effort latency is the primary goal, it can lead to violation of latency guarantees, as observed in our evaluation.

Finally, there is a long line of work in real-time scheduling, including scheduling of task graphs (see e.g., [4, 24, 31, 36, 37]). However, it assumes much simpler traffic and network models—e.g., most focus on a single-node multiprocessor—and cannot be applied to the cloud setting. Extending such techniques for NFV is an interesting future direction.

## III. System model and goals

We begin by stating a specific system model on which latency guarantee will be based; however, our approach is general and should work for other settings as well.

**Platform.** We assume the network has a fat-tree network topology [1], which is common in data centers. The platform is divided into multiple *pods*. Each pod consists of multiple racks that are connected to a number of aggregation (EoR) switches; all EoR switches are further connected to the core switches. Each rack has multiple machines and a top-of-rack (ToR) switch that connects these machines. Each network link (from a node to a switch, from a switch to a node, or from one switch to another) is associated with a total bandwidth. We assume the communication latency between two nodes in the same pod is bounded. We believe this is a reasonable assumption: any nondeterministic latency caused by collisions or ARP can be avoided in today's data center networks using SDN configuration; further, by disallowing nodes to send faster than the link capacity, we can eliminate the unpredictable delay caused by link congestion [3]. For simplicity, we assume that there are no link failures.

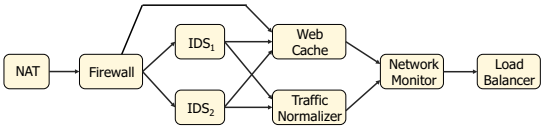


Fig. 1: An example NFV application.

Each machine hosts multiple containers (or virtual machines)[1] within which the network functions are executed. For concreteness, we assume that the kernel is configured to use the Earliest Deadline First (EDF) scheduling algorithm, as it has high utilization bound and is available in common kernels, such as Linux (the SCHED_DEADLINE scheduler) and Xen (the RTDS scheduler).

**Applications and requests.** The platform provides a range of NFV applications (services) to its users. Each application is a directed acyclic graph (DAG), whose vertices represent network functions (NFs), as illustrated in Figure 1. Each NF is characterized by a per-packet worst-case execution time (WCET), which can be obtained using existing WCET analysis [40] or by profiling. Each incoming packet is always processed by the NFs *along a path* in the DAG; the exact path depends on the packet's characteristics (e.g., header or payload) and can vary across packets (even of the same traffic flow). For instance, in Figure 1, after being processed by the NAT and the Firewall, packets are forwarded to either the Web cache or one of the IDS network functions, depending on the outcome of the Firewall function.

We associate each application with an end-to-end *deadline*, which represents the bound we wish to guarantee on the latency of any packet. This end-to-end deadline would depend on the depth of the graph and the specific NFs, and it must be met by all traffic flows (with potentially different packet rates) that are processed by the NFV graph.

We assume that the system is *dynamic*: at runtime, new users may submit requests for the provided NFV services, and existing users may leave the system. Each request specifies an NFV application the user wants to use and the packet rate of its traffic flow. (We use the terms 'flow' and 'request' interchangeably.) We say a request meets its deadline if the end-to-end latency of *every* packet of its traffic flow (i.e., from the instant the packet arrives at the

[1]Our approach is not specific to containers or VMs; however, we will use the term 'containers' in the rest of the paper for convenience.

first NF until the instant it leaves the last NF) does not exceed the application's end-to-end deadline.

**Challenges.** In this setting, our goal is to efficiently allocate resources at runtime so as to 1) meet the end-to-end latency guarantees while 2) minimizing resource consumption. A natural question to ask is whether this could be done simply by applying a known real-time scheduling algorithm. However, the scheduling problem we are facing is very complex. For each new request, we would need to decide:

1) Which containers should each NF of the requested NFV application execute in?
2) Which node should each container be executed on?
3) Which network path should be used to communicate between containers on different machines?
4) How many resources should each container get?

This problem has far more degrees of freedom than a typical scheduling problem in real-time systems, and it is too complex to solve from scratch every time a new request arrives. Most of the existing NFV solutions aim for good throughput and low *average* latency, which is not enough to bound the worst-case latency. The only system that comes close to achieving this is NFV-RT [21]; however, NFV-RT is limited to simple chains of NFs. We are not aware of any solution that can provide worst-case (or tail) latency bounds for general DAG-based NFV applications.

## IV. Overview of RTNF

Next, we present an overview of our approach and design. The detailed algorithms are discussed in Sections V–VII.

### A. Resource allocation approach

Recall that, given a new request, our goal is to compute a concrete deployment configuration for the associated NFV application – that is, an assignment of NFs to containers, an assignment of containers to machines, a route between containers, and the scheduling parameters for each container – such that the packet end-to-end latency will always be bounded by the application's end-to-end deadline.

**Basic idea.** To enable predictable latency, we first transform each complex NFV graph into a *scheduling interface* that captures the resources needed to meet the end-to-end deadline. Intuitively, the interface specifies (i) how NFs should be grouped into components, where each component will be deployed in a separate container and scheduled as a conventional real-time task, and (ii) the scheduling parameters (WCET or budget, period, deadline) for each component, such that: *any incoming request of the application will always meet its deadline if (1) all components of the interface are schedulable under their assigned parameters, and (2) there is sufficient network bandwidth for the data transfers between components.*

This interface (computed offline) provides an efficient way to allocate resources to online requests *and* to guarantee predictable latency. Specifically, as a new request of the NFV graph arrives, we can quickly determine both the assignment of NFs into containers (i.e., components) and the containers' scheduling parameters based on the interface. Further, due to the above property of the interface, we can reduce our original problem into a much simpler one: find an assignment of the containers onto the machines and a route between them, such that each container is schedulable on its assigned machine and there is sufficient bandwidth for the data transfers between containers. As containers have well-defined scheduling parameters, their schedulability can be verified using the EDF schedulability test (recall that the node-local scheduler uses EDF). Consequently, we can formulate this new problem as an ILP optimization that can be solved efficiently at run time.

**Interface computation challenges.** One challenge to realizing this approach is to decompose the graph into components. In general, having fewer components leads to lower communication overhead but higher CPU demand per component (which makes the component harder to schedule on a core). Another challenge is that the scheduling parameters of the interface's components depend on the packet rate of the request: the higher the packet rate, the higher the resource demand. However, this information is specific to each request, and it is only available when the request arrives but not during the interface computation.

**Insights.** Our abstraction algorithm within RTNF is based on the following insights: (1) Leveraging the single-path characteristic of a packet when processing through the NFV graph, we use a chain structure for the interface (i.e., the interface is a chain of components), following the topology of the DAG. Not only does this chain structure enable more efficient online assignment of (NFs within) components onto nodes, it also simplifies the computation of the components' scheduling parameters substantially. (2) To adapt to the varying online packet rates of the requests, for each application we compute not one, but a set of interfaces. Each such interface captures a range of traffic periods (i.e., inverse of the packet rates) for which the interface can ensure latency guarantees. This approach enables the system to select the most optimized interface for a request depending on its actual packet rate. (3) Our algorithm groups different parallel subpaths of the DAG with similar total WCETs into a component to minimize the computation overhead. Further, to reduce communication overhead, it minimizes the length of the interface using an efficient (and optimal) heuristic. (See Figure 4 for an example of the interfaces.)

### B. Design overview

Our design (sketched in Figure 2) consists of two interacting core components: 1) a centralized *controller*, which performs admission control and computes the resource allocations for requests, and 2) a set of distributed *run-time agents* running on the machines, which perform the deployment of the NFV graphs for new requests based on the resource allocation configurations given by the controller.

Internally, the controller executes a control agent that interacts with users (e.g., receives new requests and responds with acceptance or rejection decisions) and the run-time

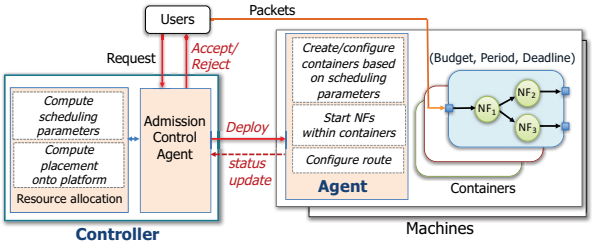



Fig. 2: RTNF design.

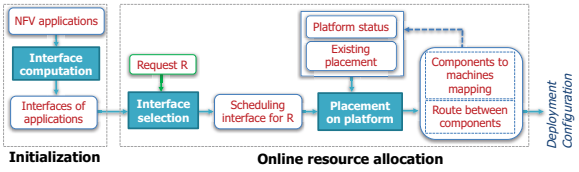



Fig. 3: Overview of the resource allocation.

agents on the nodes (e.g., to send new resource allocation configurations to deploy, and to receive node status updates). The control agent communicates with the resource allocation module, which implements our strategy to compute the resource configuration for the request. Our strategy guarantees that, if a feasible configuration exists, the request will meet its end-to-end deadline. In this case, the agent will admit the request and inform the agents on the assigned nodes to deploy their assigned NFs and containers according to the computed configuration. Upon admission, the user can begin sending its packets directly to the node on which the first NF of the graph is deployed.

Whenever the run-time agent on a node receives a deployment request from the controller, it will then create the containers, assign the NFs to containers, set scheduling parameters for each container (which will be used by the kernel's scheduler to schedule the container), and set up the route between the NFs according to the configuration given by the controller. It will then launch the NFs, which will begin receiving and processing packets from the user.

Figure 3 shows the overall flow of the RTNF resource allocation strategy. At initialization, it computes a set of interfaces for each NFV application graph. At run time, whenever a new request $R$ of an application $G$ arrives, it selects an interface of $G$ based on $R$'s traffic period (i.e., the inverse of the arrival rate), and sets concrete scheduling parameters for the components in the interface. It will then attempt to find a placement for the components of $R$'s interface onto the platform such that all components are schedulable and there is sufficient bandwidth for the transmission between them. If a feasible placement exists, the controller will admit the request and inform the run-time agent to deploy the NFs within each component on the assigned node within a separate container/VM, whose scheduling parameters are set to be that of the component.

## V. Interfaces for NFV Applications

In this section, we present the algorithm for computing the interfaces of NFV applications, which will be used by the online resource allocation in Section VII. As explained in Section IV-A, an interface of an application is a chain of components, such that the application's deadline is always met if all components are schedulable. Each component consists of multiple NFs that can be feasibly scheduled together within a container on a core. We first formally define the interface and its key properties.

### A. Interface model and properties

Recall that an NFV application is modeled by a DAG $G = (V,E)$, where each vertex $s_i \in V$ represents an NF and each edge $(s_i,s_j) \in E$ represents a possible data flow from $s_i$ to $s_j$. We denote by $\mathsf{wcet}(s_i)$ the WCET of $s_i$, and by $D$ the end-to-end deadline of $G$. We first define the components of $G$ that form its interface:

*Definition 1 (Component):* A component $C_k$ of $G$ is a subgraph of $G$ that will be scheduled together using a periodic task (within a separate container/VM) on a core. We denote by $\mathsf{period}(C_k)$, $\mathsf{deadline}(C_k)$ and $\mathsf{wcet}(C_k)$ the period, deadline, and WCET of (the task that corresponds to) $C_k$, respectively. To use the tight utilization-based EDF analysis, we require that $\mathsf{deadline}(C_k) \leq \mathsf{period}(C_k)$. The CPU bandwidth requirement of $C_k$, also known as its *density*, is given by $\mathsf{wcet}(C_k)/\mathsf{deadline}(C_k)$.

A chain of components of $G$ is determined by a map $\Pi : V \mapsto \{1,...,n\}$, where $\Pi(s_i)$ denotes the index of the component that contains $s_i$. Since the incoming traffic of a request for $G$ will be processed by the interface's chain of components, a valid chain should ensure that, for any edge $(s_i,s_j) \in E$, the component containing $s_i$ does not reside after the component containing $s_j$, i.e., we must have $\Pi(s_i) \leq \Pi(s_j)$. This property is necessary to to avoid traffic going against the order of the chain, which could significantly complicate the online assignment of the components to the platform. Further, since each component $C_k$ is a subgraph of $G$, and because each packet is processed along a path in $G$, each packet is processed along a path in $C_k$. Hence, the WCET of $C_k$ is, naturally, the largest WCET of any path in $C_k$, where the WCET of a path is defined as the total WCET of all NFs on the path.

*Definition 2 (Chain):* A chain of components, $\mathcal{C} = C_1 \rightarrow C_2 \rightarrow \cdots \rightarrow C_n$, is a valid chain of $G$ iff there exists a surjective map $\Pi : V \mapsto \{1,...,n\}$ from $V$ onto $\{1,\ldots,n\}$, such that (a) $\forall (s_i,s_j) \in E$, $\Pi(s_i) \leq \Pi(s_j)$, and (b) for all $1 \leq k \leq n$, $C_k$ consists of all $s_i \in V$ with $\Pi(s_i) = k$, and $\mathsf{wcet}(C_k) = \max\{\sum_{s_i \in P} \mathsf{wcet}(s_i) \mid \text{P is a path in } C_k\}$.

The interface captures a range of periods for which the latency guarantee of the requests is feasible; so as to enable the adaptation of components' scheduling parameters to varying traffic periods of online requests. We next highlight the characteristics of such periods.

*Definition 3 (Feasible periods):* An input period $T$ is feasible for a chain $\mathcal{C}$ of $G$ iff any request for $G$ with traffic period $T$ will meet its deadline if all $C_k$ are schedulable under the scheduling parameters $\mathsf{period}(C_k) = \mathsf{deadline}(C_k) = T$. Further, a range $\mathcal{T}$ is a feasible period range for $\mathcal{C}$ iff for all $T \in \mathcal{T}$, $T$ is a feasible period for $\mathcal{C}$.

We are now ready to formally define an interface.

*Definition 4 (Interface):* An interface $\mathcal{I}$ of $G$ specifies

- $\mathcal{C}$, a valid chain of $G$, and
- $\mathcal{T}$, a (non-empty) feasible input period range for $\mathcal{C}$.

**Properties of interfaces.** Again, recall from Section IV-A that an interface of $G$ should ensure (1) each of its components can be feasibly scheduled on a core, and (2) any request with traffic period within the interface's feasible period range will meet its deadline if the components are schedulable. First, for a component $C_k$ to be schedulable on a core, its WCET must be less than its deadline.[2] Thus, we must have $\mathsf{wcet}(C_k) < \mathsf{deadline}(C_k) \leq \mathsf{period}(C_k)$.

Second, for a request to meet its end-to-end deadline $D$, the sum of (i) the response time (i.e., queuing time plus execution time) of all components $C_k$ in $\mathcal{C}$ and (ii) the total transmission delay between any two consecutive components $C_k$ and $C_{k+1}$ must be no more than $D$. Suppose $T$ is a feasible period of the interface, and suppose $R$ is a request with incoming traffic period $T$. As the packets of $R$ arrive at each $C_k$ at a period of $T$, the period of $C_k$ is also $T$. Hence, if we set the deadline of each $C_k$ to be the same as its period (i.e., $T$), then the delay of a packet at $C_k$ is always bounded by $T$ if all components $C_k$ are schedulable.

Further, to minimize communication overhead, our online placement will always restrict the components of $G$ to nodes of the same pod (resp. rack). Let $d^{tr}$ be the maximum transmission latency of a packet from one node to another in the same pod (resp. rack) when there is sufficient network bandwidth. Combining with the preceeding property, the end-to-end latency of a packet when being processed through $\mathcal{C}$ is at most $d = n \cdot T + (n-1) \cdot d^{tr}$, where $n$ is the number of components in $\mathcal{C}$. Thus, we must ensure that $d \leq D$.

The next lemma follows directly from these properties.

*Lemma 1:* Let $\mathcal{C} = C_1 \to C_2 \to \cdots \to C_n$ be a valid chain of $G$. An input period $T$ is feasible for $\mathcal{C}$ if

1) for all $1 \leq k \leq n$, $\mathsf{wcet}(C_k) < T$, and
2) the end-to-end latency of a packet of any request with traffic period $T$ when processed through $\mathcal{C}$ must be no more than $D$, i.e., $d = n \cdot T + (n-1) \cdot d^{tr} \leq D$.

Combine Lemma 1 with Definition 3, we have:

*Corollary 1:* Let $\mathcal{C} = C_1 \to C_2 \to \cdots \to C_n$ be a valid chain of $G$. An input period range $\mathcal{T}$ is feasible for $\mathcal{C}$ if the following two conditions hold:

1) For all $T \in \mathcal{T}$, $\max_{1 \leq k \leq n} \mathsf{wcet}(C_k) < T$.
2) $T_2 \leq (D + d^{tr})/n - d^{tr}$, where $T_2 = \max_{T \in \mathcal{T}} T$.

We then imply Corollary 2; its proof is available in [1].

*Corollary 2:* Suppose $T$ is a feasible input period for $\mathcal{C} = C_1 \to C_2 \to \cdots \to C_n$, then $[T, T_n^{\max}]$ is a feasible input period range for $\mathcal{C}$, where $T_n^{\max} = (D + d^{tr})/n - d^{tr}$.

### B. Interface computation algorithm

In the following we fix an application $G = (V, E)$ and introduce our algorithm for computing its interface. Based on Lemma 1, the fewer components an interface has, the larger its feasible input period. To enable the online scheduler to find the best allocation given the actual period of a request, we will compute for $G$ *a set of interfaces* with different lengths, each of which can be used to schedule requests with a certain traffic period range. Further, to minimize communication overhead, we limit the length of the chain to be at most $N$, where $N$ is the maximum number of NFs along any path in $G$.

For each $n \leq N$, we compute a valid chain of length $n$ and an associated feasible period range for the interface. Towards this, we will search for the smallest possible input period value $T_n^*$ for which there exists a valid chain $\mathcal{C}_n^*$ of length $n$ such that $T_n^*$ is feasible, using the conditions in Lemma 1. Once obtaining $T_n^*$, the interface $\mathcal{I}_n$ of length $n$ can be constructed from the corresponding chain $\mathcal{C}_n^*$ and the input period range $[T_n^*, T_n^{\max}]$, where $T_n^{\max} = (D + d^{tr})/n - d^{tr}$. Note that $\mathcal{I}_n$ is a feasible interface for $G$, since $[T_n^*, T_n^{\max}]$ is a feasible input period range for $\mathcal{C}_n^*$ (Corollary 2).

Algorithm 1 shows the procedure for constructing the chain $\mathcal{C}_n^*$ and the feasible period range $[T_n^*, T_n^{\max}]$ of the interface $\mathcal{I}_n$, for a given interface length $n$. Due to Condition

**Algorithm 1** Find an interface with a fixed length $n$.

1) Input: An application $G = (V, E)$, end-to-end deadline $D$, and the target interface length $n$.
2) Let $T_n^{\max} = (D + d^{tr})/n - d^{tr}$, and let $e_{\max}$ be the largest WCET among all NFs in $G$.
3) Binary search the period from the range $(e_{\max}, T_n^{\max}]$, denoted by $T$, to find the minimum period $T$ such that a valid chain for $G$ with period $T$ and length $n$ exists.
   *a*) In each step of the binary search, call Algorithm 2 to find the shortest valid chain for input period $T$.
   *b*) If the length of the computed chain is $\leq n$, search for a smaller $T$; otherwise, search for a larger $T$.
4) Denote by $T_n^*$ the (smallest) period found in the above binary search and by $\mathcal{C}_n^*$ its associated chain.
5) If $T_n^* \leq T_n^{\max}$ then return $\mathcal{I}_n = \langle \mathcal{C}_n^*, [T_n^*, T_n^{\max}] \rangle$; otherwise, return NULL.

**Algorithm 2** Find shortest valid chain given an input period

1) Input: an application $G$ and an input period $T$.
2) Create a source service (vertex) $s$ whose WCET is defined to be 0, and create an edge from $s$ to every vertex in $G$.
3) Let $\ell := 1$. Repeat the following while there are vertices left in $G$:
   a) Let $V_\ell$ be the subgraph of $G$ where each vertex $v \in V_\ell$ has the property that no path from $s$ to $v$ has the total WCET $\geq T$.
   b) Assign $C_\ell$ to be the subgraph of $G$ that consists of vertices $V_\ell$; assign $\mathsf{wcet}(C_\ell)$ to be the largest WCET of a path in $C_\ell$; remove $V_\ell$ from $G$; and $\ell := \ell + 1$.
4) Return $\ell - 1$ as the length of the CSC and the $C_1 \to C_2 \cdots \to C_{l-1}$ as the output chain.

[2] We assume a negligible non-zero overhead, but scheduling-related overhead such as cache inteference, context switches, etc. can be incorporated by inflating the WCET with such overhead.

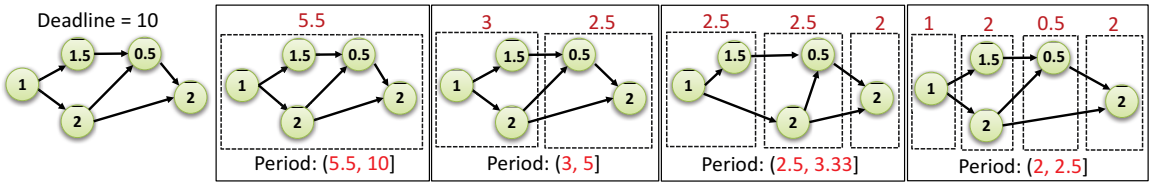



Fig. 4: An NFV application (left-most graph) and its interface table (with four interfaces, each enclosed by a solid rectangle). Here, we assume $d^{tr} = 0$ for simplicity. The number within each vertex of the graph represents the WCET of the corresponding NF. The number on top of each component (dotted rectangle) represents the component's WCET.

1 in Lemma 1, $T_n^* > \mathsf{wcet}(C_k)$ for all $1 \le k \le n$; thus, $T_n^* > e_{\max}$, where $e_{\max}$ is the maximum WCET of an NF in $G$. To compute $T_n^*$, we perform a binary search on the interval $(e_{\max}, T_n^{\max}]$. For each period value $T$ during this search, the algorithm uses Algorithm 2 as a subprocedure to compute the corresponding shortest valid chain of $G$. If the computed chain has a length larger than $n$, it will search for a larger value $T$; otherwise, it will search for a smaller value $T$. Intuitively, when $T$ is larger, the period/deadline can also be larger; hence, it is easier to group more NFs to a component, leading to a smaller length.

Algorithm 2 computes the shortest valid chain of $G$, given an incoming packet period $T$. Here, the construction of each $V_\ell$ ensures that the WCET of each component $C_k$ is no more than $T$, and for any edge $(s_i, s_j) \in E$, $\Pi(s_i) \le \Pi(s_j)$, where $\Pi$ is the corresponding mapping of the chain.

**Correctness and running time.** We now establish the correctness and running time of our algorithms. Intuitively, to minimize the length of the chain, Algorithm 2 always attempts to pack as many NFs into a component as possible, and this greedy strategy turns out to be optimal. Further, the running time of Algorithm 2 can be established by showing that the vertex sets $V_\ell$ in each iteration (Step 3.a) can be found in linear time, which can be achieved through a dynamic programming based approach. Due to space constraints, we omit the proofs.

*Lemma 2 (Correctness):* Algorithm 2 always outputs the shortest possible valid chain of $G$ for a given period $T$.

*Lemma 3 (Running time):* Algorithm 2 can be executed in $O(n+m)$ time, where $n$ is the number of vertices and $m$ is the number of edges in $G$.

Based on the above results, the overall running time for finding a feasible interface with a given length $n$ (i.e., Algorithm 1) is hence $O(d \cdot (n+m) \cdot \log W)$, where $d$ is the depth of the input DAG, $W$ is the number of different values (i.e., in the range $(e_{\max}, T_n^{\max}]$) that the incoming packet period could use, and $n$ and $m$ are the number of vertices and edges in $G$, respectively. The correctness of Algorithm 1 is followed by the correctness of finding the shortest valid chain (i.e., Algorithm 2).

**Interface table of $G$.** Using the output $\mathcal{I}_n$ of Algorithm 1 for all possible candidate interface lengths $n$, we construct a set of interfaces $\mathcal{I} = \{\mathcal{I}_n \mid 1 \le n \le N\}$ for $G$, which we refer to as the *interface table* of $G$. The $n$-th interface of $G$, $\mathcal{I}_n$, specifies a valid chain $C_n^*$ with length $n$ and an associated feasible period range $\mathcal{T}_n$. Figure 4 shows an example NFV application and its interface table.

In the next section, we describe how RTNF adapts its resource allocation to the varying period of an online request based on the application's interface table.

## VI. Online Interface Selection

Whenever an incoming request for an NFV application $G$ arrives, RTNF selects an interface from $G$'s interface table for the resource allocation for $R$ based on its traffic period. Specifically, let $T$ be the period of a new request $R$. For convenience, we say that a range $\mathcal{T}$ is larger (resp. smaller) than $T$ if for all $T' \in \mathcal{T}$, $T' > T$ (resp. $T' < T$). The concrete interface for $R$ is determined as follows.

**Case 1:** If $T$ falls in the feasible period ranges of some interfaces in $G$'s interface table, then we select the interface with the shortest chain among such interfaces (to minimize communication overhead), while setting $\mathsf{period}(C_k) = \mathsf{deadline}(C_k) = T$ for all $C_k$.

**Case 2:** If there exists some interface in the interface table whose period range is smaller than $T$, then we will choose the interface with the shortest chain among those; in addition, we set $\mathsf{period}(C_k) = T$ and $\mathsf{deadline}(C_k) = T_n^{\max}$, where $n$ is the number of components of the interface and $T_n^{\max} = (D + d^{tr})/n - d^{tr}$ is the maximum feasible period of the interface (c.f. Algorithm 1).

**Case 3:** Otherwise, $T$ must be smaller than every interface's feasible period range, and thus the request cannot be feasibly scheduled. If the request is splittable (i.e., stateless), we split its traffic flow into two different subflows with period $2T$ each, and then attempt to find the interface for each subflow in the manner as above. If the request is not splittable, however, we will reject the request.

**Example.** Consider the NFV application, and its interface table, shown in Figure 4. We denote by $\mathcal{I}_i$ the $i$-th interface (with $i$ components) of the interface table. Suppose we have three incoming requests $R_1$, $R_2$ and $R_3$, with periods $T_1 = 3$, $T_2 = 5.1$ and $T_3 = 1$, respectively.

Since $T_1$ falls into the feasible period range of only $\mathcal{I}_3$, we select this interface for $R_1$, while setting the period/deadline of its components to be equal to $T_1 = 3$.

Although no feasible period range contains $T_2$, the feasible period range $(3,5]$ of $\mathcal{I}_2$ is smaller than $T_2$; hence, we select this interface for $R_2$. In addition, the period and deadline of each component are set to be $T_2 = 5.1$ and 5, respectively.

| | |
|---|---|
| (1) $\forall k \in [1..n] : \sum_{m \in \mathsf{Nodes}; e \in \mathsf{Out}(m)} x_e^k = 1$ | (6) $\sum_{v \in \mathsf{EoR}; e \in \mathsf{Out}(v)} x_e^0 = 1$ and $\sum_{v \in \mathsf{EoR}; e \in \mathsf{In}(v)} x_e^0 = 0$ |
| (2) $\forall m \in \mathsf{Nodes} : \sum_{k \in [1..n]; e \in \mathsf{Out}(m)} x_e^k \cdot u^k \leq \mathsf{cpu}_m$ | (7) $\sum_{v \in \mathsf{EoR}; e \in \mathsf{In}(v)} x_e^n = 1$ and $\sum_{v \in \mathsf{EoR}; e \in \mathsf{Out}(v)} x_e^n = 0$ |
| (3) $\forall e \in \mathsf{Links} : \sum_{k \in [0..n]} x_e^k \cdot \beta^k \leq \mathsf{bw}_e$ | (8) $\forall k \in [1..n], \forall m \in \mathsf{Nodes} : \sum e \in \mathsf{In}(m) x_e^{k-1} = \sum_{e \in \mathsf{Out}(m)} x_e^k$ |
| (4) $\forall k \in [1..n] : \sum_{s \in \mathsf{EoR}; e \in \mathsf{In}(s)} x_e^k \leq 1$ | (9) $\forall k \in [1..n-1], \forall v \in \mathsf{EoR} : \sum_{e \in \mathsf{In}(v)} x_e^k = \sum_{e \in \mathsf{Out}(v)} x_e^k$ |
| (5) $\forall k \in [0..n-1] : \sum_{s \in \mathsf{EoR}; e \in \mathsf{Out}(s)} x_e^k \leq 1$ | (10) $\forall k \in [0..n], \forall v \in \mathsf{ToR} : \sum_{e \in \mathsf{In}(v)} x_e^k = \sum_{e \in \mathsf{Out}(v)} x_e^k$ |

TABLE I: Constraints for the online resource allocation.

As $T_3$ is smaller than all interfaces' feasible period ranges, we will reject it. One can validate that, since the maximum WCET of an NF in the application is 2, it is impossible to schedule any request with period less than or equal to 2 (see Section V-B), unless we split it into subflows.

The next lemma states the correctness of our interface selection. Its proof can be found in [1].

*Lemma 4:* If a request $R$ is not rejected, then the selected scheduling interface for $R$ guarantees that each component of the interface can be feasibly scheduled on a core, and that $R$ will meet its latency guarantee if all of the components in the interface are schedulable under the concrete period and deadline assigned by our selection strategy.

## VII. Online resource allocation

We now present our method for allocating resources to each new request $R$ based on the computed scheduling interface of $R$. Recall that the scheduling interface of $R$ specifies a chain of components $\mathcal{C} = C_1 \to C_2 \to \cdots \to C_n$, with concrete WCET, period, and deadline parameters for each $C_k$, such that $R$ meets its latency guarantee if all $C_k$ are schedulable. Hence, our allocation can be achieved by simply finding an assignment of the components to the nodes and a route between them, such that each $C_k$ is schedulable on its assigned machine and there is sufficient bandwidth for the data transfer from $C_k$ to $C_{k+1}$.

**Overview.** Our method first finds a pod, and then finds an assignment of $\mathcal{C}$ to this pod while maximizing locality. To find a pod, we maintain for each pod (a) the total available CPU bandwidth (i.e., the difference between the total number of cores and the total density of all active components on the nodes), and (b) the total available network bandwidth from EoR switches to ToR switches (downlink bandwidth) and from ToR switches to EoR switches (uplink bandwidth). For each pod, we calculate the fractions of the remaining CPU bandwidth, downlink bandwidth and uplink bandwidth if $\mathcal{C}$ is assigned to the pod. We then pick the pod with the smallest maximum value of the three fractions, to balance the network and computational resources across pods.

Once we have chosen a pod for $\mathcal{C}$, we find an assignment of $\mathcal{C}$ to the pod by formulating it as an integer linear program (ILP). To minimize energy consumption, we aim to minimize the number of active racks and thus restrict $\mathcal{C}$ to active racks only. If there exists no feasible assignment, we will activate a new rack and assign all components of $\mathcal{C}$ to the rack, or reject it if there is no such rack available. (Since an unused rack has much more resource than the demand of one request, an assignment can always be found.)

**ILP formulation.** For our formulation, we add a virtual link $(m)$ from each machine $m$ to itself, which is used by the transmission from $C_k$ to $C_{k+1}$ (denoted as $C_k \to C_{k+1}$) if both $C_k$ to $C_{k+1}$ are assigned to $m$. For ease of explanation, we include in $\mathcal{C}$ a (virtual) component $C_0$ before $C_1$, and $C_{n+1}$ after $C_n$. We use the following notations:

- $\mathsf{In}(v)$ and $\mathsf{Out}(v)$: sets of incoming and outgoing links of a switch $v$, respectively.
- $\mathsf{In}(m)$ and $\mathsf{Out}(m)$: sets of incoming and outgoing links, including virtual links, of a node $m$, respectively.
- $\mathsf{Links}$, $\mathsf{EoR}$, $\mathsf{ToR}$, $\mathsf{Nodes}$, and $\mathsf{Rack}_i$: sets of all network links, EoR switches, ToR switches, nodes, and nodes in the $i$-th rack, respectively.
- $\mathsf{bw}_e$: the available bandwidth of each network link $e$.
- $\mathsf{cpu}_m$: the available CPU bandwidth of each node $m$.
- $\beta^k$: the maximum bandwidth required by $C_k \to C_{k+1}$.
- $u^k$: the CPU bandwidth requirement (i.e., density) of $C_k$, i.e., $u^k = \mathsf{wcet}(C_k)/\mathsf{deadline}(C_k)$.

We define the binary variables $x_e^k \in \{0,1\}$ to indicate whether the link $e$ is used for the transmission $C_k \to C_{k+1}$. Their values determine both a route for the transmissions $C_k \to C_{k+1}$ and an assignment of $C_k$ to nodes: for any node $m$ and ToR switch $v$, if $x_{(m,v)}^k = 1$ or $x_{(m)}^k = 1$ – i.e., if the link $(m,v)$ or the virtual link $(m)$ is used for $C_k$'s outgoing traffic – then $C_k$ must be assigned to node $m$.

Table I shows the constraints of our ILP formulation. Constraint (1) specifies that each component $C_k$ is assigned to exactly one node (i.e., exactly one outgoing link from all nodes is used for $C_k \to C_{k+1}$). Constraint (2) specifies the schedulability condition for each node under EDF; for simplicity, we present here a sufficient condition if nodes are single core machines; extensions to multicores are discussed at the end of this section. Constraint (3) states that each network link $e$ should have sufficient available bandwidth for all transmissions that use $e$. Constraints (4)-(7) state that (i) each transmission $C_k \to C_{k+1}$ uses at most one incoming link and at most one outgoing link of all the EoR switches, (ii) $C_0 \to C_1$ uses exactly one outgoing link and no incoming link of all EoR switches; and (iii) the $C_n \to C_{n+1}$ uses exactly one incoming link and no outgoing link of all EoR switches. Constraints (8-10) establish that the links used by all $C_k \to C_{k+1}$ must form a valid route: (a) for all $k \in [1..n]$, if an incoming link of a node $m$ is used for $C_{k-1} \to C_k$ then an outgoing link of $m$ is used for $C_k \to C_{k+1}$; and (b) if an incoming link of a switch $v$ is used for $C_k \to C_{k+1}$ then an outgoing link of $v$ is also used for this transmission.

**Objective:** Our formulation aims to minimize the number of racks being used for the assignment, i.e.,

$$\text{minimize} \sum_{k\in[1..n-1];v\in\mathsf{EoR};e\in\mathsf{In}(v)} x_e^k$$

**Assignment of components to cores on each node.** The solution of the above ILP provides an assignment of components $C_k$ to nodes, and the links for the transmissions from $C_k$ to $C_{k+1}$. It always ensures that the request will meet its deadline if nodes are single core machines. Extensions to multicore cases can be done in two simple ways: (1) extending the ILP to include additional variables that indicate the assignment of components to cores, and to modify the schedulability condition (Constraint 2) to reflect the schedulability on each core (i.e., the available CPU bandwidth of each core is more than the total BW requirements of all components assigned to it); or (2) starting with the solution of the ILP, assigning the components placed on a node to the node's cores while considering the core-level schedulability, which is faster but not optimal. Our evaluation follows the first approach.

## VIII. Prototype Implementation

To test the efficacy of our system, we implemented both an event-driven simulator and a concrete prototype of RTNF.

**Simulator.** The simulation includes the entire resource provisioning algorithm (Sections V and VII) along with low-level network behavior and job scheduling. The packet-level simulator was implemented using 3510 lines of C++ code and invokes Gurobi [15] with 8 parallel threads as the ILP solver. The hypervisors for CPUs use EDF scheduling based on the budget (WCET) and deadline computed by RTNF. To cover network processing and propagation latencies, we measured the maximum latencies between two servers and between containers of a server in our testbed, and used the results ($150\mu s$ and $25\mu s$, respectively) in the simulator.

For comparison, we also implemented in our simulator two existing algorithms: E2 [27], a best-effort resource allocation framework for DAG-based NFV applications; and NFV-RT [21], a real-time resource allocation framework for NFV service *chains*. We also implemented the default Linux CFS scheduler, which is configured to be used for E2.

**Prototype.** We implemented a prototype of RTNF on real hardware. Our prototype uses a separate Docker container to execute the NFs of each component of the application's interface (determined by the controller for each incoming request). Each Docker container, which corresponds to a component of the application, has two virtual NICs (one input vNIC and one output vNIC). The vNIC of containers of an application are connected to each other by the pairs of virtual Ethernet devices (veth), which act like an Ethernet cable between network namespaces to connect two vNICs. The NFs within a Docker container are bound to both the input vNIC and loopback interface of the container: the former is to receive packets from a preceeding container or from a user, and the latter to receive packets coming from another NF in the same container. The NFs are bound to specific port numbers, thus an input packet can be routed to its destination NF based on its port number. (Details on chaining of containers in our prototype are available in [1].)

Our prototype contains 10280 LoC in C++ (3530 LoC for the controller, 2750 LoC for the run-time agent, and 4000 LoC for the request and traffic generator). For comparison, we also implemented E2 [27] and NFV-RT [21] on the testbed. The node's OS is configured to use SCHED_DEADLINE scheduler for RTNF and NFV-RT, and to use its default CFS scheduler for E2.

**Run-time overhead.** To evaluate the run-time overhead of RTNF, we measured the time taken to compute an allocation for each incoming request and to deploy the NFs using our prototype. We performed the experiments on our experimental testbed using real-world NFV workload and data center traffic traces (described in the next section). The results – obtained across all trials in our experiments – show that computing a resource allocation took less than $785\mu$s on average and less than $1762\mu$s for over 99% of the packets, and it took at most 4.1 ms even in the worst case. Deploying the NFs of a request took 3.1ms on average and 6.7ms for over 99% of the packets. This shows that RTNF incurs only minimal run-time overhead.

## IX. Evaluation

Our evaluation aims to answer four high-level questions: 1) Can RTNF provide latency guarantees for online requests of DAG-based NFV applications? 2) What is the overhead of running RTNF? 3) Can RTNF handle bursty workloads efficiently? and 4) How well does RTNF perform compared to existing solutions?

We evaluated the scalability and effectiveness of RTNF as large-scale NFV platforms through simulations and experiments on our prototype. The former allows us to explore a large parameter space for the platform and applications, such as a fat-tree topology with hundreds of machines. The latter enables us to evaluate the potential impact of practical overheads on performance in a real cloud environment. Both types of evaluation used real-world network traces and network functions.

### A. Workload

To obtain a representative experimental environment, we validated RTNF using a range of real-world network functions – including Snort, Load Balancer, Cache Proxy, NAT, Firewall, VPN, and Traffic Monitor – each with different configurations, resulting in 18 different NFs.

To obtain realistic and precise execution times of the NFs in the cloud setting, we profiled them on two sets of real-world data center traffic traces from [5]. These traces are well-known and commonly used by the networking community in research on network functions and middleboxes (e.g., [12, 13, 27, 38]). We replayed the traces, applied each NF, and recorded the processing time of each packet using the RDTSC instruction. We performed this profiling (100000 packets per NF) on every machine in our experimental testbed and measured the results across all machines. We

also analyzed the packet traces and extracted the packet rate, inter-arrival time and duration of the traffic flows. Based on the observed patterns, we generated online requests for our experiments. (More details can be found in [1].)

### B. Simulations

**Setup.** We simulated the network functions in a fat-tree datacenter topology in our simulator. The network contained 400 machines, each having 8 CPU cores. Every 10 machines formed a rack. Every network link had a bandwidth of 10 Gbps. Since RTNF first assigns requests to different pods (whose running time is negligible) and the assignment for different pods are independent of each other, we focused on evaluating the behavior of RTNF inside a single pod. To our knowledge, there is no public industrial data on NFV application structures, so we followed the standard approach and randomly generated a wide variety of DAG structures for our experiments. We generated 20 DAGs, each consisting of 4 to 8 network functions chosen randomly. The deadline of each DAG is chosen to be the WCET of (the services along a path of) the DAG plus a threshold $\Delta$, which is typically less than a few milliseconds ($\Delta = 2000\mu$s and $\Delta = 3000\mu$s in our simulations). The requests were generated to match real-world data center workloads as described earlier. Our experiments were performed on a workstation with 2.6 GHz Intel Xeon Gold 6142 processors (32 physical cores) and 376 GB of RAM. The OS was Ubuntu 16.04.

**Predictable latency guarantees.** To evaluate the effectiveness of RTNF in providing hard latency guarantees for online requests, we generated 10000 requests of different NFV graphs and measured the end-to-end latencies of the packets for all the 10000 traffic flows when being processed by RTNF. We repeated this experiment for 4 trials, and computed the *request miss rate* for each trial – the request miss rate is defined as the percentage of requests that missed their deadlines (i.e., that have at least one packet with a delay larger than the end-to-end deadline) out of all the requests. For comparison, we also performed the same experiments for the existing E2 resource allocation algorithm under two different settings: E2_10 and E2_100, with buffer thresholds set equal to 10 and 100 packets, respectively.

The results show that RTNF is able to provide a strong timing guarantee: all the requests always meet their deadlines across all trials. In contrast, E2 performs poorly in terms of meeting deadlines, where more than 71% and more than 76% of the requests missed their deadlines on average under E2_10 and E2_100 settings, respectively. These results confirm that techniques designed for throughput and average latency can severely suffer in terms of predictability.

**Average and tail latencies.** For each packet processed in the evaluation, we plot the total end-to-end delay divided by the deadline and the delay value in one random trial, presented as Figures 5a and 5b, respectively. (Results of other trials are similar.) We observe that RTNF gives a more stable packet latency compared to E2. In addition, it also reduces the average latency and 99th percentile tail latency by an order of magnitude across all trials: 165–1292× and 158–679× compared to E2_10, respectively; and 999–4283× and 984–2560× compared to E2_100, respectively. This is quite surprising, as E2 aims to optimize for the average case, and it also suggests that by making worst-case latency more predictable we also indirectly gain in terms of average and tail latencies.

**Resource consumption.** Finally, we recorded the total resource usages for one random trial over the period of the experiment. The results are shown in Figure 5c. We can observe that the resource consumption under RTNF is similar to that of E2; thus, we can conclude that RTNF is equally effective in optimizing resource use as E2 while giving substantially better latency performance.

We also observe from the results that the CPU demand in our simulation contains multiple spikes, which is consistent with the bursty nature of data center traffic [6, 18], and RTNF is able to dynamically provision more resources when demand bursts, while gradually scaling down the resource consumption afterwards (by scheduling most of the new incoming requests to active machines), and maintaining the platform resource in an energy efficient manner.

**Performance results for NFV chains.** Ideally, we would like to evaluate RTNF against existing NFV platforms that are designed for latency guarantees; however, the (only) existing platform we know of is NFV-RT [21], which only supports chains. For a fair comparison, we considered in this evaluation only chains of NFs. We generated NFV chains and requests in the same fashion as in the DAG case.

The results show that RTNF continues to provide strong latency guarantees for the requests, whereas E2 performs poorly, which is consistent to its observed performance for general DAG-based applications. Interestingly, we observed that a number of requests (0.075%) miss their deadlines under NFV-RT, even though in theory they should not; we expect that this is because NFV-RT does not consider the potential jitter between packets when it merges multiple traffic flows. Further, as shown in Figure 7, RTNF used substantially less resource than NFV-RT did (up to 5×). In short, not only is RTNF strictly more general than NFV-RT it also outperforms NFV-RT in both latency guarantees and resource usage.

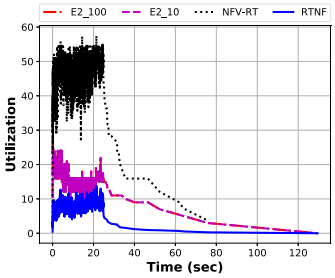
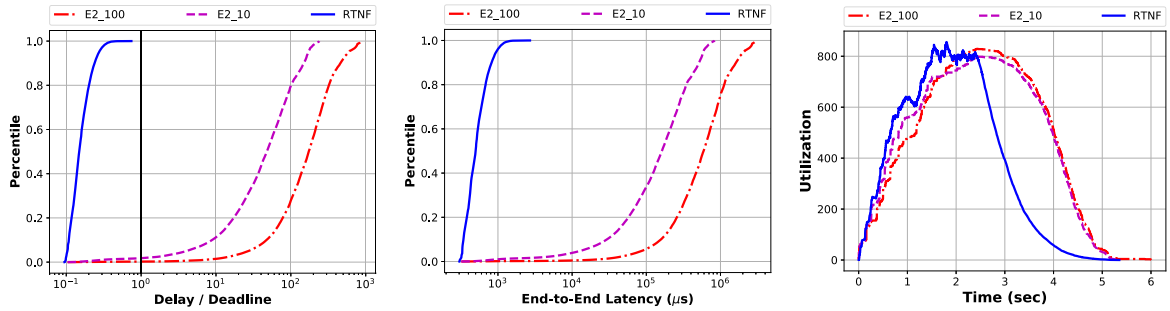


Fig. 7: NFV chains.

We also evaluated RTNF in soft real-time settings; due to space constraints, we provide the details in [1].

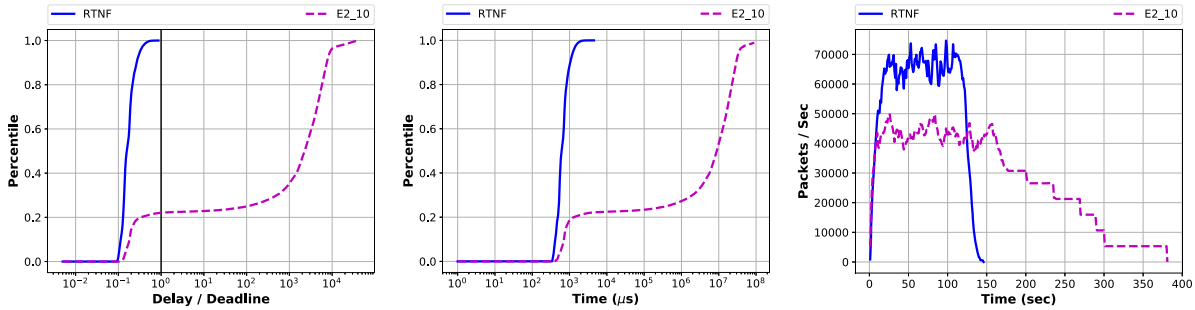
### C. Empirical evaluation

**Testbed experimental setup.** We evaluated RTNF using our prototype implementation on a local cloud testbed. The testbed contains four racks for executing NFs, which have a




(a) CDF of Delay/Deadline for packets. (b) Packet end-to-end latencies. (c) Resource usages.

Fig. 5: Simulation results for NFV graphs: Latency predictability, real-time performance and resource consumption.




(a) CDF of the latency/deadline of packets. (b) CDF of the packet latencies. (c) Throughput (number of packets/s).

Fig. 6: Empirical evaluation results for NFV graphs.

total of 96 cores across eight Intel Xeon E5-2620 v3 servers, each with 12 physical cores running at 2.40GHz and with 64 GB RAM. Machines in each rack are connected via a 1Gbps ToR switch, and all four ToR switches are connected to a 1Gbps EoR switch. To reduce interference, on each machine we disabled frequency scaling. We used one core for handling software interrupts (softirq), one core for the run-time agent, and the remaining cores for executing the network functions. These servers run Linux (Fedora 26) and Docker container software. In addition, the testbed has a separate rack connected to the EoR switch that has four 8-core Intel Xeon CPU E5-2630L servers (with 1.8 GHz CPU and 64 GB RAM), running Ubuntu 14.04 LTS. We used one server to run the controller, and the remaining three servers to generate packet flows for the users' requests. We configured the OS to use the SCHED_DEADLINE for RTNF and NFV-RT, and we used the default CFS scheduler for E2. We set the buffer threshold of E2 to be 10 packets.

We randomly created a set of 20 NFV application graphs in the same fashion as in the simulations. We generated a total of 2000 requests from across 20 users; the requests' traffic flows are created using the same two sets of real-world data center traffic traces as was done in the simulations.

**Predictable latency guarantee.** Figure 6a shows the request miss rate and the CDF of the ratios of the end-to-end latency divided by the deadline of the packets. (Note that the X-axis in the figure is in log scale.) The results show that RTNF can deliver solid timing guarantee in practice: all requests met their deadlines in our experiments. In contrast, E2 performs poorly in delivering timing guarantee: under E2, 83.25% of the requests missed their deadlines, and the majority of the packets (>64%) had an end-to-end latency that is 1000 times larger than their deadlines. We also evaluated RTNF against NFV-RT and E2 for NFV chains and observed similar results.

**End-to-end latency.** Figure 6b further shows the end-to-end latencies of the packets observed in our experiments for one trial (the results for other trials are similar). The results show that RTNF outperforms E2 by an order of magnitude in all metrics: average, 99.99th percentile, and maximum end-to-end latency. For instance, the observed packet latency under RTNF is always below 4.59 ms, whereas the packets can experience a latency of near 148 seconds under E2.

**Throughput.** Figure 6c plots the throughput of the system during the experiment duration. We can observe that RTNF achieves comparable throughput to E2. This demonstrates that through proper resource allocation, RTNF can deliver much better latency performance without impacting throughput as compared to existing techniques.

## X. Conclusion

We have presented RTNF, a scalable system for online resource allocation and scheduling of NFV applications on the cloud with latency guarantees. RTNF uses a novel consolidation technique for abstracting a complex DAG-based application into service chains with flexible timing interfaces, thus allowing for efficient resource allocation at run time. Our evaluation using real-world network traces and network functions shows that RTNF can provide solid timing guarantees with only a small overhead, and it performs substantially better than existing techniques.

## ACKNOWLEDGEMENT

This research was supported in part by ONR N00014-16-1-2195, NSF CNS 1703936, CNS 1563873 and CNS 1750158, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-16-C-0056 and HR0011-17-C-0047.

## REFERENCES

[1] S. Abedi, N. Gandhi, H. M. Demoulin, Y. Li, Y. Wu, and L. T. X. Phan. RTNF: Predictable latency for network function virtualization. Technical report, CIS Department, University of Pennsylvania, Feb 2019. http://www.cis.upenn.edu/~linhphan/papers/rtnf-long.pdf.

[2] H. Arabnejad, J. G. Barbosa, and R. Prodan. Low-time complexity budget-deadline constrained workflow scheduling on heterogeneous resources. *Future Generation Computer Systems*, 55(C):29–40, 2016.

[3] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[4] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. A generalized parallel task model for recurrent real-time processes. In *RTSS*, 2012.

[5] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *Computer Communication Review*, 40(1):92–99, 2010.

[7] A. Bremler-Barr, Y. Harchol, and D. Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *SIGCOMM*, 2016.

[8] R. N. Calheiros and R. Buyya. Meeting deadlines of scientific workflows in public clouds with tasks replication. *IEEE Transactions on Parallel and Distributed Systems*, 25(7):1787–1796, 2014.

[9] Z.-G. Chen, K.-J. Du, Z.-H. Zhan, and J. Zhang. Deadline constrained cloud computing resources scheduling for cost optimization based on dynamic objective genetic algorithm. In *CEC*, 2015.

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

[11] Y. Gao, Y. Wang, S. K. Gupta, and M. Pedram. An energy and deadline aware resource provisioning, scheduling and optimization framework for cloud systems. In *CODES+ISSS*, 2013.

[12] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *CoRR*, 2013.

[13] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *SIGCOMM*, 2014.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[15] Gurobi Optimization, Inc. http://www.gurobi.com.

[16] J. Hwang, K. K. Ramakrishnan, and T. Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *NSDI*, 2014.

[17] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea. Performance contracts for software network functions. In *NSDI*, 2019.

[18] J. W. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang. Joint VM placement and routing for data center traffic engineering. In *INFOCOM*, 2012.

[19] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *SIGCOMM*, 2008.

[20] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *SIGCOMM*, 2017.

[21] Y. Li, L. T. X. Phan, and B. T. Loo. Network functions virtualization with soft real-time guarantees. In *INFOCOM*, 2016.

[22] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *SC*, 2011.

[23] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *NSDI*, 2014.

[24] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *EMSOFT*, 2007.

[25] C. Nicutar, C. Paasch, M. Bagnulo, and C. Raiciu. Evolving the internet with connection acrobatics. In *HotMiddlebox*, 2013.

[26] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *ATC*, 2013.

[27] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A framework for NFV applications. In *SOSP*, 2015.

[28] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki. Automated synthesis of adversarial workloads for network functions. In *SIGCOMM*, 2018.

[29] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized Zero-Queue Datacenter Network. In *SIGCOMM*, 2014.

[30] Z. A. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using SDN. In *SIGCOMM*, 2013.

[31] X. Qin and H. Jiang. A dynamic and reliability-driven scheduling algorithm for parallel real-time jobs executing on heterogeneous clusters. *Journal of Parallel and Distributed Computing*, 65(8):885–900, 2005.

[32] B. P. Rimal and M. Maier. Workflow scheduling in multi-tenant cloud computing environments. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):290–304, 2017.

[33] J. Sahni and D. P. Vidyarthi. A cost-effective deadline-constrained dynamic scheduling algorithm for scientific workflows in a cloud environment. *IEEE Transactions on Cloud Computing*, 6(1):2–18, 2018.

[34] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *SIGCOMM*, 2012.

[35] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, 2012.

[36] G. L. Stavrinides and H. D. Karatza. Scheduling real-time dags in heterogeneous clusters by combining imprecise computations and bin packing techniques for the exploitation of schedule holes. *Future Generation Computer Systems*, 28(7):977–988, 2012.

[37] M. Stigge, P. Ekberg, N. Guan, and W. Yi. The digraph real-time task model. In *RTAS*, 2011.

[38] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu. Nfp: Enabling network function parallelism in nfv. In *SIGCOMM*, 2017.

[39] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker. Resq: Enabling slos in network function virtualization. In *NSDI*, 2018.

[40] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problem–overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[41] F. Wu, Q. Wu, and Y. Tan. Workflow scheduling in cloud: a survey. *The Journal of Supercomputing*, 71(9):3373–3418, 2015.

[42] J. Yu, R. Buyya, and C. K. Tham. Cost-based scheduling of scientific workflow applications on utility grids. In *eScience*, 2005.

[43] Y. Yuan, X. Li, Q. Wang, and X. Zhu. Deadline division-based heuristic for cost optimization in workflow scheduling. *Information Sciences*, 179(15):2562–2575, 2009.

[44] P. Zave, R. A. Ferreira, X. K. Zou, M. Morimoto, and J. Rexford. Dynamic service chaining with dysco. In *SIGCOMM*, 2017.