Network Functions Virtualization with Soft Real-Time Guarantees

Yang Li, Linh Thi Xuan Phan and Boon Thau Loo
University of Pennsylvania

***Abstract*—Network functions are increasingly being commoditized as software appliances on off-the-shelf machines, popularly known as Network Functions Virtualization (NFV). While this trend provides economics of scale, a key challenge is to ensure that the performance of virtual appliances match that of hardware boxes. We present the design and implementation of NFV-RT, a system that dynamically provisions resources in an NFV environment to provide timing guarantees. Specifically, given a set of service chains that each consist of some network functions, NFV-RT aims at maximizing the total number of requests that can be assigned to the cloud for each service chain, while ensuring that the assigned requests meet their deadlines. Our approach uses a linear programming model with randomized rounding to efficiently and proactively obtain a near-optimal solution. Our simulation shows that, given a cloud with thousands of machines and service chains, NFV-RT requires only a few seconds to compute the solution, while accepting three times the requests compared to baseline heuristics. In addition, under some special settings, NFV-RT can provide significant performance improvement. Our evaluation on a local testbed shows that 94% of the packets of the submitted requests meet their deadlines, which is three times that of previous reactive-based solutions.**

I. Introduction

Network functions are being commoditized as software appliances on off-the-shelf machines, popularly known as *Network Functions Virtualization (NFV)* [5]. NFV enables elastic scaling and pooling of resources in a cost-effective manner and can be deployed in a more agile fashion than traditional hardware based appliances. Example network services include functionalities for routing, load balancing, deep-packet inspection, and firewalls [4], [16]. While this approach provides economics of scale, a key challenge is to ensure that the performance of these virtual appliances matches that of traditional hardware boxes. This is because, unlike dedicated hardware appliances, virtual network appliances are deployed in virtual machines (VMs) on commodity servers, e.g., in the cloud.

Current solutions are either heuristics-based (e.g., auto-scalers on public clouds) which provides no guarantees, or reactive in nature, where the cloud operator is alerted only after the SLAs have been violated. To address this challenge, this paper presents NFV-RT, a platform for composable cloud services with soft real-time guarantees. NFV-RT dynamically provisions resources in an NFV environment to provide packet-wise timing guarantees to service requests. Specifically, we make the following contributions:

Mathematical modeling and analysis. We formulate the resource provisioning problem that NFV-RT has to solve with a mathematical model. Given a set of service chains that each consist of some network functions, NFV-RT aims at maximizing the total number of requests that can be assigned to the cloud for each service chain, while ensuring that the assigned requests meet their deadlines. Our approach uses *service chain consolidation with timing abstraction* and a linear programming technique with randomized rounding to proactively obtain a near-optimal solution. This approach scales well for large data center deployments, and it can optimize real-time performance even in online scenarios where new NFV chains are added on demand.

Implementation and evaluation. To evaluate our approach, we have implemented a simulator for NFV-RT. Our simulation results show that, for a cloud with thousands of machines and thousands of service chain requests, NFV-RT took only a few seconds to compute the solution, while accepting three times more requests than a baseline heuristics does. The results also demonstrate that under some special settings, NFV-RT can provide significant performance improvement. In addition, we have developed a prototype implementation of NFV-RT, based on RT-Xen [22], a real-time virtualization platform built upon Xen. Our evaluation on a local 40-core testbed shows that NFV-RT enabled 94% of the packets of the submitted requests to complete before their deadlines, which is three times compared to the baseline solution.

II. Cloud Model

We consider a cloud provider that supports many cloud tenants, each of whom runs one or more NFV service chains in the cloud, servicing traffic on behalf of her customers. The goal of the cloud provider is to develop a resource provisioning strategy that maximizes the number of requests with SLA guarantees (meeting deadlines) while ensuring isolation among tenants. Towards this, we aim to find an assignment such that (i) the total number of accepted requests of all tenants is maximized, (ii) the delay experienced by each packet of an accepted request does not exceed its relative deadline, and (iii) services of different tenants cannot execute in the same VM. In this paper, we focus on single-path flows through the service chains; enabling multi-path flows is an avenue for future work.

NFV-RT is a system for resource provisioning that the cloud provider can use to meet the above goal. We begin by presenting a mathematical model used by NFV-RT.

Our model considers a typical *fat tree* [10] network topology popularly used in data centers, where each node v denotes a switch or a rack of machines, and each edge (v, v') denotes a network link connecting v and v'. Figure 1 shows a three-layer fat tree: the top, middle, and bottom layers are made of sets of core switches, end-of-row switches (EoR), and racks of machines (M), respectively. Each core switch is connected to all the EoRs and serves as a point of presence (PoP) of the cloud. All NFV traffic must enter and leave through the PoPs.

Pod. The cloud has multiple disjoint *pods* that are connected through core switches (e.g., two pods in Figure 1). Each pod contains a number of EoRs and their connected racks.

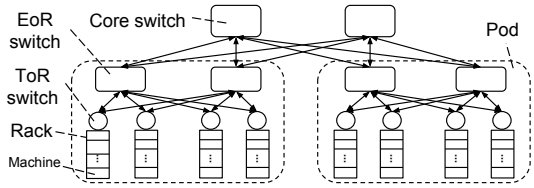


Fig. 1. An example fat tree topology.

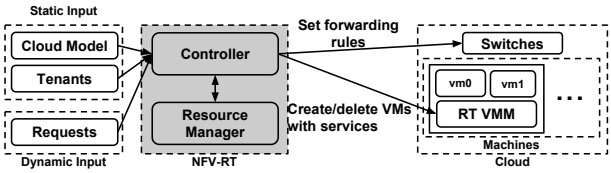


Fig. 2. An overview of NFV-RT.

Rack. Each rack contains a top-of-rack switch (ToR) and several physical machines that each run a virtual machine monitor (VMM). We model the collection of machines within a rack m as an aggregated node with cpu_m cores, where cpu_m is the total number of cores of the machines in the rack m.

Given a set of predefined executable services of an NFV service chain, we denote by $wcet_s(L)$ the worst-case execution time (WCET) that a service s takes to process a packet of size L. (The WCET of a service can be obtained using well-known WCET analysis methods [20].) As the size of the output packet of a service can differ from that of the input packet (e.g., in the case of compressing/decompressing or encryption/decryption services), we denote by γ_s the scaling function of s: if L is the size of an input packet of s, then the size of its corresponding output packet is at most $\gamma_s(L)$.

Tenant. The profile of a tenant i is defined as $tenant_i(v_i^s, v_i^t, \langle s_i^1, s_i^2, \ldots, s_i^{c_i} \rangle, d_i)$, where v_i^s and v_i^t are the core switches through which the traffic enters and leaves the cloud, respectively; $\langle s_i^1, s_i^2, \ldots, s_i^{c_i} \rangle$ is the service chain (of length c_i) that the tenant intends to route all her customers' traffic through; and d_i is the relative deadline, which is the longest tolerable packet-wise end-to-end delay. The traffic demand of a customer of a tenant is called a *request*. A request is a tuple $request_i(tid_i, \alpha_i)$, where tid_i is the identifier of the tenant, and α_i is the maximum packet rate (packets/s). NFV-RT considers one service chain per tenant, but it can easily be extended to allow more service chains per tenant.

For ease of presentation, we assume the same packet size for all requests of a tenant. The size of each incoming packet of $tenant_i$ is denoted by L_i^0, and the size of the output packet after traversing the first j services of the service chain of the tenant is denoted by L_i^j, i.e., $L_i^j = \gamma_{s_i^j}(L_i^{j-1})$ for all $j > 0$.

For each incoming request, NFV-RT needs to spawn VMs on the physical machines to execute the services. An *assignment* of a request from $tenant_i$ involves assigning (a) one or more VMs, and their corresponding machines, that execute every service of the tenant, and (b) a path starting from v_i^s, passing through the VMs that execute the services, following the order of the service chain, and ending with v_i^t. If a request is accepted, an assignment for the request must be given. An assignment of a set of requests is made of an assignment of every request in the set.

III. Overview of NFV-RT

Figure 2 shows an overview of NFV-RT. It contains two interacting components: (i) the controller, which is responsible for communicating with customers of the cloud's tenants and for performing the deployment of the services on the cloud based on a resource assignment; and (ii) the resource manager, which is responsible for determining an assignment for new requests, and for keeping track of the current status of the cloud and the current assignment of existing requests.

The controller takes two sets of inputs: (1) the cloud model (c.f. Section II) and a set of tenants, and (2) a set of customers' requests, which may arrive dynamically at run time. It then passes these inputs to the resource manager, which will perform the request admission test and compute an assignment for the accepted requests. The controller will then deploy the computed assignment on the cloud (e.g., communicates with the VMM in each machine to create and configure VMs, and sets up network paths for the accepted requests).

To enable real-time guarantees, each machine of the cloud runs a real-time VMM that supports real-time scheduling at both VMM and guest OS levels (e.g., RT-Xen [22]). NFV-RT assumes that the VMM scheduler and the guest OS within each VM follow the Earliest Deadline First (EDF) scheduling policy [11] (which is supported by existing VMM implementations [22]), so as to achieve high resource utilization. However, our system can easily be extended to other scheduling policies.

The resource manager works in two phases, as illustrated in Figures 3(a) and 3(b). In the initial phase, it performs the service chain consolidation and timing abstraction to minimize the communication and VM overhead, based on which it then determines an assignment for an initial set of requests. In the online phase, it dynamically determines a new assignment for each new set of requests as they arrive at run time. In the next two sections, we describe these two phases in greater detail.

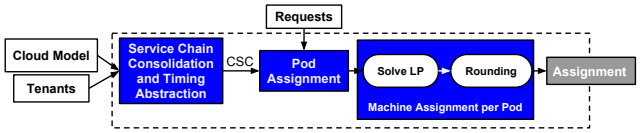
IV. Initial resource provisioning

As shown in Figure 3(a), the initial assignment consists of three consecutive stages: (1) Service request consolidation, (2) Pod assignment, and (3) Machine assignment per pod. We describe these stages in detail below.

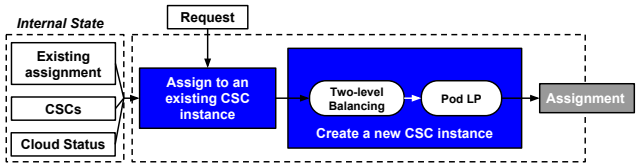
A. Stage 1: Service Request Consolidation

Overview. In this stage, we consolidate the service chain of each tenant[1] into a *consolidated service chain* (CSC) and abstract its resource requirements into a *timing interface*. The CSC contains the same services as the original service chain does, but adjacent services may be merged together to form a *consolidated service* to be executed on a single VM. To reduce the VM overhead, our consolidation aims to minimize the number of consolidated services. The CSC timing interface gives the conditions on the incoming packet rate and the CPU resource required to ensure each instance of the CSC meets the deadline. Based on this information, we can send requests

[1]To ensure isolation among tenants, two tenants cannot share the same VM; hence, consolidation is done for each tenant individually.



(a) Initial assignment phase



(b) Online assignment phase

Fig. 3. NFV-RT resource provisioning method.

to a CSC instance if they satisfy the packet rate condition, and we can assign the consolidated services to machines if the machines can satisfy the required CPU resource.

Specifically, the CSC timing interface is made of two parts: (i) the CPU resource required by each consolidated service, given in the form of $(period, budget)$, which specifies that this consolidated service requires *budget* execution time units every *period* time units; and (ii) the maximum packet rate, i.e., the maximum number of packets that can be sent through each instance of the CSC every second.

The CSC satisfies the following property: *"If the traffic of some requests goes through a CSC instance, then the packets of the requests can meet their deadlines if* (a) *the total packet rate sent through this instance is at most the CSC's maximum packet rate,* (b) *the VMs that execute the consolidated services of the instance are each given the required CPU resource, and* (c) *there is sufficient bandwidth between the VMs that execute any two adjacent consolidated services."*

Figure 4 shows an example service chain with three services $\langle s_1, s_2, s_3 \rangle$. It is consolidated into a CSC with two consolidated services, $[s_1, s_2]$ and $[s_3]$. The CSC timing interface specifies that (i) $[s_1, s_2]$ requires a budget of 1.5ms (equal to its WCETs, shown within the box) of CPU time every 2ms, and $[s_3]$ requires 2ms of CPU time every 2ms; and (ii) this CSC can receive a maximum packet rate of 500 packets/s.

After the service consolidation, the requests of each tenant are packed into a number of aggregated requests (based on the requests' packet rates) that will be assigned to different instances of the CSC. NFV-RT will then determine an assignment for these CSC instances. By construction, every aggregated request that is accepted and receives an assignment will meet its packet-wise deadline, assuming that the scheduling overhead is negligible. We next discuss the consolidation and service aggregation in detail.

1) Conditions for Consolidation and Abstraction: Consider a $tenant_i$, and denote the maximum packet rate of its CSC by cap (# packets/sec). Then, the CSC should satisfy three conditions: (1) *Feasibility condition*: Each consolidated service requires at most one core; (2) *Traffic condition*: The value of cap is at most one tenth of the smallest link bandwidth of the cloud; and (3) *Deadline condition*: If we take the packet arrival period (i.e., $1/cap$) as the maximum delay for executing

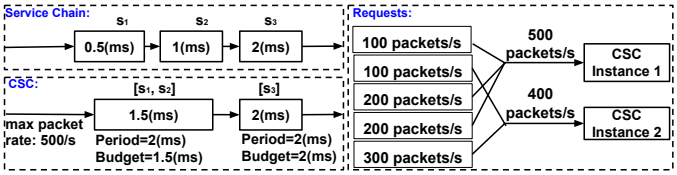


Fig. 4. An example of service request consolidation. The number within each service represents its WCET.

each service of the CSC (instead of the WCET), then the total delay for executing all consolidated services and transmitting a packet between subsequent consolidated services is at most the tenant's deadline. We now discuss each condition in detail.

Feasibility Condition. Every consolidated service of a CSC instance of the tenant will be executed by a unique VM. Like in existing real-time systems research [11], we assume that a VM can run on only one core at any time. Therefore, the CPU utilization of each consolidated service must be no more than 1 to feasibly schedule it on a core, i.e.,

$$\max_{j=1}^{c_i} \left\{ cap \times wcet_{s_i^j}(L_i^{j-1}) \right\} \leq 1. \tag{1}$$

In Eq. (1), s_i^j denotes the j-th consolidated service, $wcet_{s_i^j}(L_i^{j-1})$ denotes its WCET (for each input packet, which is of size L_i^{j-1}), and thus its utilization is $wcet_{s_i^j}(L_i^{j-1}) \times cap$.

Traffic Condition. We impose a soft constraint on the maximum amount of traffic that can traverse a CSC instance, based on the intuition that it is easier to find small chunks of available bandwidth than to find a large one. NFV-RT uses one tenth of the smallest link bandwidth of the cloud as the upper bound on the traffic rate, i.e.,

$$\max_{j=0}^{c_i} (cap \times L_i^j) \leq \frac{\min_{e_k} \{b_k\}}{10}, \tag{2}$$

where b_k is the bandwidth of each link e_k.

The above bound is an empirical bound that comes from the analysis of the Linear Programming (LP). Essentially, this bound is to ensure that, after we obtain a (fractional) solution of the LP, randomly rounding the obtained solution into an integer assignment will output an almost optimal result. Due to space constraints, we omit the details here.

Deadline Condition. We first briefly explain our estimation of the maximum delay a consolidated service takes to process a packet. Recall from Section III that, to enable predictable delay, each machine runs a real-time VMM, such as RT-Xen [22], which schedules VMs on the physical cores under EDF. NFV-RT will create VMs with designated periods p_j and budgets b_j, where p_j and b_j are specified by the interfaces of the consolidated services. In other words, each VM j requests the CPU for at most b_j execution time units every p_j time units, where $0 < b_j \leq p_j$. Due to this, when multiple VMs share the same core under EDF, if the total utilization of all the VMs is no more than one (i.e., $\sum_j b_j/p_j \leq 1$), every VM j is guaranteed to receive at least b_j execution time units every p_j time units [11]. Since the b_j CPU time units can be given towards the end of the period, each considolated service may experience a worst-case delay of up to p_j time units.

Algorithm 1 A dynamic program for determining the shortest consolidated chain

1) Input: $\langle s^1, s^2, \ldots, s^c \rangle$ with function $wcet(i, j)$, which is the WCET for executing the sequence of services $\langle s^i, s^{i+1}, \ldots, s^j \rangle$, and period T.
2) Define function $f(i)$ as the length of the shortest chain for executing services $s^1, s^2, \ldots, s^i$. Define $f(0) = 0$.
3) For i from 1 to c: $f(i) = \min_{j \in [0,i-1] \ \wedge \ wcet(j+1,i) \leq T} \{f(j) + 1\}$
4) Output $f(c)$ and the corresponding CSC.

We now establish the deadline condition. For each $tenant_i$, if the maximum packet rate of the CSC is cap, and the length of the CSC is l, then when using the period $(1/cap)$ as the delay of each service, the end-to-end delay of a packet will be $d^{tr} + (1/cap + d^{tr})l$, where d^{tr} is the maximum delay for transmitting a packet between any two machines in the same pod. (To minimize transmission delay, each CSC instance is always assigned to the same pod; however, our formulation can easily be modified to relax this condition.) Note that the adjacent services in the chain can be merged and executed in the same VM if their total CPU utilization is no more than one. Moreover, after merging the services, the number of consolidated services decreases, which in turn decreases the total estimated delay for executing the consolidated services, as well as the total transmission delay (since packets traversing services executing within the same VM do not need to go through the network). Hence, to ensure that the delay is at most the deadline d of the service chain, our goal is to find the largest cap such that there exists a way to merge the adjacent services to obtain a CSC (with length l) such that the end-to-end delay is at most d, i.e.,

$$d^{tr} + (1/cap + d^{tr})l \leq d. \tag{3}$$

2) Algorithms for Consolidation and Abstraction:
We now present an algorithm for finding the maximum cap and the corresponding CSC based on the deadline condition (Eq.(3)), using an upper bound derived from the feasibility and traffic conditions (Eqs. (1) and (2)). We use Algorithm 1 as a sub-procedure, which takes any given cap and returns the shortest length of the consolidated service chain.

The shortest length of CSC. Formally, the problem of finding an optimal way to consolidate a service chain for a fixed value of cap can be stated as follows: Given a service chain $S = \langle s_i^1, s_i^2, \ldots, s_i^{c_i} \rangle$ with period $T(= 1/cap)$, partition S into disjoint segments $S_1, S_2, \ldots S_k$ with the smallest k such that, for any S_x, the WCET of the sequence of services[2] in S_k is less than the period. In other words, if we denote by $wcet(i, j)$ the WCET of the sequence of services $s_i, s_{i+1}, \ldots, s_j$, then for any segment S_x that starts from s_i and ends with s_j, $wcet(i, j) \leq T$.

To solve this problem, we propose a dynamic program (Algorithm 1), which finds the smallest partition for every sub-chain that starts with s^1. When computing for the sub-chain from s^1 to s^i, which is of length $f(i)$, the algorithm enumerates all possible ways to partition the chain into two sub-chains, where the second sub-chain can form a single segment (the WCET less than the period). Since the smallest partition of the first sub-chain is stored by function f, it suffices to simply store the size of the smallest partition among all such bi-partition (see Algorithm 1 for details). It is straightforward to see that the algorithm is optimal and takes $O(c^2)$ time to compute, where c is the length of the original service chain.

[2]The WCET of a sequence of services is not always equal to the sum of the WCETs of the individual services (for instance, services that share the same procedure for parsing the packet, see [15]).

Algorithm 2 Find the max cap for deadline condition

1) Input: $\langle s^1, s^2, \ldots, s^c \rangle$ with function $wcet(i, j)$, which is the WCET for executing the sequence of services $\langle s^i, s^{i+1}, \ldots, s^j \rangle$.
2) For l from 1 to c:
   (a) Binary-search the packet rate to find the maximum packet rate such that the corresponding shortest service chain has length at most l.
   - In each step of the binary search, use Algorithm 1 to find the shortest CSC. If the length is less than or equal to l, search for a larger α; otherwise, search for a smaller α.

   (b) Assume α^* is the largest packet rate, if $d^{tr} + (1/\alpha^* + d^{tr})l \leq d$, list α^* as a candidate, and drop it otherwise.
3) Among all the candidates, pick cap to be the largest among them.

With the above dynamic program for computing the length of the shortest possible consolidated service chain, we can now design the algorithm to determine the maximum cap that satisfies the deadline condition (Algorithm 2). The algorithm enumerates all possible lengths (from 1 to c) for the CSC, and for each length, it performs a binary search to find the corresponding maximum cap. One can easily show that the algorithm always output the optimal cap, and the time it takes is $O(c^3 \log \alpha_{max})$, where α_{max} is the largest possible value of cap, which can be obtained from the feasibility condition (Eq. (1)) and the traffic condition (Eq. (2)).

After obtaining the maximum value of cap that satisfies all three consolidation conditions, the actual CSC can be directly found using Algorithm 1. For the CPU resource requirement of each consolidated service, the period is equal to $1/cap$ and the budget is equal to the WCET of the consolidated service.

3) Packing Requests into Aggregated requests:
After obtaining the CSC for each tenant, we will pack the requests of a tenant into a set of aggregated requests that each can be executed using an instance of the CSC. In other word, the total packet rate of each aggregated request should be at most the maximum packet rate cap. Observe that this is a standard bin packing problem, where each request is an item with size equal to its packet rate, and each aggregated request is a bin with size cap. There are well-studied algorithms for solving the bin packing problem efficiently with good approximation ratios that we can use. For instance, the First Fit Decreasing (FFD) algorithm [8] guarantees to find a solution using at most $71/60\ OPT + 1$ bins, where OPT is the minimum number of bins needed to pack all the items.

After consolidating the services and requests, we explain how to assign them to the cloud in the next two stages. We will use the following slightly abused notation for an aggregated request: $request_i(v_i^s, v_i^t, \langle s_i^1, s_i^2, \ldots, s_i^{c_i} \rangle, \alpha_i, \#requests_i)$, where v_i^s is the starting node, v_i^t is the ending node, $\langle s_i^1, s_i^2, \ldots, s_i^{c_i} \rangle$ is the consolidated service chain, α_i is the packet rate, and $\#requests_i$ is the total number of original requests that this aggregated request contains. We denote by β_i^j the traffic rate

Algorithm 3 ILP for assigning requests to pods

$$\min \ \lambda \quad \text{s.t.} \ \forall i, \sum_j x_{i,j} = 1, \quad \forall i,j, x_{i,j} \in \{0,1\} \tag{1}$$

$$\forall j, \sum_i x_{i,j}\beta_i^{in} \le \lambda \times b_j^{in}, \quad \forall j, \sum_i x_{i,j}\beta_i^{out} \le \lambda \times b_j^{out} \tag{2}$$

$$\forall j, \sum_i x_{i,j} u_i \le \lambda \times podCPU_j \tag{3}$$

after traversing the first j services, and $wcet_i^j$ the WCET per packet of the j-th (consolidated) service. We also denote β_i^{in} and β_i^{out} as the incoming and outgoing traffic rates, respectively (i.e., $\beta_i^{in} = \beta_i^0$ and $\beta_i^{out} = \beta_i^{c_i}$).

B. Stage 2: Pod Assignment

This stage is essentially a pre-assignment, which aims to evenly split the resource demands of all requests of all tenants into different pods. In the next stage, we will determine the actual assignment for each pod individually.

Assigning requests to pods is a multidimensional bin packing problem, which can be formulated as an integer linear program (ILP). Towards this, we compute for each pod_i the maximum total bandwidth available from core switches to racks and from racks to core switches (using a standard max-flow algorithm), and denote them by b_i^{in} and b_i^{out}, respectively.

Denote the total number of CPU cores of pod_i by $podCPU_i$. For each $request_i$, denote the CPU utilization of its jth service by u_i^j. Then, the total CPU utilization of the request is $u_i = \sum_{j=1}^{c_i} u_i^j$. For any $request_i$ and any pod_j, define the binary variable $x_{i,j}$ to indicate whether $request_i$ is assigned to pod_j, and denote λ as the resource utilization. We formulate the request assignment as the ILP in Algorithm 3.

Since solving an ILP can be inefficient, we use an LP with rounding approach to achieve better scalability. For this, we let the variables $x_{i,j}$ to be chosen within the range $[0,1]$ (instead of integers $\{0,1\}$), and solve the LP version. For any $request_i$, we choose a random number and assign it to pod_j with the probability $x_{i,j}$ obtained by the LP solution. One can show that this LP with rounding technique produces a solution with objective value (λ) increased by a factor of at most $(1+\delta)$, for some small $\delta > 0$, compared to that of the ILP solution. (Due to space constraints, we omit the details of the analysis, but a similar analysis can be found in [13]).

C. Stage 3: Machine Assignment

To assign a list of requests to the machines in a pod, we again formulate the assignment as an ILP, then solve the LP version and apply rounding to obtain an almost optimal solution. In the following, all operations are done for each pod individually.

1) Graph Replication for the ILP formulation: We use a graph replication technique to formulate the assignment as an ILP. In the following, we denote by c^* the length of the longest service chain of all requests.

We create (c^*+1) replications of the pod, denoted by $G_0, G_1, \ldots, G_{c^*+1}$, and the path assigned to a service chain, $\langle s^1, s^2, \ldots, s^c \rangle$, will be broken into sub-paths for the sub-chains $\langle s^1, s^2 \rangle$, $\langle s^2, s^3 \rangle$, $\ldots$, where the jth sub-path is assigned using G_j. For each rack m, let m_j be the replication of m in G_j. We create an edge from m_j to m_{j+1} (for each j), denoted by e_j^m. Hence, a path from the first replication to the cth replication can be translated into an assignment for the request and vice versa. We will formulate the ILP for finding such a path.

Algorithm 4 ILP for assigning requests on each pod

$$\max \sum_i \left(\#requests_i \cdot \sum_{e \in out(v^{EoR})} x_{i,e,0} \right)$$

$$\text{s.t.} \ \forall i, \sum_{e \in out(v^{EoR})} x_{i,e,0} \le 1, \quad \sum_{e \in in(v^{EoR})} x_{i,e,0} = 0 \tag{1}$$

$$\forall i, j \in [1, c_i - 1], \sum_{e \in in(v^{EoR})} x_{i,e,j} = \sum_{e \in out(v^{EoR})} x_{i,e,j} \tag{2}$$

$$\forall i, j, m, \ x_{i,m,j-1} + \sum_{e \in in(m)} x_{i,e,j} = x_{i,m,j} + \sum_{e \in out(m)} x_{i,e,j} \tag{3}$$

$$\forall e, \sum_{i,j} x_{i,e,j}\beta_i^j \le b_e, \quad \forall m, \sum_{i,j} x_{i,e^m,j} \cdot u_i^j \le cpu_m \tag{4}$$

$$\forall i, j, e, x_{i,e,j} \in \{0,1\} \tag{5}$$

To minimize communication overhead, we forbid a path from using any core switches apart from its starting and ending switches, v^s and v^t, and we consider the assignment without v^s and v^t. Once the sub-assignments between the services are determined, the complete assignment is achieved by simply connecting v^s and v^t to the beginning and the end, respectively.

Graph Abstraction. To reduce the complexity of solving the LP that finds a path in the replicated graph, we perform abstraction: Use an aggregated EoR to abstract all physical EoRs, and call it v^{EoR}. The edge between the aggregated EoR and a rack is defined to have bandwidth equal to the total bandwidth between this rack and all physical EoRs. Thus, after the abstraction, each pod becomes a tree of depth 2.

2) ILP Formulation: We now give the ILP formulation for finding the path in the (abstracted) graph. For any node v (which is either a rack or the aggregated EoR), let $out(v)$ and $in(v)$ denote the set of edges that start from and end with v, respectively. For each $request_i$ and each edge e, we define the binary variables $x_{i,e,j}$ to indicate whether the replication of e in G_j is used in the path for $request_i$. With a slight abuse of notation, for each rack m, we use $x_{i,m,j}$ to indicate whether the edge e_j^m is used in the path for $request_i$. The problem can be formulated as the ILP shown in Algorithm 4.

The first three constraints are to ensure that the assignment of $request_i$ is a valid path starting from v^{EoR} in G_0 and ending with v^{EoR} in G_{c_i}. Constraints (4) assert there are enough bandwidth on every link and enough CPUs on every rack. As usual, we relax $x_{i,e,j}$ to be in the range $[0,1]$ and solve the LP version (in this case, the solution is a *network flow*).

3) Rounding: To transform the fractional solution into an integer assignment, we use random rounding to obtain a list of integer solutions and pick one of them. For any $request_i$, we assign it with the path that starts from the edge $e \in out(v^{EoR})$ with probability $x_{i,e,0}$; following the edges, at each node choose an outgoing edge with probability proportional to the $x_{i,e,j}$ values; and repeat until reaching the ending v^{EoR}.

Note that the paths obtained from the above rounding are paths in the abstracted graph, and VMs are assigned only

to racks instead of to actual machines. To obtain an actual assignment, we use the following two extensions:

Aggregated EoR to physical EoR. When the path goes through the v^{EoR}, we will randomly pick a physical EoR to transform this assignment from the abstracted graph to the real cloud. Since the traffic rate of the each path is no larger than one tenth of the edge bandwidth (traffic condition), it can be shown using Chernoff inequality that with high probability, almost all edges (at least $1-\delta$ fraction, for some small $\delta > 0$) will be assigned with enough bandwidth in the original cloud.

Packing VMs. In order to determine which machine of the rack that a VM should be assigned to, for each rack, we pack all VMs that are assigned to it into machines, i.e., using utilization as key and the number of available cores of the machines as bins, and perform FFD to determine the fraction of VMs that cannot be packed.

Among all the paths generated by the random rounding, we pick one where (i) bandwidth on edges is almost satisfied, (ii) for each rack, the VMs assigned to it can be almost packed, and (iii) the number of accepted requests is almost as large as the solution of LP, where 'almost' always means $(1+\delta)$ fraction, for some small $\delta > 0$. (To guarantee the traffic not exceeding the bandwidth, one only needs to divide the original bandwidth by $(1+\delta)$ and use the result as the input to the solver.) Using Chernoff bound, it can be shown that such a path exists and can be found if rounding is performed for sufficiently many times. In our evaluation, we observe that performing rounding for 20 times is sufficient.

V. Online Assignment

As shown in Figure 3(b), the resource manager maintains as its internal state the existing set of CSCs obtained in the initial phase, the existing assignment (i.e., the existing set of CSC instances and their current assignments), and the current cloud status (available bandwidth and CPU utilizations). After the initial phase, NFV-RT will dynamically perform the online assignment as new requests arrive at run time. Like in the initial phase, NFV-RT packs the requests into CSC instances and then assigns the instances to the cloud. It works in two stages, as follows.

A. Stage 1: Assign to existing CSC instances

Given each new request R, NFV-RT first checks all existing CSC instances of the same tenant to find one that has enough slack between its current packet rate and its maximum packet rate (cap) to fit the request in. If such a CSC instance exists, R will be accepted and directly assigned to that instance.

If no such CSC instance exists, NFV-RT will 'reshuffle' the existing CSC instances using bin packing formulation (which can be solved using existing bin packing algorithms, such as FFD), as follows: The bins are the CSC instances, and the items are the requests of the tenant. The capacity of the bin is the maximum packet rate of the CSC, and the size of an item is the packet rate of the corresponding request. If all requests of the tenant, including the new request R, can be packed using only the existing CSC instances, we accept R and migrate the requests according to the output of the bin packing solution (if needed). Note that this migration only involves re-routing the packet flows of the existing requests of the same tenant but not migrating the services.

B. Stage 2: Create a new CSC Instance

If it is infeasible to assign the new request using only the existing CSC instances, NFV-RT will create a new CSC instance for the new request and assign it to the cloud.

To assign the new instance to the cloud, NFV-RT attempts to find an assignment that results in balanced resource for different pods and for different racks in each pod, by using a two-level balancing strategy. If no such assignment exists, NFV-RT will perform *Pod LP* to re-balance the resource usage of the existing assignment.

Two-level balancing. We define the in-bandwidth and out-bandwidth of a pod as the bandwidth from the core switches to its racks and from its racks to the core switches, respectively. We select a pod using the following method, which we call first level balancing: first, iterate through all pods, and for each pod, calculate the fractions of the in-bandwidth, out-bandwidth, and cores that are remained, respectively, if the new CSC instance is assigned to the pod; then, pick the pod with the smallest maximum value of the three fractions.

After choosing the pod, we select a rack in the pod to assign the entire new CSC instance. This is done using a second level balancing, as follows. We iterate through each rack and perform two steps: (i) calculate the fractions of CPU cores in the rack and the bandwidth of links adjacent to the rack that are remained, respectively, if the new CSC is assigned to the rack, and (ii) find a path with sufficient available bandwidth for the CSC instance from the starting core switch, passing through the rack, to the ending core switch. More specifically, for each rack rk, the step (ii) is done by finding two EoR switches, s_1 s_2, such that the links (v^s, s_1, rk, s_2, v^t) have sufficient bandwidth for the new CSC instance. Among the racks for which a path can be found, we pick the one with the smallest maximum value of the CPU and bandwidth fractions.

If no assignment can be found, we will perform VM or traffic migration to better organize the resources in the chosen pod, i.e., perform a re-assignment of existing CSC instances. This is done through the following Pod LP.

Pod LP. We use the ILP in Algorithm 4 with the following modifications: In Constraint (1), replace $\sum_{e \in out(v^{EoR})} x_{i,e,0} \leq 1$ with '$= 1$', to ensure that all requests will get an assignment. Define a new variable γ as a ratio reflecting the most congested link (most loaded rack). Then, in Constraint (4), replace $\sum_{i,j} x_{i,e,j} \beta_i^j \leq b_e$ with '$\leq \gamma b_e$', and $\sum_{i,j} x_{i,e^m,j} u_i^j \leq cpu_m$ which '$\leq \gamma cpu_m$'. Finally, change the objective to "minimizing γ."

We solve the obtained ILP using the LP with rounding technique, as was done in Section IV-C3. By the design of the new ILP, the output assignment will minimize rack load and link congestion. After re-balancing the requests' assignment in the pod, we use the two-level balancing heuristics again to find an assignment for the new request. If no assignment can be found, the request will be rejected.

VI. Evaluation

We evaluated the performance and scalability of NFV-RT using both simulation and on an actual testbed. We implemented a prototype of NFV-RT with all of its functionalities to perform resource provisioning for NFV requests. We used Python to implement both the controller and the resource manager of NFV-RT. For the LP solver, we used Gurobi [6] with 32 parallel threads.

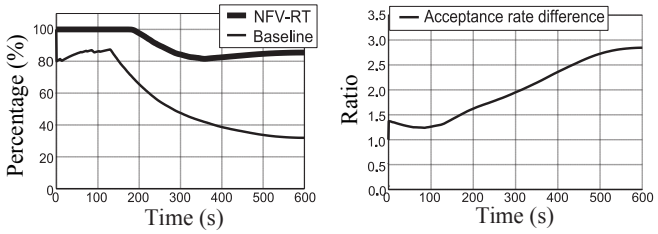
Within each machine, RT-Xen [22] is used as the real-time VMM, running on Ubuntu Server 12.04, 64-bit. Although RT-Xen only supports VM budget specified in millisecond granularity (period is in microsecond), it is sufficient for the purpose of illustration, and we simply round up the budget to the nearest millisecond. Software network bridges in Linux are set up for the communication between VMs. After the resource manager has determined an assignment, the controller uses RPC to create VMs on the machines and to configure both the services to execute and the next hop to forward packets.

We compared our approach against a greedy algorithm as our baseline. The greedy strategy constantly monitors the status of the cloud to detect resource bottlenecks; if a VM with high CPU utilization (over 85%) is found, it splits the load and spawns a new VM, then attempts to place the new VM in the same rack as the overloaded VM. If the rack is full, it tries the racks that are two-hop away (i.e., racks in the same pod), before considering any other racks. In addition, for simulation, whenever a request arrives at run time, it checks whether assigning this request to the cloud using the planned assignment will cause a resource bottleneck; if so, it immediately attempts to spawn a new VM to avoid the bottleneck (instead of reactively responding after a bottleneck is observed). If no location can be found for spawning the new VM, the request will be rejected.

A. Simulation

Setup. We used a 16-core machine to simulate a cloud setup with the same topology as shown in Figure 1. The setup contains 1600 machines, with each having 4 CPU cores, and every 40 machines form a rack. The cloud is divided into 10 pods, with 4 racks and 2 EoR switches each. There are 4 core switches that are shared by all pods. Every link in the cloud has a bandwidth of 10Gb/s.

There are ten services that the cloud can execute, each with WCET randomly picked from the range $[5,50]$ μs. There are 50 tenants, each with a service chain of length at most 10 and a deadline between 5ms and 10ms. The starting and ending cores switches were randomly picked. A request of a tenant was generated by randomly choosing packet rate from 1000 to 4000 packets/s. Hence, the largest traffic rate is about 50Mbps. Once the requests are assigned to pods, the actual assignment for each pod is independent of each other; hence, the machine assignment for the pods is fully parallelizable. Therefore, the execution time of NFV-RT consists of (1) the time to pack the requests into CSC instances, (2) the time to assign CSC instances to different pods, and (3) the maximum time for the machine assignment among all pods.



(a) Single trial acceptance rate (b) Aggregated performance

Fig. 5. Simulation for real time performance

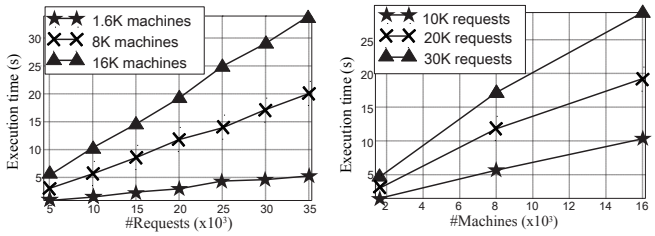
Efficiency of LP with rounding. We evaluated NFV-RT with 30K requests and 10 randomly generated test cases, where we recorded both the execution time for finding the assignment and the number of accepted requests (the acceptance rate). The results show that our LP rounding based resource manager is highly efficient: the average execution time of NFV-RT is less than 5s, which is orders of magnitude better than the traditional ILP based solutions, which can take 1800s [14] for the offline stage. In addition, NFV-RT is also effective in utilizing the resource: it accepts about 75% of the requests, and almost all links adjacent to the core switches are fully utilized, i.e., no more requests can be assigned to the cloud.

Real-time performance. We next evaluated the online real-time performance of NFV-RT and the baseline strategy (where requests arrive dynamically at run time). For this, we chose a 10-minute interval and generated 60K requests (i.e., about 100 new requests every second). The total traffic rate of each tenant formed a bimodal distribution over time, so as to capture the bursty nature of data center network [7], [2]. Specifically, for each tenant, we selected two time slots as the traffic peaks, and for each request, we randomly chose one peak, generated a number x from the normal distribution $\mathcal{N}(0,150)$, and finally, let the starting time and ending time be the peak time $\pm x$. We generated 10 test cases and constantly recorded the current acceptance rate (the number of accepted requests divided by the number of requests so far).

Acceptance rate. Figure 5(a) shows the results for a single trial, where the x-axis represents time, and the y-axis gives the current acceptance rate. Observe that NFV-RT always outperforms the baseline, and the performance improvement increases with time. This is expected, since long running requests accumulate in the cloud over time, and it becomes more difficult for a greedy strategy to fully utilize the cloud.

Figure 5(b) shows the aggregated results of 10 trials. The y-axis gives the acceptance rate of NFV-RT divided by that of the baseline, averaged over 10 trials. As shown in the figure, NFV-RT accepts about 3 times the requests compared to the baseline. Note that the smaller improvement at the beginning is expected, because the cloud had a lot of resources available and the baseline was consuming resources aggressively. When the resources in the cloud are almost saturated (at around 100s), the baseline performance began to drop substantially. In contrast, NFV-RT started to demonstrate its ability to schedule requests efficiently under limited resource availability.

The aggregated results also show that the online assignment of NFV-RT incurs only small overhead (less than 0.5ms if Pod



(a) Vary the number of requests (b) Vary the number of machines

Fig. 6. Simulation for scalability

LP for each pod is invoked at most once every second, which allows us to process over a thousand requests per second).

Schedulable rate. The design of NFV-RT guarantees that the packets of every accepted request will meet their deadlines. In contrast, without considering the timing constraints, the greedy baseline does not have such a guarantee: although our simulation ensures that CPUs and links are never overloaded, accepted requests might still miss their deadlines. Specifically, let the *schedulable rate* be the percentage of requests that meet their deadlines. Then, in Figure 5(a), the acceptance rate of NFV-RT is also its schedulable rate, whereas the acceptance rate of the baseline is an upper bound of its best possible schedulable rate. Further, the performance improvement of NFV-RT over the baseline in terms of schedulable rate is at least equal to that was shown in Figure 5(b).

With or without Pod LP. We evaluated NFV-RT with and without Pod LP. We observed that, in the same setting, having Pod LP does not offer significant improvement. However, there are scenarios where the benefit of Pod LP can be arbitrary large, such as the following. Consider a pod with two racks, rk_0 and rk_1. At time 1, a large number of CPU-intensive requests arrive, which use up all CPUs in both racks. At time 2, all requests assigned to rk_0 finish and thus, rk_0 becomes fully available whereas rk_1 has no CPU left. At time 3, a large number of bandwidth-intensive requests arrive, each of which needs very little CPU resource. Without using Pod LP (i.e., no migration), NFV-RT can only use rk_0 to assign these requests. However, the new assignment given by Pod LP allows requests from rk_1 to migrate to rk_0, making the bandwidth on the links adjacent to rk_1 available for the new requests (which is previously unusable as rk_1 has no CPU left). Theoretically, the gain could be made arbitrarily large if we set the CPU requirement of the new requests to be arbitrarily small. We investigated one case in this setting, and we observed that Pod LP enables over 25% more requests to be accepted.

Scalability. We evaluated the scalability of NFV-RT by considering larger cloud topologies. We consider respectively 1.6K, 8K, and 16K machines by increasing the number of pods in the cloud and keeping the size and the topology of each pod unchanged. Note that the execution time in this evaluation is for the initial phase of NFV-RT, and when entering the online phase, the overhead of NFV-RT for scheduling requests is negligible (less than 0.5ms). We varied the number of requests from 5K to 35K. Figures 6(a) and 6(b) show the execution time when varying the number of requests and the number of machines, respectively. The execution time grows linearly in terms of the number of requests, but grows sightly less than linear in terms of the number of machines. This is because the complexity of LP depends on both the number of machines and the number of edges, and scaling up the topology will increase both of them. In all cases, the initial phase is completed within 35 seconds, even for cloud with 16K machines.

B. Actual Testbed

We evaluated NFV-RT in a local cluster consisting of 40 cores in total across 4 physical machines hosting VMs. An addition physical machine serves as the traffic generator. The traffic of a request will always start from the generator, travel through a list of VMs, and be sent back to the generator. The generator will record the sending time and receiving time of every packet to compute the packet's delay.

VMs. Two of the VM hosts have 4 Intel(R) Xeon(R) 2.40GHz CPU cores with 12GB RAM, and the other two have 16 Intel(R) Xeon(R) 2.10GHz cores (32 core threads with hyper-threading) with 24GB RAM. The 16 core machines have hyper-threading enabled, and hence, we are able to get $32\times$ parallelism. All VMs run Ubuntu Server 12.04, 64-bit with 256MB RAM. One core is reserved for the VMM in each 4-core machine, and two cores are reserved for the VMM in each 16-core machine, all with 2GB RAM. We run 4 VMs on each of the quad-core machines, and 32 VMs on each of the 16 core machines, for a total of 72 VMs, of which 66 are used for running the service chains.

Network. On our testbed, we generated delays by running the machines across different racks. All three racks (more specifically, the two 16-core machines and the switch that connects the two 4-core machines) and the traffic generator are connected by another switch. The local cloud is viewed as a single pod, with each link having 1Gb/s bandwidth.

Services. We implemented two types of services in C: firewall and network address translation (NAT). The firewall and NAT will both attempt to match the source IP of a packet with a list of pre-defined rules, and, respectively, mark a specific bit of the packet and change the source IP to a different IP when a match is found. Each service has a parameter specifying the number of rules it needs to match. For instance, FW50000 and NAT100000 stand for firewall with 50K rules and NAT with 100K rules, respectively. We used four different services: FW50000 (WCET 2.5ms), FW100000 (WCET 5ms), NAT50000 (WCET 2.5ms), and NAT100000 (WCET 5ms). The WCET was estimated by measurement [20].

Requests. We generated 5 tenants, three of whom have service chains of length 1 and the remaining two have service chains of length 4. The service chains are chosen randomly with no repeated service for each tenant. The deadline of each tenant is the sum of the WCETs of its services plus a random number chosen between 10ms and 20ms. Each tenant has 10 requests, and the packet rate of each request is randomly chosen from the range $[50, 100]$ (i.e., the maximum data rate is about 1.2Mb/s). All requests arrive at the beginning and last for 2 minutes. In other words, we created a network burst at time zero, and examined the performance.

Figure 7 shows the CDF of the packet delay. The x-axis represents the delay/deadline ratio, and the y-axis represents the percentage of the packets that meet their deadlines. For each algorithm, the point with x-value equal to 1 indicates the percentage of packets that meet their deadlines (the actual numbers are listed in the legend).

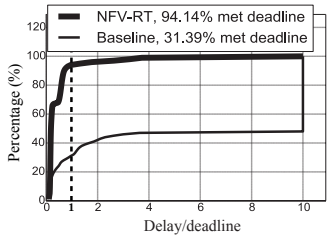


Fig. 7. The delay/deadline CDF

The results show that NFV-RT provides timing guarantees for more than 94% of the packets, which is approximately three times that can be guaranteed by baseline (31.39%). We further observed that the packets that missed their deadlines under NFV-RT were either caused by network outliers or happened near the time that the socket was just created.[3] We also observed that the CDF under the greedy baseline is stabilized at a percentage value of less than 50%, i.e., over 50% of the packets have arbitrarily long delays. We investigated the data and confirmed that these packets were lost and never received by the generator. In contrast, the maximum packet delay under NFV-RT is always bounded, and only a very small fraction (about 1%) of the packets have delays larger than 4 times their deadlines. Finally, NFV-RT takes only a few milliseconds to find an assignment in our experiments.

VII. Related Work

Existing work in NFV resource management does not consider the dynamic deployment of services in a general virtualized cloud setting with real-time constraints. For instance, PLayer [9] and SIMPLE [14] are effective "traffic steering" tools for routing traffic flows between static middleboxes (MBox), but they do not consider virtualization or dynamic MBox placement. Similarly, CoMb [15] considers dynamic MBox placement but in a special setting where the routing path is fixed, and its goal is simply to determine which machines on the path should be chosen to execute the MBoxes.

Stratos [3] monitors the cloud to detect resource bottlenecks and responses accordingly by duplicating or migrating VMs. However, the 'reactive' nature of Stratos leads to poor delay guarantees (e.g., when a bottleneck is detected, some deadlines may have been missed). In contrast, NFV-RT proactively assigns the resources to services based on a formal analysis, thus enabling better timing guarantees and performance predictability. NFV-RT can enable millisecond-level delay guarantees, which is not possible under such a reactive approach.

There exists an extensive literature in cloud resource management (e.g. [18], [1], [19]) but they target non-real-time applications. Techniques for virtual machine placement and migration have also been developed, e.g., [21], [12], [17]. However, these techniques do not simultaneously solve the VM placement and the traffic steering problem, which in general can lead to sub-optimal solutions. We are not aware of any existing work that can provide formal timing guarantees for real-time services in the cloud environments.

[3]Incoporating these overheads will be considered in our future work.

VIII. Conclusion

We have presented the design, implementation and evaluation of NFV-RT, a real-time resource provisioning system for NFV. NFV-RT integrates timing analysis with several novel techniques, such as service chain consolidation, timing abstraction, and linear programming with rounding, to enable efficient resource provisioning while ensuring timing guarantees. Our evaluation using both simulation and emulation shows that not only NFV-RT is effective in meeting deadlines and offers significant real-time performance improvement compared to a baseline approach, but it is also highly efficient and scalable.

Acknowledgement

Supported in part by ONR N00014-13-1-0802, ONR N00014-16-1-2195, and NSF grants CNS-0845552, CNS-1117185, ECCS-1135630, CNS-1218066, CNS-1329984, CNS-1505799 and the Intel-NSF Partnership for Cyber-Physical Systems Security and Privacy.

References

[1] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea. Chatty tenants and the cloud network sharing problem. In *NSDI*, 2013.
[2] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *Computer Communication Review*, 2010.
[3] 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.
[4] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward software-defined middlebox networking. In *HotNets*, 2012.
[5] R. Guerzoni et al. Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper. In *SDN and OpenFlow World Congress*, 2012.
[6] I. Gurobi Optimization. Gurobi optimizer reference manual, 2014.
[7] 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.
[8] D. S. Johnson and M. R. Garey. A 71/60 theorem for bin packing. *Journal of Complexity*, 1(1):65 – 106, 1985.
[9] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *SIGCOMM*, 2008.
[10] S. Kandula, S. Sengupta, A. G. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *IMC*, 2009.
[11] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.
[12] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM*, 2010.
[13] R. Motwani and P. Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.
[14] Z. A. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using SDN. In *SIGCOMM*, 2013.
[15] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *NSDI*, 2012.
[16] 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.
[17] V. Shrivastava, P. Zerfos, K. Lee, H. Jamjoom, Y. Liu, and S. Banerjee. Application-aware virtual machine migration in data centers. In *INFOCOM*, 2011.
[18] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, pages 349–362, 2012.
[19] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.
[20] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.
[21] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.
[22] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. D. Gill, O. Sokolsky, and I. Lee. Real-Time Multi-Core Virtual Machine Scheduling in Xen. In *EMSOFT*.