A Scalable Network-Aware Framework for Cloud Monitoring Orchestration

Masoume Jabbarifar [a], Alireza Shameli-Sendi [b,*], Bettina Kemme [a]
[a]*School of Computer Science, McGill University, Montreal, Canada*
[b]*Faculty of Computer Science and Engineering, Shahid Beheshti University (SBU), Tehran, Iran*
Email: jabbarifar@cs.mcgill.ca, a_shameli@sbu.ac.ir, kemme@cs.mcgill.ca

***Abstract*—Monitoring components are implanted in clouds to evaluate performance, detect the failures, and assess component interactions by message analysis. In this paper, we propose Monitoring as a Service (MaaS) installed across its software defined network in the cloud. All switches in datacenter only forward traffic. Specific SDN application on top of the network controller has been implemented in order to orchestrate multiple network tenants monitoring needs. Applications declare the required observation for traffics and their specific monitoring needs to the Maas. Therefore, a set of virtual monitoring functions (vMF) are prescribed to be placed in datacenter for flows. An optimal placement algorithm places vMF with respect to the network and computing utilization maximization objectives. Network objective refers to minimize traffic delay for flows needed to be monitored and computing objective expresses balancing nodes computing resources. The optimal placement of virtual network functions is known to be an NP-hard problem.**

Compared to the existing work, we discovered different problem which how a vMF can be split into smaller pieces to decrease the total placement cost for a set of flows required, based on four patterns: *free*, *parser-collocation*, *job-collocation*, and *full-collocation*. Moreover, we proposed three heuristics to make our placement algorithm scalable for a large network, called, *chain partitioning*, *topology partitioning*, and *zoning*. We show the feasibility of our approach in a large dataset consists of 540k nodes and 11.5M edges, with about 40 requests (flows) at the same time. Furthermore, the proposed solution saves at least 20 percent of total monitoring cost, in average, compared to the latest related work.

***Index Terms*—Cloud computing, SDN, NFV, Service chaining, Monitoring functions, Optimal placement.**

1. Introduction

The task of monitoring applications that are deployed in the cloud is a major process for both cloud providers and cloud customers [1]. It is a fundamental service that is needed for performance tuning: to determine performance bottlenecks [2], faults or anomalies [3], to determine how many resources are actually needed to handle the current workload, or to ensure correct behavior after reconfigurations [4], [5].

Many applications deployed in the cloud are distributed in nature as they consist of different modules and tiers, that are distributed across virtual machines and physical nodes. Consider the load balancing architecture configuration for a distributed application as depicted in Figure 1 [6] where a front-end load-balancer distributes requests across two application servers that both might access a main-memory cache and/or a database backend. Thus, client requests lead to a workflow of service calls, where one component calls upon services of some of the other modules.

Monitoring such a complex application is non-trivial [5]. So far, cloud providers offer limited monitoring services, including, for instance, the resource utilization of the virtual machines such as CPU, main memory, I/O and network bandwidth [7]. Apart of this, application developers have to rely on middleware-based monitoring frameworks or even implement, integrate and deploy their own monitoring functions. This is tedious and often ad-hoc, and requires significant expertise and holistic knowledge of the entire application architecture. Often, it starts with measuring throughput and end-to-end response time, and only when performance anomalies are found, further measurements are instantiated.

Therefore, we believe it would be interesting for cloud providers to offer more advanced off-the-shelf monitoring functionality to large-scale customers, some form of *Monitoring as a Service (MaaS)*. This might not only have the potential to bring monetary benefits to the cloud provider but also help the cloud provider itself to tune its system configuration [8], [9].

But in order to be more holistic and useful, it is not sufficient to offer services at the individual virtual machine level (CPU, memory, etc.) or the physical node level [10]. Instead, we would like to argue that, given the significant in-cloud network traffic and data exchange these complex applications produce, offering *monitoring functionality at the network traffic level* could be extremely beneficial to understand inter-dependencies between modules, detect bottlenecks, control security, and detect vulnerabilities. Obvious examples that should be easy to measure at the network level, are bandwidth, jitter, number of packages, package

- *Alireza Shameli-Sendi (Corresponding author)

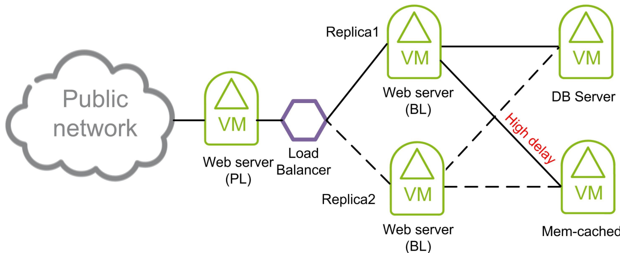


Figure 1: A typical component-based application architecture.

sizes etc [11]. These could be measured for any given network flow between two components by intercepting the application messages. A change in any of these metrics could indicate load changes, bottlenecks or misconfigurations. More complex functions could determine the delay between two switches or between two components. Yet more interesting could be to determine response time at the network level if one would be able to detect request/reply message pairs. In the example of Figure 1, for instance, response time of the cache could be really bad due to misconfiguration. Thus, monitoring resource utilization will not be useful, but spikes in response times can nevertheless determine where the problem resides [5]. Even more advanced pattern analysis could determine relationships and dependencies between consecutive messages. In fact, one could even envision that application specific information can be extracted at the network layer as long as the network has the capability to inspect messages in depth.

The interesting aspect is that with the advancements in virtualization technologies and the elasticity offered by the cloud, network level monitoring can be provided in form of virtualized functions implemented in software [12], [13]. Moreover, they can be dynamically instantiated, automatically deployed, and transparently inserted to inspect the traffic thanks to the programmability power of Software Defined Networking (SDN) [14], [15], [16]. As SDN controllers allow dynamic routing of packets, flows can potentially be redirected, e.g., they can be routed through virtualized network functions [17]. At the current state, these virtualized network functions typically have to reside on an end host. With this setting they could be instantiated on a host in the same rack of either sender or receiver and then the switch of this rack could be programmed to forward copies of the messages to the host of the monitoring function [18]. However, as switches become more and more powerful, in particular higher up in the network hierarchy, we believe it will become feasible to deploy more general software modules on the switches themselves, offering the potential to intercept, monitor and analyze message flows anywhere in the network between sender and receiver [19]. Then, the SDN updates all flow tables for the selected optimal routes. In this paper, we are looking at such an architecture, and analyze how such smart switches can be used to provide advanced monitoring functionality at the network layer.

Overall, we envision the cloud provider to offer a set of specialized monitoring functions (MFs), each implementing a particular functionality in terms of monitoring capacity. The simplest ones, e.g., can count the number of individual messages or sum the bandwidth per time unit.

These MFs can be instantiated on the switches of the cloud's network [19]. If an application requests that a given flow is monitored, the cloud provider can, in principle, deploy the corresponding MFs on any switch on the path between the flow's endpoints. However, it is not straightforward to decide which would be the best switch. An additional important aspect is that the cloud provider, by using SDN, does not only have a choice of where in the network to deploy the MFs but is also the one that decides on the exact path each individual flow takes [20]. This means, it can actually force a flow to go through a particular switch if this switch is suitable to host MFs [21].

In this paper, we propose a solution to this placement problem. Given a set of application flows that need to be monitored, our solution determines the following: (i) It decides through which switches to direct each of these flows and (ii) It determines where on these flow paths to deploy the MFs that are required.

There are several things to consider. First, we believe that collocation of MFs onto a few switches is beneficial as common functionality can then be shared, such as installing and running the appropriate runtime environment, parsing messages, etc. That is, our assumption is that the overall execution costs of the MFs (in terms of CPU) will be smaller if collocated on the same switch compared to when they are distributed across many switches. For instance, as one extreme solution all monitoring functions could be put on one very powerful switch. However, such a solution requires all flows to be forced to go through that switch, which is likely to increase overall network traffic and individual path lengths. At the other extreme, one could first determine optimal paths for all flows, and then deploy the required monitoring functions somewhere on these paths. However, collocation might then not always be possible, making monitoring potentially more costly. Therefore, our solution aims at finding a good trade-off between keeping the overall costs of monitoring low and having short paths for each of the flows.

Overall, the paper makes the following contributions:

- We motivate the usefulness of monitoring at the network level and discuss its potential implementation as a service. We outline possible schemes of how MFs can be collocated in order to reduce overall processing costs. One considers to collocate MFs for the same flow, the other to collocate instances of the same MF for different flows. For each of the schemes, we discuss its potential and its limitations in terms of security, adaptability and potential for cost savings.
- We identify four patterns, based on our own experience, for placing virtual monitoring functions in the cloud. Those patterns are: *free*, *parser-collocation*, *job-collocation*, and *full-collocation*. These patterns

allow expressing different deployment solutions for different tenants and situations. In this paper, for the first time to the best of our knowledge, an integer linear programming (ILP) formulation is proposed to implement these patterns in the cloud.
- We propose three heuristics to make our placement algorithm scalable for a large network, 540k nodes and 11.5M edges, with about 40 requests (flows) at the same time. Those heuristics called, *chain partitioning*, *topology partitioning*, and *zoning*.

The remainder of this paper is structured as follows. Section 2 provides the background and literature review. Section 3 describes the problem that we intend to solve. Section 4 presents the proposed framework and the mathematical formulation of the proposed placement algorithm. Section 5 discusses the prototype, its integration into OpenStack, and the experimental results. Finally, Section 6 concludes the paper.

2. Related Work

Network monitoring using OpenFlow has been explored in recent years. Adrichem et al. [22] proposed an open-source software implementation to monitor per-flow metrics, especially throughput, delay, and packet loss, in OpenFlow networks. Chowdhury et al. [11] proposed a flexible and extendable monitoring framework for SDN, called PayLess. They focused on the trade-off between monitoring accuracy, timeliness and network overhead. The presented model supports different types of monitoring: performance, security, fault-tolerance, etc. For each selected monitoring type, a set of metrics is proposed. Performance metrics may include delay, latency, jitter, throughput, etc. As seen, the aforementioned work explored different virtual monitoring functions (delay, latency, jitter, throughput, packet loss) relying on OpenFlow or the other programmable routing platform which are needed in the cloud. Virtual monitoring functions are part of Network Function Visualization (NFV) and the optimal placement of VNF became a hot topic in cloud recently.

The optimal placement of VNFs is known to be an NP-hard problem [23], [24]. Particularly, optimizing both network and computing resources at the same time may make the problem difficult to solve in a reasonable period of time [25]. Thus, several research initiatives either propose to optimize these resources separately or to tackle the more general problem by proposing heuristics to address the pure optimization problem. Related works can be classified into four main categories: (i) in the first category, a fixed number of middleboxes are assumed to be already deployed in the network, and the optimal solution attempts to find for each traffic flow the optimal routes through them. (ii) in the second category, a set of pre-defined routes is assumed and the optimal solution is found for placing the VNFs within static routes. (iii) the third category tackles these two objectives separately. The placement objectives are prioritized and then run sequentially, and (iv) in the last category, both objectives are considered simultaneously to tackle the optimal placement.

In [26], an ILP formulation was proposed to optimally steer flows through static middleboxes in the network. Sekar et al. [27] proposed an optimization approach that assumes predefined paths between sources and destinations and then tried to find the optimal placement of the virtual appliances on these paths. Zhang et al. [28] and Addis et al. [29] proposed to prioritize the objective functions and then run them sequentially. The first step is to dynamically find the optimal computing node to initiate a service function, and then traffic is routed through the initiated service. In contrast, Addis et al. [29] minimized first the maximal link utilization; and then, after setting the routes, as a second objective, they minimized the network core utilization.

By considering both objectives dynamically, link and node core utilization, optimal placement of a chain becomes more challenging. That is, when the number of nodes in the network increases, the running time enhances exponentially. To tackle this issue, Mohammadkhan et al. [30] proposed to partition the problem into smaller pieces such that the final result remains close to the optimal solution. In contrast, Jarraya et al. [31] presented some heuristics to make the result scalable. For example, they restricted the resource availability in the network with respect to the type of VNF.

Compared to the existing works, we discovered different problem which how a virtual network function can be split into smaller pieces to decrease the total VNFs placement for a set of flows required to monitor a cloud-scale distributed application. Saving cost can be obtained by collocating the proper pieces between VNFs. To show the feasibility of our approach, the monitoring functions considered in this paper. We identified four patterns, based on our own experience, for placing monitoring functions in the cloud. Those patterns are: *free*, *parser-collocation*, *job-collocation*, and *full-collocation*. These patterns allow expressing different deployment solutions for different tenants and situations in the cloud. Moreover, we proposed three heuristics to make our placement algorithm scalable for a large network, 540k nodes and 1.5M links, with about 40 requests at the same time, each asking for placing three monitoring functions. Those heuristics called, chain partitioning, topology partitioning, and zoning. Those are our main contributions that have not been previously addressed in the aforementioned research works.

3. Motivation and Problem Statement

3.1. Monitoring Functions in a Cloud SDN

Figure 2 provides an illustration of the problem we want to tackle. Cloud providers usually use a *k-ary* fat-tree network architecture [32], [33], [34] (the figure depicts a 4-ary tree) with a layer of core switches, a layer of *pods* and a layer of hosts (server machines). There are k pods, each containing k switches distributed across two layers. The switches of the lower connect to $\frac{k}{2}$ of the $\frac{k^3}{4}$ hosts and to

$\frac{k}{2}$ switches in the upper layer which in turn are connected to $\frac{k}{2}$ of the core switches. With this architecture, there are many possible paths to connect any two hosts in the network. In this paper, we assume that all switches are SDN based. Moreover, it is assumed that the routing in fat-tree is not Equal Cost Multiple Path (ECMP) [35], since it is not based on weight function. We propose our cost function to model the cost of each link.

The figure further shows two three-tier applications A and B. Tier $A1$ of application A makes calls to tier $A2$ which in turns makes calls to $A3$. Thus, there are communication flows between $A1$ and $A2$ (one in each direction), and between $A2$ and $A3$. Similar holds for application B. The tiers of both applications are spread across the hosts of the cloud center [36]. Determining the paths is likely driven by attempting to choose the shortest paths for all flows without overloading the bandwidth capacity of any link. Choosing the shortest paths keeps message delay low and produces the overall smallest bandwidth consumption. The figure shows possible shortest paths between the tiers of application A in bold red lines and of application B in dotted blue lines. The cloud provider, however, might have to choose longer paths if the shortest path allocation exceeds the bandwidth capacity of any given link.

The application providers of applications A and B, might, at any time, want to start observing and monitoring the message flows. Simple monitoring functions could be to count the number of messages, the average message size or the overall bandwidth usage. But one could imagine more complex analysis, e.g., counting the number of messages that belong to a specific user session, the delay in which consecutive messages are sent, or even some analysis that is related to the message context. In some of the cases, switches only need to do some basic arithmetic operations at the time they receive a message and before they forward it. Such functionality is already provided in SDN implementations such as OpenFlow. But we envision that with further advances, switches can perform additional functionality. For instance, at the same time a switch forwards a message to the appropriate outbound link, it could also redirect the message to a locally installed MF that performs advanced analysis of the message. One can imagine that the cloud provider offers a suite of MFs that cover a wide range of functionality. However, it might even be possible that the applications themselves (or multi-tier platforms) develop their own MFs that can then be given to the cloud provider to be installed and executed on the switches as needed. We envision an implementation of this concept in a way that these MFs run within their own virtual machines on the switch. At the time a switch puts a message into the queue of the appropriate outbound link, it could also duplicate the message into specialized queues from where the MFs can pick it up and process it.

In our running example, we assume there are three different monitoring functions, *MF1-MF3*, and each of the four flows $A1 \rightarrow A2$, $A2 \rightarrow A3$, $B1 \rightarrow B2$, and $B2 \rightarrow B3$ would like to run all three of these monitoring functions[1]. In a larger setting, an application provider can indicate at any time for each of the flows the type of MFs it wants to execute. One can expect that most of the time no monitoring is needed, but when some performance problems are observed or during testing, monitoring is activated.

Thus, given a set of flows and the MFs each of these flows requests, the cloud provider now needs to decide on which switches these monitoring functions should run. Obviously, for a given flow the requested MF(s) must run on one of the switches on the path of the flow. Furthermore, the chosen switch must have the capacity to execute the MF.

There are several possible approaches to find proper allocations [37]. The most obvious one is to first determine the paths for all flows that minimizes overall bandwidth usage, and then deploy the requested MFs on one of the switches on the corresponding paths. However, MFs require processing and memory capacity. Thus, the assignment of MFs to switches should not exceed the processing capacity of individual switches. As such, when using this two-step approach, it might happen that the switches on the paths do not have sufficient processing capacity to execute all the requested MFs.

3.2. Allocation Challenge

Therefore, in our approach, we use a holistic allocation. Given a set of flows and a set of MFs for each of the flows, our solution will find a path for each flow and an allocation of MFs to switches such that each flow passes through switches that are able to host the requested MFs. At the same time, overall bandwidth usage is kept as small as possible.

In the next section, we present this allocation task as an optimization problem. We present it as 4-step approach that reflects different scenarios of how to model the problem in detail.

The first one considers the costs of each MF in isolation. Under all possible allocations, we aim to find one that (i) chooses paths so as to keep overall bandwidth usage as low as possible, (ii) prefers links with highest free bandwidth capacity, and (iii) prefers to put MFs on nodes with the highest free processing capacity.

The next three strategies look deeper into the allocation of MFs on switches. In particular, we believe that if several MFs run on the same switch the overall processing costs can be reduced compared to an approach where MFs run on different machines.

Per-flow collocation: In the first situation, we consider cost savings in the case where a flow requires more than one MF and the MFs are collocated on the same switch. The motivation behind this is that monitoring typically consists of two steps. In a first *pre-processing* step, a message needs to be parsed and decomposed and transformed before the second step is able to access and analyze the individual

1. for ease of illustration no monitoring functionalities are requested for the flows in opposite direction, i.e., $A2 \rightarrow A1$ etc.

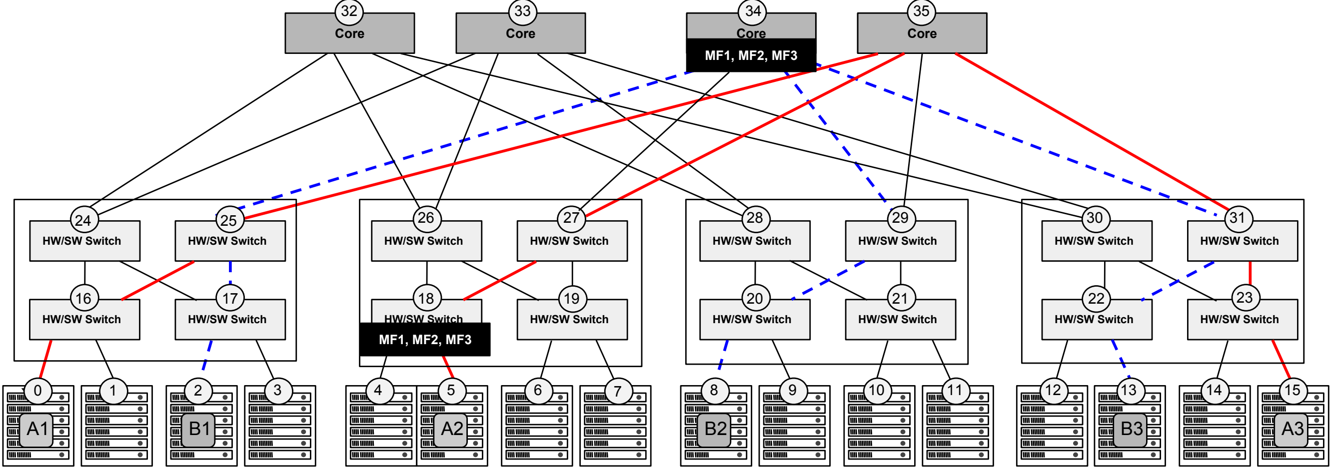


Figure 2: A fat-tree K=4 with 36 nodes; 4 core switches, 16 aggregation/edge switches, and 16 hosts. Two three-tier applications are placed; first one in hosts 0, 5, and 15, and the second one in hosts 2, 8, and 13. Examples of optimal paths for the flows of application A are shown in bold red lines and of application B in dotted blue lines. Moreover, the optimal nodes for placing three types of monitoring functions for these four flows ($A1 \rightarrow A2$, $A2 \rightarrow A3$, $B1 \rightarrow B2$, and $B2 \rightarrow B3$) are nodes 18 (edge switch) and 34 (core switch).

parts of a message. This has been discussed in detail in, e.g., [18]. If there are various monitoring tasks, the pre-processing step only needs to be executed once, leading to cost savings if several monitoring functions for the same flow are collocated compared to an approach where different nodes perform monitoring for the same flow.

Per-MF collocation: In similar spirit, we also consider that having a given MF installed on a particular node observing several flows will likely lead to cost savings compared to having independent installations of this same MF (possibly on different nodes), one for each flow. The reasons for such cost-savings are similar in spirit to any service-based architecture where the service execution within a single server instance handles the requests of many customers: use of less virtual machines, the use of multi-threading compared to multi-process solutions, more compact memory structures and resource sharing to maintain them, etc.

Finally, we look at a formalization that considers all of the above.

Note that in this paper, we assume that the locations of the individual components ($A1-A3$ and $B1-B3$ in our example), are fixed in advance and do not change. It might be interesting to also dynamically decide on the location of the components but this is so far out of the scope of our work as it would put yet another dimension to the problem.

As a final comment, we would like to note that the problem we address in this paper is far of being a static one. In particular, the set of flows that need to be observed is likely to change very often. For instance, applications typically want their flows observed during testing phases, at peak times, or when they observe any anomaly in the application's performance. However, when everything runs smoothly, they would likely want to switch off monitoring as it does involve overhead, as well as having monetary costs as we expect cloud providers to charge for this service. Thus, we have to find not only an effective but also an efficient solution to the problem, as one can expect that new assignments have to be determined on a periodic basis.

4. Proposed Framework

In this section, we present notations, variables, objective function and constraints that are used in the integer linear programming formulation of the optimal virtual monitoring function placement problem.

Given a set of applications, our framework has to determine the paths of each flow between application components and the location of any MF that is requested. Path and MF allocations need to be regenerated on a regular basis because not only the bandwidth and monitoring requirements of the applications under consideration might change but also the available bandwidth and execution capacity on the nodes on the cloud network might change due to other applications.

We express our problem as an optimization problem. We want to place flows on links of the network (determine the path a flow takes from start to end node) and place monitoring functions on nodes in the system such that both the overall bandwidth consumption of the flows as well as the overall processing costs for monitoring functions is low. The following section describes the input that our framework requires to be able to compute an appropriate placement, and how we express the solution as output of the optimization problem.

4.1. Notations and Configuration

4.1.1. Input parameters. Our framework needs as input information about the system configuration, information about

the costs of the MFs and information about the applications and their requirements in terms of bandwidth and MFs. Table 1 summarizes the basic input parameters that are needed. Table 2 represents the result of optimal placement, the best path and node(s) for each flow.

System configuration. $N = \{1, \cdots, n\}$ are the identifiers of all nodes in the system. This includes the leaf nodes of the network tree hosting the application components and the switches in the network hierarchy. L is the set of all tuples (i, j) such that there is a physical link between nodes i and j. That is, L describes the physical overlay of the network.

For each $(i, j) \in L$, $l_{i,j}$ indicates the current available capacity of bandwidth for this link. Note that this is capacity that is available for the flows under consideration for the current assignment execution. At any given time, this can be much less than the actual physical maximum capacity of the link, as other applications can use the link, too. While N and L are rather static (the physical nodes and network connections will rarely change), $l_{i,j}$ will change from one iteration to the next, and is likely derived from the current network utilization of the link. Our problem formalization will ensure that the sum of all flows assigned to a link does not exceed $l_{i,j}$. At the same time, our assignment will generally prefer to assign flows to links with higher $l_{i,j}$ (more available bandwidth) as it will allow for a more equal distribution and reduce the likelihood that links will not be able to provide peak needs.

In similar spirit, n_i is the current processing capacity of node i available for monitoring. It can be set to 0 if one does not want to put monitoring functionality on this node. However, we will later see another mechanism that will be more efficient. Our problem formalization will ensure that the sum of all MF processing costs assigned to node i will not exceed n_i. Again, our assignment will generally prefer to assign MFs to nodes with larger n_i as this will make the assignment more resistant to fluctuations.

Application. Each application is defined through the flows between its components and the location of these components as they determine the start and end points of the flows. Thus, we denote with $F = \{1, \cdots, h\}$ the identifiers of the flows under consideration and for each $f \in F$, the identifiers of start node $s_f \in N$ of flow f and destination node $d_f \in N$. Note that if messages between two components go in either direction then this creates two flows. Each flow $f \in F$ has a bandwidth requirement b_f. This bandwidth requirement can be given by the application (e.g., through a service level agreement), or can be automatically determined through some predication mechanism (e.g., estimated by the bandwidth requirement during the last interval of observation). For each of the flows $f \in F$ we have to determine the flow path consisting of links so that the first link starts at s_f and the last link ends at d_f.

Monitoring Functions. Finally, $M = \{1, \cdots, k\}$ represent the identifiers of all available MFs. In order to model cost-savings introduced by per-flow collocation, we represent the monitoring processing costs for MFs with two parameters. The first, c^p, represents the pre-processing costs. In our *per-flow collocation* model pre-processing can be done once per flow independently of the number of MFs for this flow. It presents the cost needed to parse and decompose incoming messages. Furthermore, each $m \in M$ has a *function-specific* processing cost c^s_m to perform the particular tasks implemented in m that are executed after the pre-processing step.

Both c^p and the c^ss are proportional to how heavy the flow is. Thus, c^p and the c^ss indicate the costs per bandwidth unit. For instance, if a node hosts MF m for flow f with bandwidth requirements b_f, then the m-specific processing costs will be $c^s_m \cdot b_f$.

We are aware that a bandwidth dependent cost is only an approximation. In fact, processing costs might rather depend on the number of messages than on the overall message size. But this would require yet another input parameter (number of messages per flow per time unit). Clearly, if it turns out that this would be a more appropriate parameter, then our approach can be easily adjusted to consider it.

In case of *per-MF collocation*, we assume that having a particular instance of MF m on a node i handling several flows will cost less than having an individual instance of m for each flow (possibly on different nodes). We express these reduce costs through the parameter P.

In order to link monitoring functions with applications, we further need to know for each flow what kind of monitoring it requires. Therefore, we define a binary variable $r_{m,f}$ for each MF $m \in M$ and flow $f \in F$ indicating whether flow f requires monitoring function m. One can also express this as a matrix R with rows being the monitoring functions and flows the columns and there is a 1 in $r_{m,f}$ if monitoring function m is needed by flow f. Finally, sa_f represents a sampling rate that can be provided by the application for each of its flows. If it is set to 1, then all messages are parsed and monitoring. At lower rates, the monitoring system will only take and analyze randomly selected messages from the message stream so that a rate sa_f of messages are monitored. This allows for a fine-tuning of the monitoring overhead.

4.1.2. Output Parameters. The output of our solution are two sets of binary variables depicted in Table 7. The first set determines the paths for all flows under consideration. The second set determines the deployments of monitoring functions and nodes and which flows they observe. That is, there is a binary variable $x_{i,j,f}$ for each physical link $(i, j) \in L$ and flow $f \in F$. The variable is set to 1 if f passes through the link. There is a binary variable $y_{i,m,f}$ for each node $i \in N$, MF $m \in M$ and flow $f \in F$. It is set to one if i executes m over flow f.

4.2. Objective Functions

As discussed in Section 3.2, we formalize our optimization in a 4-step approach. In the first step, S_1, we look at minimizing bandwidth cost while finding "good" links for the flow paths and "good" nodes for hosting monitoring functions. We will define further below what we consider "good" links and nodes. In this first step, no collocation

TABLE 1: Input parameters for the optimization problem.

System	$N = \{1, \cdots, n\}$	set of network nodes
	$L = \{(i,j)\}$	set of physical links (between nodes i and j)
	$l_{i,j}$	current maximum available capacity in units of bandwidth in link (i,j)
	n_i	current maximum processing capacity of node i available for monitoring
Application	$F = \{1, \cdots, h\}$	set of (unidirectional) flows between pairs of nodes
	s_f	node source of the flow f, $s_f \in N$
	d_f	node destination of the flow f, $d_f \in N$
	b_f	number of units of bandwidth needed by a flow f
Monitoring	$M = \{1, \cdots, k\}$	set of possible MFs
	c^p	cost for pre-processing messages (per bandwidth unit)
	c^s_m	function-specific cost for monitoring function $m \in M$ (per bandwidth unit)
	P	cost reduction factor in case of *per-MF* collocation
	$r_{m,f}$	binary variable set to 1 if MF $m \in M$ is required for $f \in F$
	sa_f	sampling rate for flow $f \in F$ as given by application

TABLE 2: Output.

$x_{i,j,f}$	routing decision variable for link $(i,j) \in L$ and flow f
$y_{i,m,f}$	monitor placement decision variable for MF m for node i of flow f

optimization is considered. In S_2 we consider additionally *per-flow* collocation. That is, whenever two MFs for the same flow are assigned to the same node, the pre-processing costs are only considered once while they have to considered twice if the two MFs are assigned to different nodes. Thus, S_2 favors collocation of MFs of the same flow whenever processing capacity allows. In S_3, we consider *per-MF* collocation. That is, whenever a MF m is installed on a node it can potentially handle all flows that go through this node. We assume that bundling the execution in one m installation reduces processing costs compared to a scenario where m is installed on several nodes, each time handling a different flow. Thus, S_3 favors collocation of the same MF for many flows on the same node. S_4 considers both collocations together. In cases $S_2 - S_4$, collocation might actually lead to flows with non-minimal bandwidth costs as they might be redirected through a different set of switches. However, our optimization functions ensure that the savings in processing costs outweigh the potentially additional bandwidth costs.

Note that in this section we do not consider the constraints that the allocation must fulfill. Examples of such constraints are that a link must have enough bandwidth capacity for all the flows that pass through the link and a node must have enough processing capacity to execute all the MFs that are running on it. These constraints will be discussed in Section 4.4.

Step one (S_1): Our first focus is to find short paths for each of the flows while finding "good" links and nodes to assign paths and MFs. In particular, our objective function prefers links with high available bandwidth over links with low available bandwidth, and nodes with high available processing capacity over nodes with lower available capacity. This reduces the potential for saturation.

Formally, this can be expressed as:

$$\begin{aligned} &MinimizeS_1 = \\ &\sum_{i\in N}\sum_{j\in N}\sum_{f\in F}\frac{1}{l_{i,j}}\cdot b_f \cdot x_{i,j,f} \quad + \\ &\sum_{i\in N}\sum_{f\in F}\sum_{m\in M}\frac{1}{n_i}\cdot(c^p + c^s_m)\cdot b_f \cdot sa_f \cdot y_{i,m,f} \end{aligned} \tag{1}$$

The first line of sums reflects the bandwidth costs. The actual bandwidth costs are $\sum_{i\in N}\sum_{j\in N}\sum_{f\in F} b_f \cdot x_{i,j,f}$. That is, for each link we have to sum up the bandwidth needs of each flow that goes through this link. The fewer links are used ($x_{i,j,f} = 1$) the better. Our objective function additionally weights these actual bandwidth needs with the factor $\frac{1}{l_{i,j}}$. With this, using links with a lot of available bandwidths (low $\frac{1}{l_{i,j}}$) are favored.

The second line of sums reflects the processing cost. Without considering any collocation, the actual processing costs are $\sum_{i\in N}\sum_{f\in F}\sum_{m\in M}(c^p+c^s_m)\cdot b_f \cdot sa_f \cdot y_{i,m,f}$. That is, for each node i and flow f that goes through this node, if there is a MF m for this flow running on the node ($y_{i,m,f} = 1$), then the node has to handle the parsing and the function-specific costs ($c^p + c^s_m$). This cost depends on the bandwidth needs and sampling rate of the flow ($sa_f \cdot b_f$). Our objective function additionally weights these actual processing costs with the factor $\frac{1}{n_i}$. With this, deploying MFs on nodes with a lot of processing capacity (low $\frac{1}{n_i}$) are favored.

Step two (S_2): The second step modifies S_1 by considering the scenario where collocation of MFs for the same flow reduces the pre-processing costs as it has to be done only once per node and flow. Formally, this enhancement leads to:

$$\begin{aligned} &MinimizeS_2 = \\ &\sum_{i\in N}\sum_{j\in N}\sum_{f\in F}\frac{1}{l_{i,j}}\cdot b_f \cdot x_{i,j,f} \quad + \\ &\sum_{i\in N}\sum_{f\in F}\sum_{m\in M}\frac{1}{n_i}\cdot c^s_m \cdot b_f \cdot sa_f \cdot y_{i,m,f} \quad + \\ &\sum_{i\in N}\sum_{f\in F}\frac{1}{n_i}\cdot c^p . b_f \cdot sa_f \cdot U_{if} \\ &U_{if} = \begin{cases} 1 & if \sum_{m\in M} y_{i,m,f} \geq 1 \\ 0 & otherwise \end{cases} \end{aligned} \tag{2}$$

The third line of Formula 2 considers that pre-processing messages of flow f has to take place only once for each node that has at least one MF for f while function-specific

processing has to take place for all MFs of this flow independent of their location as depicted in line 3.

Step three (S_3): The third step modifies again S_1 and considers processing cost reduction whenever a MF installed on a given node can handle more than one flow. Our modified objective is:

$$
\begin{aligned}
&S_3 = \\
&\sum_{i \in N} \sum_{j \in N} \sum_{f \in F} \frac{1}{l_{i,j}} \cdot b_f \cdot x_{i,j,f} \quad + \\
&\sum_{i \in N} \sum_{f \in F} \sum_{m \in M} \frac{1}{n_i} \cdot (c^p + c^s_m) \cdot b_f \cdot sa_f \cdot y_{i,m,f} - \\
&\sum_{i \in N} \sum_{m \in M} \sum_{f \in F} \frac{1}{n_i} \cdot c^s_m \cdot b_f \cdot sa_f \cdot P \cdot V_{imf} \\
&V_{imf} = \begin{cases} 1 & (if \sum_{f \in F} y_{i,m,f} > 1) \;\; \& \;\; (y_{i,m,f} = 1) \\ 0 & otherwise \end{cases}
\end{aligned}
\tag{3}
$$

That is, whenever there is more than one flow that use monitoring function m on node i, execution costs for each of these flows is reduced by P. Collocation means there are only one installation of m with common data structures and thread control mechanisms for all flows reduction overall processing costs. As the exact reduction depends on many factors we express it as a percentage value. The exact percentage value can be determined through micro-benchmarking. In our experiments, we used $P = 0.1$.

Step four (S_4): We finally combine per-flow and per-MF collocation. The resulting objective function to minimize is a direct combination of S_2 and S_3.

This strategy meets all objective functions. In this strategy, not only job collocation is considered among flows when they ask for same monitoring function but also parser collocation is applied per flow when that flow requires several monitoring functions. The main advantages of this strategy are: (1) only one VM per flow is created on a node regardless as soon as at least one MF is placed there. Therefore, only one VM creation cost is considered regardless of the number of monitoring functions related to that flow. Moreover, we will create one VM for all the same type of monitoring functions between different flows in order to job collocation, (2) we eliminate duplicated MFs in the network among flows. Hence, resource saving is significant, and (3) there is also a remarkable saving in the pre-processing monitoring cost per flow. For the disadvantages, this strategy suffers the same problems as the second and third strategies do.

Minimize S_4:

$$
\begin{aligned}
&S_4 = \\
&\sum_{i \in N} \sum_{j \in N} \sum_{f \in F} \frac{1}{l_{i,j}} \cdot b_f \cdot x_{i,j,f} \quad + \\
&\sum_{i \in N} \sum_{f \in F} \sum_{m \in M} \frac{1}{n_i} \cdot c^s_m \cdot b_f \cdot sa_f \cdot y_{i,m,f} \quad + \\
&\sum_{i \in N} \sum_{f \in F} \frac{1}{n_i} \cdot c^p . b_f \cdot sa_f \cdot U_{if} \quad - \\
&\sum_{i \in N} \sum_{m \in M} \sum_{f \in F} \frac{1}{n_i} \cdot c^s_m \cdot b_f \cdot sa_f \cdot P \cdot V_{imf} \\
&U_{if} = \begin{cases} 1 & if \sum_{m \in M} y_{i,m,f} \geq 1 \\ 0 & otherwise \end{cases} \\
&V_{imf} = \begin{cases} 1 & (if \sum_{f \in F} y_{i,m,f} > 1) \;\; \& \;\; (y_{i,m,f} = 1) \\ 0 & otherwise \end{cases}
\end{aligned}
\tag{4}
$$

The main advantages and disadvantages of these four strategies are summarized in Table 3.

4.3. Minimum Constraints for a correct solution

The following is the description of the general constraints that represent the base of our optimization model when both computing and networking resources are optimized simultaneously. These constraints are necessary to get only solutions that make sense.

4.3.1. Well-formed flow paths. The first set of constraints guarantees that we have well-formed paths for each flow.

First of all, we have to enforce that all $x_{i,j,f}$ are binary variables.

$$\forall f \in F, i, j \in N: \quad x_{i,j,f} \in \{0, 1\} \tag{5}$$

The next constraint avoids forming cycles in the solution path. Each node j can be at most once the end node of a link used by flow f. Each node i can be at most once the start node of a link used by flow f.

$$
\begin{aligned}
\forall j \in N, f \in F: \quad & \sum_{i \in N} x_{i,j,f} \leq 1 \\
\forall i \in N, f \in F: \quad & \sum_{j \in N} x_{i,j,f} \leq 1
\end{aligned}
\tag{6}
$$

The source node of a flow cannot be the end node of a link used for flow f, neither can the destination node of a flow be the start node of a link used by flow f.

$$
\begin{aligned}
\forall f \in F: \quad & \sum_{i \in N} x_{i,s_f,f} = 0 \\
\forall f \in F: \quad & \sum_{j \in N} x_{d_f,j,f} = 0
\end{aligned}
\tag{7}
$$

TABLE 3: Summary of advantages and disadvantages of different strategies of monitoring placement algorithm. Each strategy is looking for different objectives: O_1: maximal link utilization, O_2: maximal node processing utilization, O_3: parser collocation, O_4: job collocation.

Strategy	Covered objectives	Advantages	Disadvantages
S_1	$O_1 + O_2$	1. migration possibility 2. security, since the functions are deployed as separate VMs	1. existing duplicate MFs 2. VM creation cost 3. more monetary cost for cloud customer 4. more total cost compared to other strategies
S_2	$O_1 + O_2 + O_3$	1. saving VM creation cost per flow 2. saving the pre-processing monitoring cost per flow 3. security, since the functions are deployed as separate VMs 4. MF migration possibility through the nodes in the flow path	1. existing duplicate MFs 2. increasing the computing cost of nodes rapidly 3. difficult MF migration
S_3	$O_1 + O_2 + O_4$	1. eliminating duplicated MFs 2. MF migration possibility through the nodes in the common flows path	1. security problem, since different flows are steered through the same function [38] 2. high load on a node 3. MF migration is costly in terms of steering traffic
S_4	$O_1 + O_2 + O_3 + O_4$	1. saving VM creation cost per flow and per MF 2. eliminating duplicated MFs 3. saving the pre-processing monitoring cost per flow	1. security problem, since different flows are steered through the same function 2. difficult MF migration 3. high load on a node 4. MF migration is costly in terms of steering traffic 5. increasing the computing cost of nodes rapidly

The next constraint ensures that for any node i, if it is the end node of a link used by flow f then it also has to be start node of a link used by f unless it is the destination node.

$$\forall i \in N, \forall f \in F: \quad \sum_{j \in N} x_{j,i,f} + (1 \text{ if } i = s_f) = \sum_{r \in N} x_{i,r,f} + (1 \text{ if } i = d_f) \tag{8}$$

Next, we need a constraint to ensure that the allocated bandwidth does not exceed the current capacity of links, as follows:

$$\forall i, j \in N: \sum_{f \in F} b_f . x_{i,j,f} \leq l_{i,j} \tag{9}$$

4.3.2. Placing monitoring functions. The next set of constraints is in regard to the location of monitoring functions. The first constraint enforces that all $y_{i,l,f}$ are binary variables.

$$\forall f \in F, i \in N, m \in M: \quad y_{i,m,f} \in \{0,1\} \tag{10}$$

Next, we stipulate that the exact quantity of monitoring functions given in the demand list should be instantiated. That is, whenever a $r_{m,f} = 1$ in matrix R, there should be exactly one $y_{i,m,f}$ for one node i be set to one.

$$\forall m \in M, \forall f \in F: \quad \sum_{i \in N} y_{i,m,f} = r_{l,f} \tag{11}$$

The next constraint indicates that a monitoring function m for a flow f needs to be executed by a node i that is actually on the path of f. That is, if $y_{i,m,f}$ is set to one, there must be one $x_{i,j,f}$ set to one for any $j \in N$.

$$\forall i \in N \setminus \{s_f, d_f\}, m \in M, f \in F: \sum_{j \in N} x_{i,j,f} - y_{i,m,f} \geq 0 \tag{12}$$

The next constraint imposes the restriction that if a set of monitoring functions are put on a node i all costs associated with these functions do not exceed the current capacity of that node.

$$\begin{aligned} \forall i \in N: & \sum_{m \in M} \sum_{f \in F} (c^p + c^s_m) \cdot y_{i,m,f} - \sum_{f \in F} c^p \cdot U_{if} - \\ & \sum_{m \in M} c^s_m \cdot P \cdot V_{im} \quad \leq \quad n_i \\ & where \\ & U_{if} = \begin{cases} \sum_{m \in M} y_{i,m,f} - 1 & if \sum_{m \in M} y_{i,m,f} > 1 \\ 0 & otherwise \end{cases} \\ & V_{im} = \begin{cases} 1 & if \sum_{f \in F} y_{i,m,f} > 1 \\ 0 & otherwise \end{cases} \end{aligned} \tag{13}$$

4.4. Further restricting the search space

To find the best solution, solver needs to examine all solutions, all paths and nodes, using the usual methods like branch-and-cut [39]. Using proper constraints, we can help the solver to eliminate useless spaces that do not have an answer to the problem. In this Section, three constraints are proposed to restrict the search space.

The first one is related to finding all possible paths between the source and destination nodes. Clearly, if two components that have to be connected reside in two nodes that are very much apart from each other, it will be impossible to find a path that is only one or two hops long.

Furthermore, if two components are really close to each other, then a very long path is unlikely to be beneficial.

That is, we introduce for each flow f a H_f^{min} and H_f^{max} indicating the minimum and maximum link length.

$$\forall f \in F: \quad \sum_{i \in N} \sum_{j \in N} x_{i,j,f} \geq H_f^{min} \tag{14}$$

$$\forall f \in F: \quad \sum_{i \in N} \sum_{j \in N} x_{i,j,f} \leq H_f^{max} \tag{15}$$

Three famous datacenter architectures are fat-tree, BCube, and DCell. The fat-tree is the only architecture such that the shortest path length is equal to six, even the link failure is increased. For a high failure link ratio, DCell has an increase of up to 7 hops in average shortest path length, while in BCube, the maximum increase is 2 hops [40]. Our equations presented in Eq (14) and Eq (15) work for any datacenter. When the failure link ratio increases, H_f^{min} and H_f^{max} should be adopted.

The variables $l_{i,j}$ and n_i already provide some incentive to use or not use certain links for paths and nodes for monitoring functions. However, the search space is still large.

A first obvious optimization is that end nodes (compute nodes) can only be start or end nodes of flows but never intermediate nodes in a routing path. So, we define a region of nodes in the graph which should not be visited, called $\bar{R}$. It consists of all nodes in host level except the source and destination of all flows.

$$\forall i \in N, f \in F: \quad \sum_{j \in \bar{R}} x_{i,j,f} \leq 0 \tag{16}$$

In fact, in our implementation we simply set of network nodes N that we consider to the intermediate nodes in the fat tree plus the hosts (leaf nodes) in the tree that are source or destination node of at least one flow.

Furthermore, we allow the system to explicitly state the nodes on which monitoring functions can be set. We could make the probability low that a node is chosen to host monitoring functions by setting their n_i really high, but it is more efficient to explicitly exclude them. For instance, one can decide to never put a monitoring function on the leaf nodes but only put them on the intermediate network nodes. Furthermore, we can even choose only a subset of intermediate nodes to host monitoring functions. For instance, we might not consider it beneficial to have the switches in the highest level of the tree to host monitoring functions. Thus, we assume a set $MZ \in N$ is defined, and monitoring functions can only put on those nodes:

$$\forall l \in M, f \in F: \quad \sum_{i \in N \setminus MZ} y_{i,l,f} = 0 \tag{17}$$

5. Experimental Results

5.1. Implementation and OpenStack Integration

As a proof of concept, we implemented our algorithms and integrated them into OpenStack[2] and OpendDayLight (ODL)[3]. To realize the service chaining and traffic steering in the network [41], we use ODL. ODL is an SDN controller enabling rich and flexible networking functionalities. ODL has two roles in our framework: 1) obtaining the network and computing information, including the network topology and nodes and links loads and 2) updating the openflow tables when the result of optimal placement is found and new routes are identified for requested flows. The OpenStack Icehouse version[4] is used with a networking plug-in mechanism to connect to the ODL controller.

The tenant or cloud-provider interacts with our framework and specifies 4 items; monitoring needs, a set of flows and a set of vMFs, activated for each flow, along with monitoring policy (e.g., job collocation). Our framework applies chain partitioning heuristic to categorize the flows based on the same source and destination. Then, network and computing information, including the network topology and nodes and links loads are collected using interaction with OpenStack and ODL. Using this information, a graph annotated with costs and availabilities is built. Then, our framework applies topology partitioning and zoning process for each chain, separately. The whole partitioned-chains cannot be run in parallel, since they have the common path in fat-tree. The last heuristic, zoning, defines different zones in a partitioned-topology and restricts the search space to place monitoring functions. Finally, we find the best solution and ask OpenStack and ODL to deploy vMFs and steer traffic accordingly.

5.2. Simulation Setup

Table 4 shows how the variables are initialized for the monitoring optimization placement problem. The experiments run in fat-tree networks with $k = 4$ and $k = 8$, in which, there are 36 nodes with 48 links and 200 nodes with 386 links, respectively. The link capacity, CPU and memory sizes are generated randomly between 1 and 100.

In order to evaluate the presented monitoring placement algorithm for a set of distributed three-tier applications, we distinguish two possible cases:

- **Vertical distribution**: we consider 20 three-tier applications, placed in six racks. We partitioned them into two groups, A and B. Each group consists of ten three-tier applications placed in three racks.
- **Horizontal distribution**: we consider 20 three-tier applications, distributed in different racks, such that the racks have only one tier at most.

2. http://www.openstack.org
3. http://www.opendaylight.org/
4. http://www.openstack.org/software/icehouse/

TABLE 4: Monitoring optimization problem parameters initialization.

System	$N = \{1, \cdots, n\}$	fat-tree k={4,8,...,128}
	$l_{i,j}$	1Mbps...100Mbps
	n_i^{cpu}	1...100
	n_i^{mem}	1G...100G
	H_f^{min}	6
	H_f^{max}	6
Application	$F = \{1, \cdots, h\}$	2-40 flows
	b_f	1Mbps...5Mbps
Monitoring	$M = \{1, \cdots, m\}$	3 MFs
	$c_m^{s^{cpu}}$	1..3
	$c_m^{s^{mem}}$	1G..3G
	$c^{p^{cpu}}$	1
	P	10%,30%,50%
	$c^{p^{mem}}$	1G
	sa_f	1%...100%

In both cases, we run our algorithm for from one to 20 three-tier applications (2 to 40 flows, each three-tier application needs two flows). Moreover, we assume that the tiers of each application are not placed in the same rack. These 20 three-tier applications do not need the monitoring functions in the beginning. We assume that in each placement execution, some of them ask to be monitored. To initialize b_f, the number of units of bandwidth required by a flow f, random number is generated between $1Mpbs$ and $5Mpbs$. To have this flexibility that all flows can pass all links, b_f of the flows are less than the current available capacity in units of the link bandwidth in the network.

All applications need all three monitoring functions to measure delay, packet-loss, and jitter. Therefore, each application requires six MFs; three monitoring functions for each flow. c_m^j, the cost of monitoring function m in terms of CPU and memory, are randomly generated between 1 and 3 for CPU and $1G$ and $3G$ for memory. Different CPU and memory usage are considered for different MFs; delay, packet-loss, and jitter. Based on the objectives defined in Eq (3) and Eq (4), we consider a cost reduction when a job collocation happens in the network, which depends on parameter P. For example, when a job collocation happens, one three-tier application uses only three instead of six MFs in the best case. Thus, the cost reduction is considered based on a ratio of total monitoring process cost, parameter P.

We assume that the pre-processing of a flow requires one CPU ($c^{p^{cpu}}$) and $1G$ memory ($c^{p^{mem}}$). sa_f, which is the sampling rate, is generated randomly between 1 and 100 percent for each flow. $r_{m,f}$ for all flows and MFs is set to 1, since all flows ask for all monitoring functions. Since a path solution with less or greater than six hops is not realistic in fat-tree, they should be ignored by the framework. In this case, H_f^{min} and H_f^{max} are set to six.

When the MFs are placed on the optimized nodes, the optimized flows are steered through MFs from sources to destinations. Then, the resources of the related nodes and links are decreased. Finally, the network with updated available resources is fed to the inputs of the next placement execution. The GLPK (GNU Linear Programming Kit) is used on a machine with 6 cores Intel Westmere (XEON L5638) clocked at 2.30 GHz with 24 GB RAM to find the optimal solution and verify the performance of the proposed method.

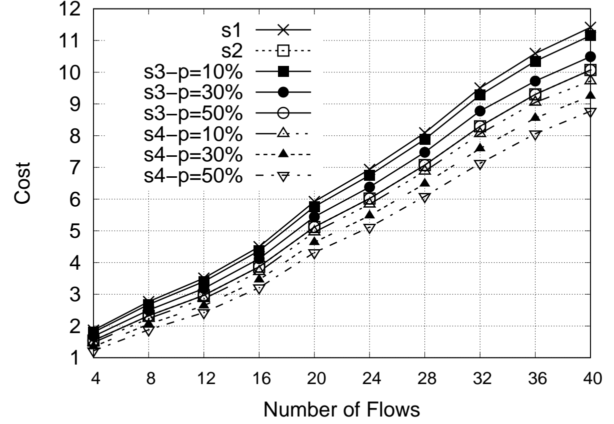


Figure 3: The cost of four different strategies to place three monitoring functions for two sets of three-tier applications (four flows) to 20 three-tier applications (40 flows), in fat-tree K=4. Note that, the first, second, and third tier of first set of applications are placed in rack 0, rack 5, and rack 15 and the first, second, and third tier of second set of applications are placed in rack 2, rack 8, and rack 13, respectively.

5.3. Result

5.3.1. Monitoring Cost Evaluation. Figure 3 compares our three proposed methods to the latest related works in terms of the monitoring placement cost of three-tier applications.

The scenario is monitoring two different sets of three-tier applications such that the three tiers of first set are placed in rack 0, rack 5 and rack 15, and the three tiers of second set are placed in rack 2, rack 8 and 13, respectively as seen in Figure 2. The experiments run in fat-tree K=4 with 36 nodes and 48 links. First, we evaluate two three-tier applications and four flows (first two flows between first and second tiers and second flows between second and third tiers of two applications) from two sets. Then, for the defined scenario, we incrementally increase the demand for monitoring. For each flow three monitoring functions are requested. The number of CPU and Memory size for these three monitoring functions are $(2, 2G)$, $(3, 2G)$, and $(1, 2G)$, respectively. The resources of the nodes (CPU and Memory size) vary between 12 and 96. For the links, the bandwidth capacity varies between 40Mbps and 95Mbps. The demanded bandwidth (b_f) by the flows varies between 1Mpbs and 5Mpbs. Moreover, the sampling rate of the flows (s_a) varies between 0.1 and 0.6. The simulation for strategies S_3 and S_4, were repeated three times for three different P, which indicate 10%, 30%, and 50% saved

monitoring processing cost while parser collocation, as seen in Figure 3.

In this Section, we compare strategies S_2, S_3, and S_4 to S_1 and strategy S_4 to S_2 and S_3. Strategy S_2 compared to S_1 is saving pre-processing cost. Strategy S_3 is reducing the monitoring functionally cost in comparison to S_1. Strategies S_2 and S_3 are not comparable, since they are reducing two different cost types. Strategy S_4 is reducing pre-processing and monitoring functionality costs simultaneously, therefore, S_4 is compared to S_2 and S_3.

Figure 4a shows the saving cost percentage of strategy S_4, which applies parser and job collocation simultaneously, compared to the only job collocation solution, S_3. The number of three-tier applications is experimented from two (four flows) to 20 (40 flows) applications. As shown, when there are only four flows, strategy S_4 saves around 20% of monitoring cost compared to S_3 for the medium P. As the number of flows increases, the number of requests for monitoring enhances. Therefore, available resources required by monitoring functions are decreased. Thus, the saving percentage decreases over time. However, the number of monitoring requests for the flows at a moment is limited. Moreover, the allocated resources are released while the monitoring time is over. Hence, we can claim that we saved 15% of the monitoring cost in average compared to when job collocation is applied.

Selection of strategies should be based on customer requirements and network load conditions. The strategies S_3 and S_4 apply heavy load on nodes, because they seek to put the MFs on a node to satisfy their own objectives. But strategy S_3 is better, in terms of the distribution of the MFs, than strategy S_4. Someone may neglect the 15% improvement of S_4 compared to S_3, when the load of many nodes in the datacenter is high.

Figure 4b illustrates the saving cost percentage of strategy S_4 compared to only the parser collocation strategy. As shown, once we consider higher contribution for job collocation, higher P, the difference of saved cost is higher, about 15% in average. However, when the job collocation share is limited, $P = 10\%$, the saving percentage is significantly close.

Strategy S_2 has several benefits. For example, the MF migration is possible through the nodes in the path of each flow. Security is another important issue. In Strategy S_2, since the monitoring functions are deployed as separate VMs, there is not security concern between different applications. Someone may choose strategy S_2 in spite of S_3 considerable improvement.

Figure 4c shows the percentage of the saved cost of strategies S_2, S_3, and S_4 compared to the latest works, strategy S_1. Latest works only optimize node and link costs to place monitoring functions without considering any pattern. As shown, strategy S_4 saves 30 and 20 percent of total cost in average where P equals to 50% and 10% respectively. Moreover, as seen the lowest cost improvement belongs to the strategy S_3 with $P = 10\%$ which has less than 5% improvement. However, strategy S_4 always stays at the top of saving cost.

Note that although strategy S_1 has the worst cost result compared to other strategies, it has its own two main benefits: 1) monitoring function migration is possible, since there is not correlation between monitoring functions per flow and among flows and 2) security concern is resolved between different applications, since each function works per flow.

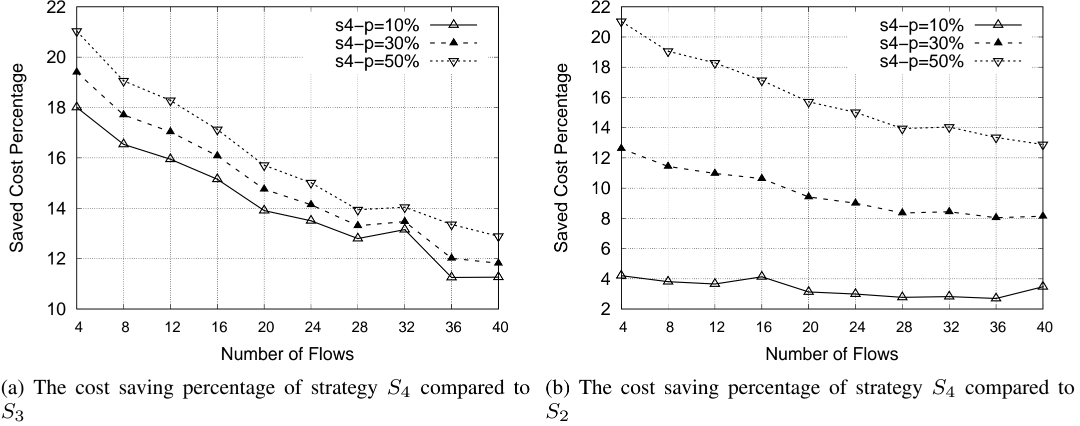
As a conclusion in this Section, our proposed placing method that considers parser and job collocation at the same time, strategy S_4, improves 20-30 percent of total monitoring costs in the network, which is a significant improvement.

Algorithm 1 Scalable Heuristic for Monitoring Functions Placement.

```
Require: G: Topology graph
Require: G.Status: cloud infra. links and nodes current available
    resources
Require: Chains[]: each chain has MFs with their types and
    needed resources
 1: C[][] = ChainPartitioning(Chains[])                ▷ /* Step1 */
 2: i = 1
 3: for all partitions in C[][] do
 4:     RG′ = ExtractGraph(G,C[i][].src,C[i][].dest) ▷ /* Step2 */
 5:     RG′′ = ApplyZoning(RG′)                        ▷ /* Step3 */
 6:     T = TopologyPartitioning(RG′′)                 ▷ /* Step4 */
 7:     Sol[i] = FindBestSolution(T,C[i][])            ▷ /* Step5 */
 8:     G.Status′ = UpdateStatus(G.Status,Sol[i])      ▷ /* Step6 */
 9:     i = i + 1
10: end for
11: i = 1
12: TotalCost = 0
13: TotalTime = 0
14: for all partitions in C[][] do
15:     TotalCost= TotalCost + Sol[i].cost
16:     TotalTime= TotalTime + Sol[i].time
17:     i = i + 1
18: end for
```

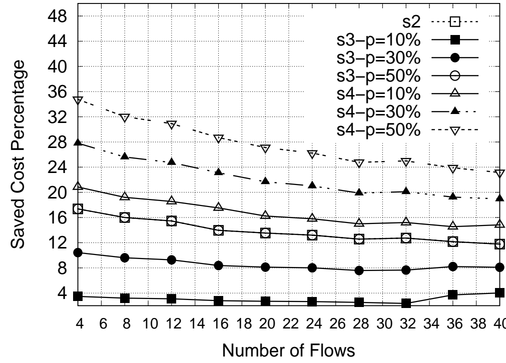
5.3.2. Scalability. The majority of existing placement approaches suffers the scalability problem [26], [27], [29], [30]. When the number of nodes increases the execution time enhances, exponentially [42], [43]. Table 5 shows the placement execution time of four different strategies, discussed in this paper, for from 14 three-tier applications (28 flows) to 20 (40 flows), in a fat-tree K=4, which consists of 36 nodes and 48 links. It is observed that the strategy S_1 is scalable when the number of flows is increased. Strategy S_4 which is the best among these four strategies, with respect to the cost function (see Section 4.2), is not scalable even in a small network. As seen, the strategy S_4 execution time, for $P = 10\%$, takes 1221.8s (about 20 minutes) for 40 flows, bolded in Table 5. In a real cloud datacenter, which consists of huge number of nodes and links with fair number of applications in real time, the placement algorithm should be scalable. Therefore, we propose a solution to tackle this scalability problem.

We propose three heuristics to combat the scalability issue: (1) Chain partitioning, (2) Zoning, and (3) Topology



(a) The cost saving percentage of strategy S_4 compared to S_3

(b) The cost saving percentage of strategy S_4 compared to S_2



(c) The cost saving percentage of strategy S_4 compared to S_1

Figure 4: The cost saving percentage of strategy S_4 compared to other strategies, S_1, S_2, and S_3. S_3 contains only job collocation, S_2 contains only parser collocation, and S_1 does not contain any job/parser collocation.

TABLE 5: The monitoring function placement execution time of different strategies (S_1 to S_4) for from 14 three-tier applications (28 flows) to 20 three-tier applications (40 flows), each flow requests three monitoring functions. This result is for fat-tree K=4, which consists of 36 nodes and 48 links.

Flows	S_1	S_2	S_3			S_4		
			$P = 10\%$	$P = 30\%$	$P = 50\%$	$P = 10\%$	$P = 30\%$	$P = 50\%$
28	2.1s	1.8s	614.9s	309.8s	8s	82.6s	53.5s	41.8s
32	2.4s	3.1s	$\geq$ 2h	1340.3s	57.1s	108.5s	54s	43.6s
36	2.5s	4.2s	$\geq$ 2h	$\geq$ 2h	539s	213.6s	97.9s	65s
40	3.7s	118.9s	$\geq$ 2h	$\geq$ 2h	581.1s	**1221.8s**	98.6s	81.1s

partitioning. The first heuristic is about partitioning the whole chains into separate parts (Algorithm 1, line 1). Each part has the same source and destination. For example, assume that the three tiers of n applications are placed in rack x, rack y, and rack z, respectively. So, the first partitioned-chain includes the $\frac{n}{2}$ flows go from rack x to rack y and the second partitioned-chain includes the $\frac{n}{2}$ flows go from rack y to rack z. Therefore, two different solvers are run simultaneously in order to consider the monitoring requests. The main weakness of this heuristic is that there is no job collocation between two partitioned-chains. For example, imagine a simple scenario that there is one three-tier application, two flows, and each flow needs only packet-loss monitoring function. Two different solvers do not see each other to collocate two MFs, even if they put them in the same node (in the common path). The main point is that, we still take the advantages of the job collocation between $\frac{n}{2}$ flows and parser collocation per flow. In step two, for each partitioned-chain which consists of the requests with the same source and destination, we extract the related sub-graph from fat-tree.

In the second heuristic, step 3, the sub-graph is divided into three zones. In 3-level k-ary fat-tree, there are $(\frac{k}{2})^2$ paths between two hosts on different pods. The first zone

consists of one edge and $\frac{k}{2}$ aggregation switches. The second zone covers $\frac{k}{2}$ core switches and the last zone consists of $\frac{k}{2}$ aggregation switches and one edge switch. Therefore, each monitoring function should be placed in only one zone. The main weakness of the zoning heuristic is to eliminate the parser collocation, since the MFs belong to each flow are placed on three different nodes.

In the third heuristic, step 4, we apply the topology partitioning for each partitioned-chain. We observed that there are several independent paths in each sub-graph. Thus, there is no need to run the whole part of sub-graph. We run our optimization algorithm in all partitioned-topology in parallel to determine the optimal placement of the MFs and the routes among them. Finally, the best solution is selected from one of the partitioned-topology, who minimizes our objective function optimally. Note that the third heuristic does not have any impact on the result and the optimal value stays the same with or without topology partitioning.

Note that the whole partitioned-chains cannot be run in parallel. The reason is that the partitioned-chains have the common path in fat-tree. If we run them in parallel, the available link and node resources in the common paths are considered independent which is wrong. So, we need to run them sequentially, and after each run we should update the available resources for the next run.

Now, we present the asymptotic complexity of our proposed scalable heuristic, presented in Algorithm 1. The complexity of step 1 is $\mathcal{O}(|R|\cdot|SD|)$, where $|R|$ is the number of requests (flows) and $|SD|$ is the number of different sources and destinations among the requests. For example, if all requests have the same source and destination, $|SD| = 1$. Step 2 is related to extracting sub-graph from fat-tree based on each partitioned-chain source and destination. There is no need to extract the sub-graph in real time. The structure of all sub-graphs is similar. The difference between them is only the link and node costs. Therefore, the complexity of step 2 is $\mathcal{O}(1)$. In step 3, the zoning is applied on the sub-graph, which is static. The first zone consists of one edge and $\frac{k}{2}$ aggregation switches. The second zone covers $\frac{k}{2}$ core switches and the last zone consists of $\frac{k}{2}$ aggregation switches and one edge switch. Therefore, the complexity of step 3 is $\mathcal{O}(1)$. An inherent characteristic of tree-based topology structure is the organization of the connectivity using sets of disjoint paths in the same PoD or in different PoDs, between any two hosts. This partitioning is done in advance. Therefore, the complexity of step 4 is $\mathcal{O}(1)$, too. For step 5, there are $|T|$ independent partitions and our placement algorithm runs in all partitions in parallel and the best solution is selected from the one who minimizes our objective function better. Therefore, the complexity of step 5 is $\mathcal{O}(|T|)$. In the last step, we need to update the network status, routing and node cost, for next partitioned-chain. In the worst case, six links and five nodes should be updated. Therefore, the complexity of step 6 is $\mathcal{O}(1)$. The total complexity of our proposed heuristic is $\mathcal{O}(|R| \cdot |SD| + |CP| \cdot |T|)$, where $|R|$, $|SD|$, $|CP|$, and $|T|$ are number of requests, number of different sources and destinations among the requests, number of partitioned-chains and number of independent partitioned-topologies in each sub-graph, respectively.

TABLE 6: Chain partitioning, topology partitioning, and zoning heuristics settle scalability issue when our proposed approach (S_4 with P = 0.1) is employed in the fat-trees with different Ks.

K	Node	Link	Time(s)	
			Topology Partitioning (TP)	TP+Zoning
4	36	48	$0.1s \times 4 = 0.4s$	$0s \times 4 = 0s$
8	200	384	$0.2s \times 4 = 0.8s$	$0.06s \times 4 = 0.24s$
16	1,296	3,072	$0.6s \times 4 = 2.4s$	$0.2s \times 4 = 0.8s$
24	4,056	10,368	$0.9s \times 4 = 3.6s$	$0.3s \times 4 = 1.2s$
32	9,248	24,576	$1.6s \times 4 = 6.4s$	$0.5s \times 4 = 2s$
48	30,000	82,944	$2.5s \times 4 = 10s$	$0.8s \times 4 = 3.2s$
64	69,696	196,608	$3.6s \times 4 = 14.4s$	$1.4s \times 4 = 5.6s$
128	540,800	1,572,864	$15.5s \times 4 = 62s$	$8.5s \times 4 = 34s$

Table 6 shows how our three heuristics improve the execution time significantly. Second and third columns illustrate the number of nodes and links in the fat-trees with $K = 4$ to 128, respectively. As shown in Table 5, our proposed model execution time for the placement of 20 three-tier applications (40 flows) takes more than 20 minutes in a small fat-tree with $K = 4$. The execution time in a real cloud with a huge network must be reasonable and the MFs must be placed in real time. As shown, in Table 6, by applying the chain and topology partitioning, we could decrease the execution time from 1221.8s to 0.4s for fat-tree $K = 4$. Note that in our scenario, there are 20 three-tier applications in the fat-tree, 10 in position A and 10 in position B. So, when the chain partitioning is applied, the whole chains are divided into four partitions: A1-A2, A2-A3, B1-B2, and B2-B3 (see Figure 2). Hence, four sub-graphs are considered in fat-tree. Then, the second and third heuristics are applied for each sub-graphs. As seen in Table 6, the execution time for each partitioned-chain takes 0.1s for fat-tree $K = 4$. Since we need to find the optimal placement for four partitioned-chains, it is multiplied into four ($0.1s \times 4 = 0.4s$). The last column shows the execution time by applying the whole three heuristics. As seen, the zoning heuristic speeds up about two to three times better the execution time.

It is worth to investigate the penalty cost percentage of non-scalable strategy S_4 and scalable one for $P = 10\%$. As mentioned, there are three heuristics in our scalable algorithm. Table 7 compares only the chain and topology partitioning heuristics without zoning. The scenario is the same as explained for Figure 3. For two three-tier applications (4 flows), since there is not any same type of MFs in each partitioned-chain (there are four partitions and each one consists of only one flow), there is no chance for job collocation. Thus, the result is same as strategy S_2. Therefore, we should pay the cost of job collocation which is about 4.4% in this case.

In the execution of optimal placement algorithm by considering the whole graph, if optimal nodes for placing the MFs are selected in non-common path between two chains of a three-tier application, for sure the proposed heuristics obtains the same result by running the partitioned-chains, sequentially. Thus, there is no penalty cost in this situation. Moreover, when the optimum node is in the common path

TABLE 7: Penalty cost percentage of non-scalable and scalable solutions for strategy S_4 with $P = 10\%$, for K=4. Two heuristics are considered for the scalable solution, chain and topology partitioning.

Flows No.	$S_4(Non-Scalable)$	$S_4(Scalable)$	Penalty
4	1.47841	1.543333	4.4%
8	2.23623	2.236233	0%
12	2.85811	2.85811	0%
16	3.70936	3.70936	0%
20	4.96065	4.96065	0%
24	5.83917	5.83917	0%
28	6.87618	6.87618	0%
32	8.05934	8.05934	0%
36	9.05015	9.05136	0.01%
40	9.72013	9.80622	0.88%

of two chains and it has available enough resources for both chains, we do not pay any penalty, too. However, when the node in the common path does not have available enough resources for both chains and another node in the common path is used for the placement, we pay some penalty. Since heuristic algorithm runs sequentially for the first and then the second chains, the resource management is not optimally controlled. First, the heuristic algorithm allocates the whole required resources for the first chain. Then, it allocates the rest of resources for the second chain. As all resources of the first node is allocated, another node is chosen. However, it may be optimal if the resource allocation is done reverse.

As illustrated in Table 7, for the cases of 8 to 32 flows (2 to 8 three-tier applications), there is no penalty. The reason is that the proposed heuristic solution has the same picture as non-scalable. In both cases, all the MFs are placed on nodes 18, 25, and 24. Node 25 is not in the common path from rack 0 to rack 5 and rack 5 to rack 15 while the node 18 is in the common path. For the second group of applications placed in rack 2, rack 6, and rack 13, we have the same situation. Node 25 is not in the common path between rack 2 and rack 6 and rack 8 and rack 13 while the node 34 is in the common path. As seen, we do not have two common nodes in the same paths. In the last two cases, for the flows go from rack 0 to rack 5 and rack 5 to rack 15, another node is selected, node 27, since node 18 does not have enough resources to place all MFs. These two nodes, 18 and 27, exist in the common paths between rack 0 and rack 5 and rack 5 and rack 15. Thus, the execution of the placement algorithm in the whole graphs has better picture to minimize the cost between nodes 18 and 27. While, in our heuristic for the execution related to the second partitioned-chain, the node 18 is selected first to place all MFs related to the 10 flows of the first partitioned-chain and there is no chain to reorganize again the whole cost when the second partitioned-chain is selected.

6. Conclusion

An optimum and high scalable monitoring function placement method has been modelled for cloud computing infrastructures in current paper. The general idea is to reduce the requested monitoring cost by placing monitoring functions in the optimum nodes besides steering the flows through the most advantageous links. A noteworthy step has provided in this paper considering monitoring functions collocations therewith nodes and links available resources in order to minimize the total monitoring cost. Two types of collocations are contemplated; first, collocating two similar monitoring functions for two different flows (job collocation) and second, collocating two distinct monitoring functions for the same flow (parser collocation). We formulated the cost reduction when only one instead of at least two similar monitoring functions are set on a node to perform that type of monitoring function. Furthermore, parsing and serialization of the received packets to extract the monitored flow on a node is costly. Thus, we considered that when at least two monitoring functions for a specific monitored flow are placed on one node the parsing is done once. Therefore, the parsing cost diminished to less than half.

Moreover, we proposed heuristics to improve the scalability of the placement algorithm. These heuristics are (1) Chain partitioning, (2) Topology partitioning, and (3) Zoning. We represented detailed comparative study of the proposed model and heuristics on fat-tree topology. The proposed model saves at least 20 percent of total monitoring cost in average compared to the latest related works. Further, our presented heuristics reduce placement algorithm execution time from couple of hours to seconds scale. This scalable solution is applicable in a large dataset consists of 520,800 nodes and 1,572,864 edges (Fat-tree K = 128).

7. Acknowledgement

We are grateful to FRQNT of Canada, Fonds de recherche du Quebec - Nature et technologies, for partially financial support.

References

[1] T. Choi, S. Kang, S. Yoon, S. Yang, S. Song, and H. Park, "Suvmf: software-defined unified virtual monitoring function for sdn-based large-scale networks," in *Proceedings of The Ninth International Conference on Future Internet Technologies*. ACM, 2014, p. 4.

[2] C. Yu, C. Lumezanu, A. Sharma, Q. Xu, G. Jiang, and H. V. Madhyastha, "Software-defined latency monitoring in data center networks," in *Passive and Active Measurement*. Springer, 2015, pp. 360–372.

[3] Y. Zhang, "An adaptive flow counting method for anomaly detection in sdn," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 25–30.

[4] G. Huang, C.-W. Chang, C.-N. Chuah, and B. Lin, "Measurement-aware monitor placement and routing: a joint optimization approach for network-wide measurements," *IEEE Transactions on Network and Service Management*, vol. 9, no. 1, pp. 48–59, 2012.

[5] G. Liu and T. Wood, "Cloud-scale application performance monitoring with sdn and nfv," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, March 2015, pp. 440–445.

[6] B. Adler, "Architecting scalable applications in the cloud: The caching tier," Tech. Rep., 2012, http://www.rightscale.com/blog/enterprise-cloud-strategies/architecting-scalable-applications-cloud-caching-tier.

[7] H. J. Syed, A. Gani, R. W. Ahmad, M. K. Khan, and A. I. A. Ahmed, "Cloud monitoring: A review, taxonomy, and open research issues," *Journal of Network and Computer Applications*, vol. 98, pp. 11–26, 2017.

[8] R. Yu, G. Xue, V. T. Kilari, and X. Zhang, "Network function virtualization in the multi-tenant cloud," *IEEE Network*, vol. 29, no. 3, pp. 42–47, 2015.

[9] L. Askari, A. Hmaity, F. Musumeci, and M. Tornatore, "Virtual-network-function placement for dynamic service chaining in metro-area networks," in *2018 International Conference on Optical Network Design and Modeling (ONDM)*. IEEE, 2018, pp. 136–141.

[10] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, "A survey on service function chaining," *Journal of Network and Computer Applications*, vol. 75, pp. 138–155, 2016.

[11] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "Payless: A low cost network monitoring framework for software defined networks," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–9.

[12] B. Johnson, "Opnfv sfc," Tech. Rep., https://docs.google.com/presentation/d/1gbhAnrTYbLCrNMhMXin0lxjyg-7IHNPjrlBTIjwAzys/edit?pli=1#slide=id.p.

[13] R. Penno and P. Quinn, "Odl: Service function chaining," Tech. Rep., https://docs.google.com/presentation/d/1JOWN-Xf0_GrV5v_n5q9_j9nUQcM8MjlNl3Sw1YN1bxU/edit?pli=1#slide=id.p4.

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[15] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[16] Y. Yu, C. Qian, and X. Li, "Distributed and collaborative traffic monitoring in software defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 85–90.

[17] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[18] G. Liu, M. Trotter, Y. Ren, and T. Wood, "Netalytics: Cloud-scale application performance monitoring with sdn and nfv," in *ACM/IFIP/USENIX Int. Middleware Conference*, 2016.

[19] L. Jose, M. Yu, and J. Rexford, "Online measurement of large traffic aggregates on commodity switches." in *Hot-ICE*, 2011.

[20] Y. Jarraya, T. Madi, and M. Debbabi, "A survey and a layered taxonomy of software-defined networking," *IEEE communications surveys & tutorials*, vol. 16, no. 4, pp. 1955–1980, 2014.

[21] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[22] N. L. Van Adrichem, C. Doerr, F. Kuipers *et al.*, "Opennetmon: Network monitoring in openflow software-defined networks," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–8.

[23] C. Pham, N. H. Tran, S. Ren, W. Saad, and C. S. Hong, "Traffic-aware and energy-efficient vnf placement for service chaining: Joint sampling and matching approach," *IEEE Transactions on Services Computing*, 2017.

[24] Q. Zhang, Y. Xiao, F. Liu, J. C. Lui, J. Guo, and T. Wang, "Joint optimization of chain placement and request scheduling for network function virtualization," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 731–741.

[25] P. Cappanera, F. Paganelli, and F. Paradiso, "Vnf placement for service chaining in a distributed cloud environment with multiple stakeholders," *Computer Communications*, vol. 133, pp. 24–40, 2019.

[26] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella, "Stratos: A network-aware orchestration layer for virtual middleboxes in clouds," *arXiv preprint arXiv:1305.0209*, 2013.

[27] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 24–24.

[28] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan *et al.*, "Steering: A software-defined networking for inline service chaining," in *Network Protocols (ICNP), 2013 21st IEEE International Conference on*. IEEE, 2013, pp. 1–10.

[29] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," 2015.

[30] A. Mohammadkhan, S. Ghapani, G. Liu, W. Zhang, K. Ramakrishnan, and T. Wood, "Virtual function placement and traffic steering in flexible and dynamic software defined networks," in *Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on*. IEEE, 2015, pp. 1–6.

[31] Y. Jarraya, A. Shameli-Sendi, M. Pourzandi, and M. Cheriet, "Multi-stage ocdo: Scalable security provisioning optimization in sdn-based cloud," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 572–579.

[32] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xomb: Extensible open middleboxes with commodity servers," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM, 2012, pp. 49–60.

[33] R. Alshahrani and H. Peyravi, "Modeling and simulation of data center networks," in *Proceedings of the 2Nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '14. New York, NY, USA: ACM, 2014, pp. 75–82. [Online]. Available: http://doi.acm.org/10.1145/2601381.2601389

[34] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1402946.1402967

[35] D. Thaler and C. Hopps, "Multipath issues in unicast and multicast next-hop selection," Tech. Rep., 2000.

[36] M. H. Ferdaus, M. Murshed, R. N. Calheiros, and R. Buyya, "An algorithm for network and data-aware placement of multi-tier applications in cloud data centers," *Journal of Network and Computer Applications*, vol. 98, pp. 65–83, 2017.

[37] H. Moens and F. De Turck, "Vnf-p: A model for efficient placement of virtualized network functions," in *Network and Service Management (CNSM), 2014 10th International Conference on*. IEEE, 2014, pp. 418–423.

[38] S. Lal, T. Taleb, and A. Dutta, "Nfv: Security threats and best practices," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 211–217, 2017.

[39] R. A. Stubbs and S. Mehrotra, "A branch-and-cut method for 0-1 mixed convex programming," *Mathematical programming*, vol. 86, no. 3, pp. 515–532, 1999.

[40] R. de Souza Couto, S. Secci, M. E. M. Campista, and L. H. M. K. Costa, "Reliability and survivability analysis of data center network topologies," *Journal of Network and Systems Management*, vol. 24, no. 2, pp. 346–392, 2016.

[41] S. Gu, Z. Li, C. Wu, and C. Huang, "An efficient auction mechanism for service chains in the nfv market," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 2016, pp. 1–9.

[42] W. Ma, J. Beltran, Z. Pan, D. Pan, and N. Pissinou, "Sdn-based traffic aware placement of nfv middleboxes," *IEEE Transactions on Network and Service Management*, vol. 14, no. 3, pp. 528–542, 2017.

[43] A. Gupta, B. Jaumard, M. Tornatore, and B. Mukherjee, "A scalable approach for service chain mapping with multiple sc instances in a wide-area network," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 529–541, 2018.