SDN-based Virtual Machine Management for Cloud Data Centers

Richard Cziva, *Student Member, IEEE*, Simon Jouët, *Student Member, IEEE*, David Stapleton, Fung Po Tso, *Member, IEEE*, and Dimitrios P. Pezaros, *Senior Member, IEEE*

***Abstract*—Software-Defined Networking (SDN) is an emerging paradigm to logically centralize the network control plane and automate the configuration of individual network elements. At the same time, in Cloud Data Centers (DCs), although network and server resources are collocated and managed by a single administrative entity, disjoint control mechanisms are used for their respective management. In this article, we propose a unified server-network resource management for such converged Information and Communication Technology (ICT) environments. We present a SDN-based orchestration framework for live Virtual Machine (VM) management that exploits temporal network information to migrate VMs and minimize the network-wide communication cost of the resulting traffic dynamics. A prototype implementation is presented, and a Cloud DC testbed is used to evaluate the impact of diverse orchestration algorithms. Our live VM management has been shown to reduce the network-wide communication cost, especially for the high-cost and congestion-prone core and aggregation layers of the DC. Our results show an increase in network-wide throughput by over 6 times, as well as over 70% communication cost reduction by migrating less than 50% of the VMs.**

***Keywords*—Software Defined Networking, Virtual Machine Management, Cloud Data Centers**

I. Introduction

The advent of Cloud Computing has given rise to new and exciting prospects for individuals, small and medium-sized enterprises and large organizations who can flexibly lease processing, storage, and network resources on-demand, according to their temporal needs. Underpinning Cloud Computing are Data Center (DC) infrastructures, maintained and managed at scale by local as well as global operators such as Amazon, Rackspace, Microsoft, and Google, typically offered as-a-service to corporate and individual customers over the Internet. Each of these Cloud DCs typically house tens of thousands of servers [1].

In order to be sustainable, the significant capital outlay required for building a DC makes maximization of Return on Investment (RoI) crucial, which in turn necessitates efficient and adaptive resource usage. With the advent of virtualization and multi-tenancy, computing resources are shared amongst multiple tenants, preventing hard resource commitment and low server utilization. In particular, Virtual Machines (VMs) are used as fundamental entities that encapsulate a running system and abstract it from the underlying hardware physically hosting it. VMs can be statically or dynamically allocated over a DC infrastructure in order to improve application performance for the customers, and at the same time efficiently utilise the provider's physical resources and alleviate bottlenecks. Live VM migration in particular [2][3], is mainly employed to improve server-side resource usage (e.g., CPU, RAM, I/O) and to reduce power consumption at run-time [4]. Consolidation has also been suggested for reducing the number of network switches that need to be powered on at any time [5].

While server-side metrics are useful to ensure server resources are fully utilised and can be used to reduce the number of servers required to be powered on at any given time, they take no account of the resulting network congestion. Recent evidence suggests that machine virtualization can adversely impact Cloud environments, causing dramatic performance and cost variations which mainly relate to networking rather than software bottlenecks. In particular, consolidation itself has a significant impact on network congestion [6][7], especially at the core layers of DC topologies which in turn become the main bottleneck throughout the infrastructure [8], hindering efficient resource usage and consequently providers' revenue.

At the same time, Software-Defined Networking (SDN) has been penetrating such highly dynamic environments due to its centralised, network-wide abstraction of the control plane that can be exploited for fast service deployment and network virtualization [9][10]. SDN allows policies, configuration and network resource management to be applied in short timescales, and a single control protocol to implement a range of functions such as routing, traffic engineering and access control [11][5]. Most SDN controllers (e.g., *OpenDaylight*, *Ryu*, *POX*, *FloodLight*) expose APIs to configure network components, manage firewalls, get traffic counters, etc. They have also been widely used for different network-related projects such as, e.g., for complete network migration [12], new management interfaces [13], QoS management [14], or participatory networking [15].

However, current SDN interfaces are explicitly network-centric and do not inter-operate with VMs, hypervisors or other control interfaces to convey information of the temporal network state that could subsequently be exploited for admitting server resources without causing network-wide congestion and bandwidth bottlenecks [16][6][17]. As an approach to

R. Cziva, S. Jouët, and D.P. Pezaros are with the School of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK (e-mail: {richard.cziva, simon.jouet, dimitrios.pezaros}@glasgow.ac.uk).

D. Stapleton is with Brocade Communications Systems, UK (e-mail: dstaplet@brocade.com).

F.P. Tso is with the School of Computer Science, Liverpool John Moores University, Liverpool L3 3AF, UK (e-mail: p.tso@ljmu.ac.uk).

Manuscript received June 7, 2015; revised October 13, 2015; accepted January 18, 2015

overcome these inefficiencies, in this article, we present a SDN-based framework to facilitate synergistic network-server resource management over Cloud Data Center infrastructures. We exploit live VM migration in order to reduce the network-wide communication cost of the resulting traffic dynamics, and alleviate congestion of the high-cost, highly-oversubscribed links at the higher layers of the DC topology. We build on our previous work on S-CORE [18], a distributed, measurement-based live VM migration algorithm, and develop a novel VM management framework that measures temporal network load, computes the end-to-end paths of pairwise VM flows, and makes migration decisions in short timescales. Instead of using proprietary interfaces and complex extensions to server software, we extend a popular open-source SDN framework to allow inter-operation and communication of temporal, network-wide properties and static parameters between the network infrastructure and compute resources. At the same time, we explore the fine balance between the efficiency of the centralised orchestration of network-wide state, and the scalability of distributing algorithmic intelligence throughout the network to approximate computationally infeasible algorithms. We evaluate our implementation over a representative Cloud DC testbed with three different orchestration schemes, and demonstrate significant reduction in topology-wide communication cost ($>$70%) in short timescales, while migrating $<$50% of VMs and improving overall throughput by a factor of six.

Our work provides important insight into two areas of DC resource provisioning that currently attract significant attention: first, we develop and evaluate a measurement-based resource provisioning closed loop that alleviates the need for slow-evolving and expensive model-based demand prediction to achieve sustainable performance [8]. And second, we explore SDN as the basis of a unified control plane for collocated ICT infrastructures that synergistically manages the allocation of both network and server resources in order to offer predictable services even during short term, high utilization fluctuations.

The remainder of this article is structured as follows: Section II outlines the S-CORE VM migration algorithm and highlights the components that interface with SDN for enabling converged resource management. Section III presents the system architecture and the implementation of diverse orchestration algorithms. Section IV describes the experimental parameters and results as well as S-CORE's improvement in network-wide communication cost reduction and link utilization. Section V outlines related work. Finally, Section VI concludes the paper.

II. Distributed VM Migration

S-CORE is a communication cost reduction scheme that exploits live VM migration to minimise the overall communication footprint of active traffic flows over a DC topology, based on locally available information [18][20]. In a DC network hierarchy, the links situated closer to the core are typically heavily over-subscribed and subject to congestion even when spare capacity exists in other segments of the topology [16].

A typical means for distinguishing links based on this notion of cost is to associate a weight metric for each link and subsequently use aggregate weightings multiplied with the temporal bandwidth utilization to determine the overall communication cost for a given flow. S-CORE then uses this derived value to migrate VMs to other hypervisors that result in utilising links with smaller weightings.

A. S-CORE Algorithm

Link utilization is dictated by the intensity of pairwise traffic between VMs. Let $\lambda(u,v)$ denote the average *traffic load* per time unit exchanged between VMs u and v (incoming and outgoing), over a certain time window. We compute the cost of non-collocated VMs, i.e., VMs whose pairwise traffic flows are routed through at least one level of switches in the topology. For VMs u and v, level $\ell^{\mathcal{A}}(u,v) = 1$, if data is exchanged over two links, i.e., over a Top-of-Rack (ToR) switch, as illustrated in Figure 1 for two representative DC network topologies. The corresponding link weight for using each link is c_1. For each of the links, the product $\lambda(u,v)\mathsf{c}_1$ corresponds to a weighted communication cost for utilising the particular 1-level link. Similarly, if the flow is routed through level 2 of the network hierarchy (i.e., $\ell^{\mathcal{A}}(u,v) = 2$), data exchanges take place over four links, two being 2-level (weight c_2) and two 1-level (weight c_1) links. In general, when the communication among two VMs u and v is of level $\ell^{\mathcal{A}}(u,v)$, the communication cost corresponds to $2\lambda(u,v)\sum_{i=1}^{\ell^{\mathcal{A}}(u,v)}\mathsf{c}_i$. Given that any VM u communicates with all VMs in a set $\mathbb{V}_u$, there is a *communication cost*, denoted by $C^{\mathcal{A}}(u)$, attributed to VM u, for a given overall VM allocation $\mathcal{A}$,

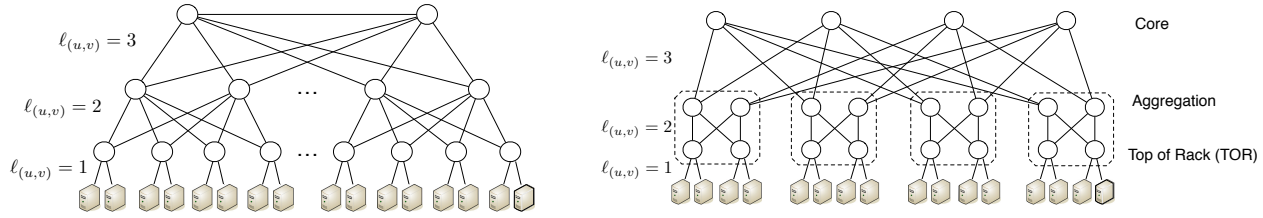
$$C^{\mathcal{A}}(u) = 2\sum_{\forall v\in\mathbb{V}_u}\lambda(u,v)\sum_{i=1}^{\ell^{\mathcal{A}}(u,v)}\mathsf{c}_i. \tag{1}$$

We can derive an expression with respect to the *overall communication cost*, $C^{\mathcal{A}}$, for all VM-to-VM communication over the DC:

$$C^{\mathcal{A}} = 2\sum_{\forall u\in\mathbb{V}}\sum_{\forall v\in\mathbb{V}_u}\lambda(u,v)\sum_{i=1}^{\ell^{\mathcal{A}}(u,v)}\mathsf{c}_i. \tag{2}$$

Eq. (2) does not take into account traffic in or out of the DC. For this case, any shortest path is along ToR, aggregation and core switches for any allocation $\mathcal{A}$. In order to derive a particular allocation $\mathcal{A}_{opt}$ for which the overall communication cost is minimised (i.e., optimal), it is required that $C^{\mathcal{A}_{opt}} \leq C^{\mathcal{A}}$, for any possible $\mathcal{A}$. Computing such optimal allocation can be shown to be infeasible due to (i) its high complexity (given the number of permutations that must be considered in an exhaustive search approach), and (ii) the global knowledge required in a highly dynamic environment like a DC. Every time the traffic dynamics change, optimal values need to be recomputed. Obviously, such a centralised approach does not scale with the number of VMs and the size of current DC topologies.

We have therefore derived the S-CORE *distributed migration policy* which is an approximation of the optimal allocation



(a) Canonical tree (b) Fat tree

Fig. 1: The two most common DC network topologies [16][19].

and is based on local measurement of the pairwise traffic load between each VM u and the VMs it communicates with in $\mathbb{V}_u$. A VM u migrates from a server x to another server $\hat{x}$, provided that Eq. (3) is satisfied, i.e., given the locally observed traffic, a VM u individually tests the candidate servers (for new placement) and migrates only when the benefit outweighs the migration cost c_m. We refer interested readers to [18] in which we have formulated, proved and compared S-CORE against other VM migration schemes.

$$2 \sum_{\forall z \in \mathbb{V}_u} \lambda(z,u) \left(\sum_{i=1}^{\ell^{\mathcal{A}}(z,u)} \mathsf{c}_i - \sum_{i=1}^{\ell^{\mathcal{A}_{u \to \hat{x}}}(z,u)} \mathsf{c}_i \right) > c_m. \quad (3)$$

B. SDN Dependencies

S-CORE's migration decision process is shown in Algorithm 1. Although a fully distributed prototype implementation of the algorithm based solely on information available locally at each VM (by running the Algorithm at each VM) has been implemented in [20], it inevitably results in a static, non-extensible deployment that does not take full advantage of network-wide resource utilization. First, a VM-based implementation would duplicate effort in measuring per-flow traffic load at each VM, as each traffic flow would be counted at both source and destination VMs. Second, cost values against which each migration decision should be evaluated would have to be manually set at each VM and would be very hard to change throughout the DC, should a service provider wish to alter them to reflect a different cost policy or function. Most importantly, the entire network topology would have to be fed into each VM u, in order to be able to compute the communication level values based on which layer of the network hierarchy flows to each other VM in $\mathbb{V}_u$ are routed through. This would couple the entire system too tightly with a given topology and, although the algorithm itself is topology-neutral, it would be too costly to deploy in diverse DCs given the (hundreds of) thousands of VMs that would need to be updated.

In an SDN-enabled environment, all the above information is either readily available in-the-network or can be retrieved centrally and then efficiently propagated throughout the entire topology, while the core of the algorithm still retains its scalable and distributed nature. In particular, looking more closely at Algorithm 1, the *getFlows(VM_IP)* method (line #2) can exploit the SDN API to obtain flow information for a given IP address from switches that retain active flow tables, thereby giving the hypervisor knowledge of all active flows of all collocated VMs. Weights must be assigned to all network links to calculate communication costs and hence determine whether migration is worthwhile. Instead of obtaining link weights for VM pairs in a static manner relying on a table instantiated at startup (line #6), SDN can be used to programmatically calculate link weights when necessary, as the controller maintains a real-time view of the network links and their status. If a link was to fail, the controller can compensate it by adjusting other relevant link weights accordingly to avoid other problems such as, e.g., logical over-subscription on other links. Another drawback of a static lookup of the weights is the inability to account for new VMs that are created on-demand in a DC and therefore would not be included in the weight table.

Orchestration based on which individual VMs make a unilateral decision on whether to migrate at a particular run of the algorithm is a crucial part of the implementation. In a static environment, a token mechanism that orders VMs based on some metric can be used, however this would impose additional requirement for network configuration to enable all hypervisors to send and receive tokens [18]. Instead, SDN handles dynamic environments too, as it monitors and reacts to real-time changes and automatically updates the relevant network parameters. As part of the migration decision calculation, the link weight for potential paths (if a VM was to be migrated) is also required to work out if the migration will result in the highest cost saving (line #17). In SDN, the logically centralised control plane can possess topology information allowing the link weight for any given path to be retrieved with minimal additional computation.

III. System Design

To overcome the challenges mentioned in Section II, an SDN framework has been extended to support VM management. Enabling VM managers to access a decoupled network control plane through a logically centralized software controller gives a new rise to network-aware VM management algorithms:

Algorithm 1 Algorithm for migration decision

Require: location ▷ location of the current VM
1: $totalCost \leftarrow 0$
2: $flows \leftarrow$ GETFLOWS(VM_IP)
3: **for all** $flows$ **do**
4: $bytes \leftarrow$ GETFLOWBYTES$(flow)$
5: $dest \leftarrow$ GETDESTLOC$(flow)$
6: $weight \leftarrow$ GETLINKWEIGHT$(location, dest)$
7: $commCost \leftarrow bytes \times weight$
8: $totalCost \leftarrow totalCost + commCost$
9: **end for**
10: $flow, cost \leftarrow$ GETHIGHESTCOMMFLOW$(flows)$
11: **while** $cost! = 0$ **do**
12: $newLocation \leftarrow$ GETDESTLOC$(flow)$
13: $newTotalCost \leftarrow 0$
14: **for all** $flows$ **do**
15: $bytes \leftarrow$ GETFLOWBYTES$(flow)$
16: $dest \leftarrow$ GETDEST$(flow)$
17: $weight \leftarrow$ GETWEIGHT$(newLocation, dest)$
18: $commCost \leftarrow bytes \times weight$
19: $newTotalCost \leftarrow newTotalCost + commCost$
20: **end for**
21: **if** $newTotalCost < totalCost$ **then**
22: **return** $newLocation$ ▷ migrate!
23: **end if**
24: $flow, cost \leftarrow$ GETHIGHESTCOMMFLOW$(flows)$
25: **end while**

- The programmable nature of the network means it can dynamically adapt to changing traffic and therefore adjust the applied network policies in short timescales, such as by reassigning the network routes and link priorities.
- The logical centralization of the network control plane makes it much simpler to query the global state. The controller can keep a view of the entire topology from active switch connections and link discovery events, and can also request individual switches for aggregated port or flow statistics on demand.
- The complexity of network devices is reduced since they only need to be optimized for data plane performance, thus for matching packets in the flow table and forwarding them to the right port.

A. System Architecture

Our system has been designed to follow today's commercial (e.g., VMware, HyperV) and open-source (e.g., OpenStack, XEN, Eucalyptus) Cloud systems in terms of virtualization, VM management and networking. Figure 2 presents the high-level architecture of the proposed system unifying control over server and network types of resources. The following subsections provide details of the key aspects.

Server Virtualization: Our implementation[1] relies on Libvirt[2], the most popular open-source VM management API supporting many hypervisors and virtualization technologies. It is also the default driver for OpenStack, a popular open-source Cloud platform. The orchestration software, detailed in section III-C connects to Libvirt daemons running on the hypervisors to start, stop and migrate VMs. For the migration, live-migration (migrating without any downtime) is performed with the copy of the entire disk at every migration. No bespoke features of Libvirt have been used in this implementation, therefore other virtualization and migration techniques (e.g. using VMware's API) could easily be adopted without significant changes to the controller modules or to the migration algorithms detailed in Section III-C.

[1]https://github.com/simon-jouet/sdnscore
[2]http://libvirt.org

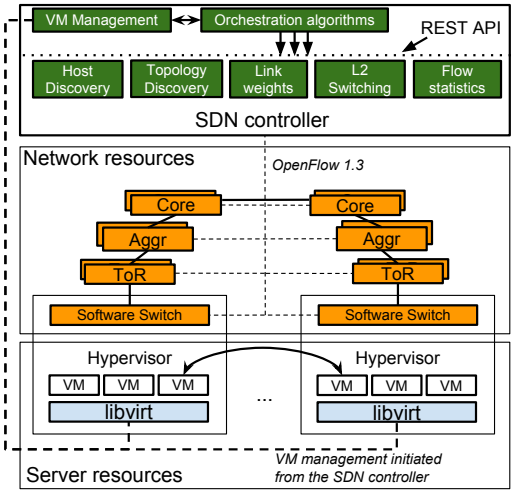


Fig. 2: High-level system architecture.

OpenFlow Networking: The switching fabric of our system has been designed using OpenFlow switches connected through an Out-of-Band (OoB) network to the controller. OpenFlow [21] is the most popular realization of today's SDN, providing an open protocol to manage the data plane of switches allowing simple match-action flow entries to control forwarding decisions. At the hypervisors, Open vSwitch is used and connected to the central SDN controller, allowing flow statistics between co-located VMs to be retrieved. Open vSwitch has been selected as the software switch due to its widespread use, support for the latest OpenFlow specifications and low resource consumption. It is also the default software switch for OpenStack and XEN's Cloud Platform to manage virtual networks and interfaces at the hypervisors.

The SDN-controlled data network that carries the traffic between communicating VMs has been configured with a *flat-addressing scheme*, where both physical and virtual machines get allocated IPs from the same range. This decision has been made in order to simplify the switching module explained in Section III-B. Although there are many other networking schemes used in production Cloud DCs (such as GRE tunnels, NATs, etc), OpenFlow's multiple flow tables can be used to perform both VM-to-VM traffic accounting in one table and, e.g., local NATing and forwarding in the remaining tables.

TABLE I: Ryu events used by our prototype system

Event	Origin	Description
SwitchEnter SwitchLeave	OpenFlow Switch	Occurs on switch-to-controller connection establishment and closure.
SwitchFeatures	OpenFlow Switch	Informs the controller of the features the connected switch supports.
LinkAdd LinkDelete	Topology Discovery	Informs listeners that a link has been added or removed.
PacketIn	OpenFlow Switch	OpenFlow *asynchronous* message sent in case the packet has not matched any flow.
FlowStatsReply	OpenFlow Switch	Response of a FlowStatsRequest containing switch's per flow statistics.

SDN Controller: The Ryu SDN controller has been used as the base platform for this framework due to its popularity, support for the latest OpenFlow specifications and its active development community. Its component and event-based architecture provides an easy framework to rapidly implement custom modules for it in Python. It has been designed to support multiple network device management protocols including OpenFlow, Netconf and OF-config.

Our SDN controller has two responsibilities. First, it communicates with the switches to set routing and forwarding information through a southbound API, using the OpenFlow protocol. Secondly, the northbound API allows third party applications to query or alter global state information, such as network topology or installed flows. Ryu relies on Python's Web Server Gateway Interface (WSGI) for modules to define new REST API endpoints and to provide interaction with third party applications. Our system utilizes Ryu's publish / subscribe paradigm and relies on the standard OpenFlow 1.3 protocol [22][21] between switches and the controller. Table I summarizes the events our controller modules subscribe to.

B. Controller Modules

In this Section, we describe the modules designed to form an API to access temporal network statistics.

Topology Discovery: The knowledge of the temporal network topology is crucial for the proposed system to calculate network-wide properties. The Topology Discovery module is bundled with Ryu and used to construct the network topology by detecting links between switches. It works by utilising the OpenFlow Discovery Protocol (OFDP), which relies on the well-established Link Layer Discovery Protocol (LLDP) [23] with minor modifications in order to forward the LLDP information on all ports of the OpenFlow switches. Through the *SwitchEnter* and *SwitchLeave* events triggered when a new OpenFlow connection from a switch to the controller is established, individual active switches within the topology can be accounted for. Subsequently, *LinkAdd* and *LinkDelete* events are triggered on addition and removal of network links, specifying interconnected switches and physical ports used.

Host Discovery: The host discovery module locates compute hosts in the network topology. Individual hosts are discovered when they start transmitting data and trigger a *PacketIn* event at the first switch. While this is sufficient for most SDN deployments, our system also needs to account for the hosts (both VMs and hypervisors) that have never exchanged traffic (e.g. new VMs, empty hypervisors) in order to take appropriate resource management decisions. As these hosts have never triggered a *PacketIn* at a switch, we are discovering them by periodically sending ARP probe messages that they reply to.

L2 Switching: A new module has been created to provide classic layer 2 (L2) switching within the topology. This module is similar to a standard learning switch implementation with OpenFlow, however to be able to track pairwise flow statistics, our installed flow entries match for both Ethernet source and destination, while in standard L2 switch implementations they match only for destination Ethernet address. When a *PacketIn* event is triggered at the controller, the shortest path between the two endpoints is retrieved from the topology discovery module and two flow entries (one matching and forwarding packets from the source to the destination and one matching and forwarding packets from the destination to the source) are inserted in all switches along the path using *FlowMod* messages.

Flow Statistics: To calculate pairwise utilization between VMs, flow statistics are collected from all edge switches by periodically requesting OpenFlow flow statistics from them. Edge switches contain all the necessary flows as all the VM-related flows must be installed on them. A *flow stats* object contains the number of packets and bytes processed by the flow entry since the flow was installed. As both edge switches at the source and destination have similar flow entries for a particular VM-to-VM communication, therefore during collection of flow statistics, it is important to collect the metrics from the same switch for a particular VM-to-VM flow. In the current implementation, the first time a new flow is discovered from the flow statistics, the Data Path ID (DPID) of the switch is stored and subsequent measurements have to come from the same switch or will otherwise be discarded. Building on these counters, the delta of bytes transmitted between two statistic queries is calculated for a given flow as well as the average traffic rate in this time period. Since the flow's byte count is crucial, OpenFlow hard timeouts have been disabled for all installed flows to avoid flow removals by the switch. While in this implementation flow statistics are periodically requested, the latest version of OpenFlow allows triggers to be set at the switch to asynchronously push flow statistics without having to request for them explicitly from the controller. Instead of OpenFlow flow counters, one could use other techniques to collect flow statistics, such as sFlow [24] or NetFlow [25]. However, these are separate protocols that require additional configuration at the switches and the controller and are not available on all OpenFlow switches. Since OpenFlow counters deliver exactly the functionality we need, our prototype is based on them.

Link Weights: In order to represent a higher communication cost when higher layers of the topology are used to carry traffic, the link weights module assigns increasing weights as it traverses through the different layers of the network, namely the hypervisor, Top of Rack (ToR), Aggregation (Agg), and Core layers. In our implementation, the default weight of

TABLE II: Controller REST API endpoints

Method	Endpoint	Description
POST	/query	Send a flow stat request to all the edge switches discovered.
GET	/view	Retrieve the byte count, traffic rate for all the flows.
GET	/placement	Get a map of Virtual Machine to hypervisor placement
GET	/cost	Get the number of links, cost and maximum layer traversed from hypervisor to hypervisor.
POST	/discovery	Trigger an ARP probe packet for the provided IP.
GET	/hypervisors	Get the list of hypervisors MAC address and associated switch DPID.
POST	/remove	Remove a host from the topology and all installed flows associated.

communication for co-located VMs at the same hypervisor is 0 and it is increasing as a square function at each layer, giving weights 1, 4 and 9 to the ToR, Agg and Core layers, respectively. The weight assignment can be adjusted, and weights can be manually set by operators to reflect a particular optimization objective and corresponding cost function. Apart from the oversubscription ratio, link weights can also reflect, e.g., the cost of equipment (cabling, network elements) in non-oversubscribed topologies and, apart from removing congestion, the proposed system can be utilized to make VM-to-VM communication links shorter (resulting in lower latency and higher throughput between VMs), and to free network elements for energy efficiency, maintenance, etc. By applying the migration algorithm only on a subset of the VMs, properties such as the VMs' desired geographical location can be retained. This is usually required for multi-site DCs and to avoid mirrored (replicated) VMs to be co-located by the algorithm, as they intentionally rely on expensive links (such as inter-DC links).

REST API for Decoupled Orchestration: One of the key aspects of this implementation has been to decouple the controller and orchestration logic. Therefore, all the modules above provide the information required for the orchestration, but do not trigger any migration decisions. Using this approach, the same controller can be reused in multiple scenarios ranging from different hypervisors providing different VM migration APIs, to centrally placing the orchestration logic or leaving it to individual hypervisors to orchestrate the VMs they are hosting. Table II is a breakdown of the REST API endpoints exposed by the controller and necessary for the orchestration software to migrate VMs based on the current state of the network. As shown, these simple calls provide a generic way to retrieve topology and traffic information of the network, accounting all VMs and hypervisors in the infrastructure. The network data gathered from these calls can be used in various Cloud managers (e.g. OpenStack or Eucalyptus), as input parameters for VM live-migration and placement.

C. Orchestration of the VM management

Orchestration of the VM management is decoupled from our SDN controller so the migration algorithms can be altered without affecting the collection of the flow statistics, as seen in Figure 2.

Matrices: We construct three different matrices in our orchestrator using the values returned from the REST API of our SDN application, described in Section III-B. These generic matrices can be used by various orchestration algorithms to make resource management decisions.

1) Traffic Matrix: The traffic matrix is a n-by-n matrix with n being the number of VMs in the infrastructure. In Eq. 3, it represents $\lambda(z, u)$, the average traffic load between VMs u and v. It is constructed by querying the */view* endpoint, iterating over the active flows, and setting the traffic rate for all VM pairs accordingly. As the traffic is measured for VM-to-VM irrespective of the traffic direction in the current implementation, the traffic matrix is symmetric.

2) Weight Matrix: The weight matrix is of the same dimensions as the traffic matrix and is constructed by querying the */placement* and */cost* endpoints. In Eq. 3, the weight is represented by c_i. VM-to-VM weight is assigned by first retrieving the hypervisor of each VM from the */placement* endpoint and then getting the hypervisor-to-hypervisor cost from the */cost* endpoint. As the traffic rate from the traffic matrix is bidirectional and the links in our topology are symmetrical, this matrix is also symmetric.

3) Cost Matrix: The cost matrix is the matrix product of the *traffic* and *weight* matrices, and it represents the communication cost between all VM pairs in the topology as defined by Eq. 3. From the n-by-n cost matrix, we can derive the n-by-1 matrix (vector) summing the total communication cost of each VM (Eq. 1). Lastly, the overall communication cost of the network is computed as the sum of each cost in the cost matrix (Eq. 2).

Migration Algorithm Orchestration: Three orchestration algorithms have been implemented: *Round-Robin*, *Best-Fit* and *Lookahead*. Each algorithm reduces the total communication cost of the network by identifying new candidate hypervisors for VMs. By modifying the weight matrix to reflect a potential migration, the total communication cost of the network after migration can be estimated. While the algorithms provide VM-level decisions (i.e., where a particular VM should be migrated), each decision contributes towards the reduction of the network-wide communication cost (*cf.* the distributed reduction of the S-CORE algorithm in Eq. (3)).

All algorithms described below go through the VMs one-by-one and select a destination hypervisor for them. A hypervisor is selected if it is the best choice for 3 consecutive rounds with a short waiting time between rounds. By using this technique, the traffic instability (e.g., finishing a migration of a VM, setting up new flows) can be eliminated from migration decisions. This value can be changed with respect to the duration of the flows and the time it takes to migrate VMs in the infrastructure. On one hand, a small number of rounds allows the orchestration to include short-lived flows and bursty traffic in the migration decision, on the other hand a large number of rounds excludes short-lived traffic to only consider background flows.

a) Round-Robin (RR) Orchestration Scheme: RR (presented in Algorithm 2) is the simplest and least compute-intensive

Algorithm 2 Round Robin algorithm

Require: currentCost, selectedVM
Require: traffic, placement, weight ▷ Matrices
1: $currentLocation \leftarrow$ GETLOCATION$(selectedVM)$
2: $potentialDests \leftarrow hypervisors \setminus currentLocation$
3: $bestDest, bestCost \leftarrow currentLocation, currentCost$
4: **for all** $potentialDests$ **do**
5: $placement[selectedVM] \leftarrow potentialDest$
6: $cost \leftarrow$ COSTMATRIX$(traffic, placement, weight)$
7: **if** $cost < bestCost$ **then**
8: $bestDest, bestCost \leftarrow potentialDest, cost$
9: **end if**
10: **end for**
11: **return** $bestDest, bestCost$

orchestration algorithm. It iterates over the discovered virtual machines which contribute to the total cost and allows them to migrate to a different hypervisor if the expected cost after migration is lower than the current total cost. For each migration to reduce the cost as much as possible, the hypervisor which would lead to the biggest saving is selected, therefore the complexity is linear as the number of hypervisors increases. At each round, the RR orchestration scheme selects the single hypervisor with the biggest saving or nothing if other hypervisors do not provide cost saving. After three rounds of the same hypervisor being selected, the VM is migrated and the iterator moved to the next VM in the list.

b) Best-Fit (BF) Orchestration Scheme: BF (presented in Algorithm 3) orchestration scheme uses the traffic matrix and the current placement of the VM to determine which VM should be migrated to which hypervisor in order to achieve the biggest reduction in total cost after a single migration. In the RR orchestration scheme, only the currently selected VM is able to migrate to any other hypervisor and the hypervisor reducing the total cost the most will be selected. In BF, there is no concept of selected VM, any VM can be migrated to any other hypervisor. The complexity of this approach is higher as the number of computations is the product of every machine contributing to the total cost times the number of hypervisors these VMs can migrate to.

c) Lookahead (LA) Orchestration Scheme: LA (presented in Algorithm 4) orchestration scheme is highly similar to BF but instead of doing a best fit for a single migration, we consider the impact on the network after two migrations even if the first migration increases the total cost. This algorithm is the most complex and is the most compute-intensive as it is the square of the Best Fit described before. In the BF implementation, it is possible for the orchestration logic to find no single migration able to improve the cost even if the core layer is still highly utilized. For instance, if two communicating VMs are hosted on one side and two other on the opposite side, migrating a VM from one side to the other might not reduce the cost as the two VMs on the same side are also exchanging traffic therefore creating a new flows through the core layer. However, using LA, the orchestration can detect the cost benefit after migrating both VMs to the other side.

Algorithm 3 Best Fit algorithm

Require: currentCost
Require: traffic, placement, weight ▷ Matrices
1: $bestDest, bestCost \leftarrow nil, currentCost$
2: **for all** VMs **do**
3: $currentLocation \leftarrow$ GETLOCATION(VM)
4: $potentialDests \leftarrow hypervisors \setminus currentLocation$
5: **for all** $potentialDests$ **do**
6: $placement[VM] \leftarrow potentialDest$
7: $cost \leftarrow$ COSTMATRIX$(traffic, placement, weight)$
8: **if** $cost < bestCost$ **then**
9: $bestDest, bestCost \leftarrow potentialDest, cost$
10: **end if**
11: **end for**
12: **end for**
13: **return** $bestDest, bestCost$

Algorithm 4 Lookahead algorithm

Require: currentCost
Require: traffic, placement, weight ▷ Matrices

1: **function** PMIGRATIONS(placement)
2: $migrations \leftarrow$ empty list
3: **for all** VMs **do**
4: $currentLoc \leftarrow$ GETLOCATION(VM)
5: $potentialDests \leftarrow hypervisors \setminus currentLoc$
6: **for all** $potentialDests$ **do**
7: $placmt[VM] \leftarrow potentialDest$
8: $cost \leftarrow$ COSTMATRIX$(traffic, placmt, weight)$
9: $migrations.add(\{VM, dest, placmt, cost\})$
10: **end for**
11: **end for**
12: **return** $migrations$
13: **end function**

14: $bestCost \leftarrow currentCost$
15: $selectedMigrations \leftarrow$ empty list
16: **for** migr1 **in** PMIGRATIONS(placement) **do**
17: **for** migr2 **in** PMIGRATIONS(migr1.placmt) **do**
18: **if** migr2.cost $< bestCost$ **then**
19: $selectedMigrations \leftarrow [migr1, migr2]$
20: $bestCost \leftarrow$ migr2.cost
21: **end if**
22: **end for**
23: **end for**
24: **return** $selectedMigrations$

D. Operator Interface

The proposed system can be divided into two high-level components: a north-bound operator interface and a south-bound Cloud interface. The Cloud interface communicates with the network elements and the hypervisors, while the operator interface allows DC operators to manually re-assign link weights, change and set the orchestration algorithms, and start/stop the communication cost reduction algorithm,

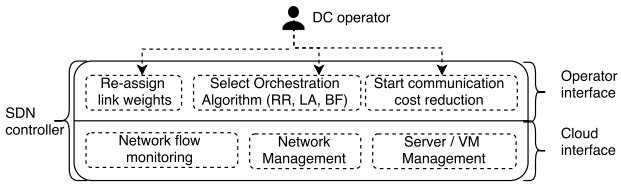


Fig. 3: Operations of the proposed system.

as shown in Figure 3. Re-assigning link weights is useful to reflect the temporal changes in the infrastructure (e.g., outages, maintenance and other operational issues) that are not discovered and handled by the SDN controller automatically.

IV. Evaluation

We have evaluated our converged, generic, SDN-based server-network management interface by using the S-CORE VM migration algorithm. Five main experiments were performed to evaluate our system. The first measures link utilization at all levels of the topology under the execution of the SDN-orchestrated live VM migration algorithm. The second experiment computes the overall communication cost of each resulting allocation according to Eq. 2, and captures how this evolves following individual VM migrations. The third experiment measures VM-to-VM communication cost throughout the run of the different algorithms (RR, BF, LA). Also, the increase of aggregate throughput and the number of migrations taken by the various algorithms has been measured.

S-CORE has been previously compared with Remedy [26], a network-aware VM management system sharing some common characteristics with our system [18]. We have implemented Remedy alongside S-CORE in *ns-3* and used a sparse traffic matrix under which Remedy achieves best results. We have demonstrated that S-CORE greatly reduces link utilization on core and aggregation links, whereas Remedy marginally alleviates core link utilization and slightly reduces aggregation. This is because Remedy tries to balance network traffic as much as possible while S-CORE takes the topology into consideration and explicitly avoids links in higher layers which are often oversubscribed.

In the following experiments, we measure the utilization on the network that carries only the traffic between VMs. The management and migration traffic has been excluded from measurements, since it depends highly on the infrastructure (e.g., OpenStack suggests separated network for VM migrations [27]), the virtualization used (e.g., full or paravirtualization, KVM or VMware VMs), the migration method applied (e.g., pre-copy methods, live migration with shared storage) and on the application workload, as reported in [28]. However, in our previous paper [18], we analyzed the migration overhead of the memory state and showed that it is negligible (1-seconds worth of transmission time over a 1 Gb/s link) for the most popular VM instance types. To account for migration cost in the system presented in this work, one can

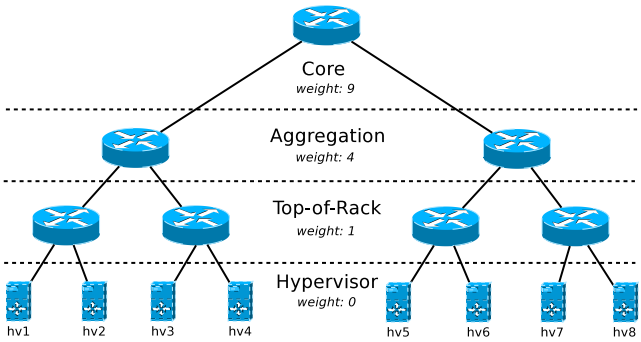


Fig. 4: Experimental network topology based on a canonical tree. In this Figure, exponential link weights are used.

TABLE III: Part of the traffic generated in our setup with initial costs and locations.

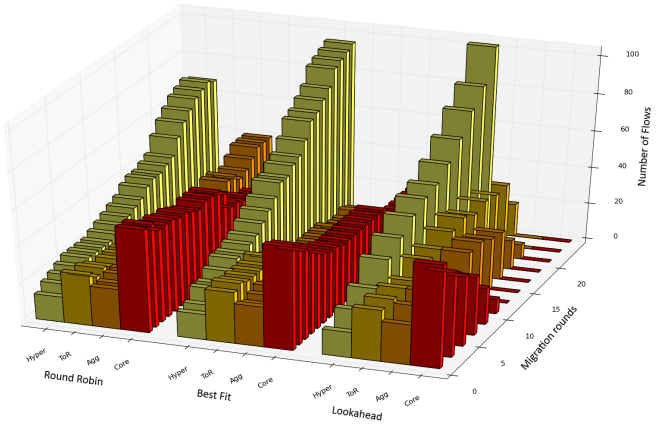
Source VM	Source HV	Dest. VM	Dest. HV	Link cost
VM1	HV1	VM2	HV1	0
VM1	HV1	VM5	HV2	2
VM1	HV1	VM23	HV8	28
VM2	HV1	VM11	HV4	10
VM2	HV1	VM20	HV7	28
VM2	HV1	VM21	HV7	28
		...		

calculate the additional cost of the migrations by multiplying the cost of links used with the amount of data to be transferred and use that in the orchestration algorithms when a potential destination is selected (note the cost parameter c_m in Eq.3).

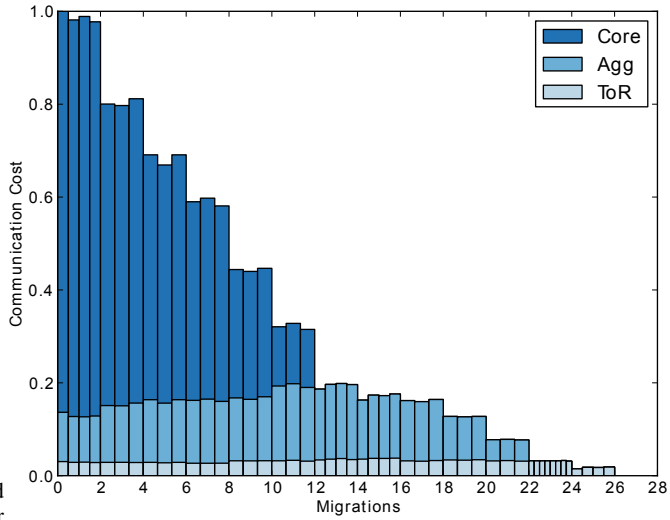
A. Experimental Set-Up

Extending our previous, Mininet-based simulation results presented in [29], we have evaluated our system on a testbed. For the experiments we used Dell servers with 16GB of memory and Intel i7-3770 2.4GHz processors, running Ubuntu 14.04. The VMs are virtualized with kernel-based virtualization (KVM) using VT-X. Each VM has Ubuntu 14.04 installed with 256MB memory allocated. Our experimental network topology is based on Cisco's reference topology, as can be seen in Figure 4. In order to represent and evaluate the increasing cost at higher layers of the topology, our network contains the three physical layers of the switching fabric with an oversubscription ratio increasing at each layer. Such topology allows us to assign distinctive weight to each layer and consequently evaluate the cost reduction achievable by our three orchestration schemes.

Initially, 3 VMs were placed at each hypervisor in the order of their ids. VMs 1, 2 and 3 are hosted on HV1, while VMs 4, 5, 6 on HV2 and so on. Traffic is generated from each VM to three other VMs. Table III shows the partial traffic matrix of the flows initiated by VM1 and VM2 to three other VMs. As shown, different VM pairs have various link costs associated to their communication, based on the partial topology over which pairwise flows are routed. In the table, we present exponential link weights 1, 4 and 9 for ToR, Aggregation, and Core layers,



(a) Comparing the three different algorithms by the highest layer used for VM-to-VM communications, *LA* reduces the utilization of the higher layers the fastest.



(b) *LA* reduces the utilization of more expensive layers first.

Fig. 5: Link utilization improvement while migrating.

respectively. For example, the link cost between VM2 and VM23 (row 3 in the table) is 28, as it uses all types of links twice and $(1+4+9)*2=28$. We set the hypervisors to host a maximum of 7 VMs to match the number of available CPU cores with one core dedicated to the hypervisor.

B. Traffic Generation

As mentioned above, traffic is generated from all VMs to three other, randomly selected VMs. For the link utilization (Section IV-C), overall communication cost (Section IV-D) and VM-to-VM cost (Section IV-F) measurements *Nping* is used to transmit MTU-sized packets at a rate of 9000 pps over TCP, resulting in an average utilization reported by similar experiments [30]. For the aggregate throughput experiment (Section IV-E), *Iperf* in TCP mode has been utilized to saturate the network with the maximum achievable bandwidth. By generating as much as possible between communicating VM pairs, the oversubscribed links throttle the speed between few VMs that is a key inefficiency in today's Cloud environments that our VM migration algorithm tackles.

C. Link Utilization

Link utilization has been measured for each layer of the network during each run of the algorithms. The calculation uses the *traffic matrix* from the orchestrator and the network topology from the SDN controller. During a run of the orchestrator, the placement and traffic matrices used for every measurement are logged. Therefore, by knowing the network topology, the location of the VMs and the traffic rate between VM pairs, it is possible to reconstruct the average link utilization between each measurement.

Figure 5a compares the three orchestration algorithms by showing the number of flows traversing each layer throughout migration rounds. For clarity, this figure counts flows by highest level layer traversed, hence, if a flow traverses the core it will be counted at the core layer, not in the three less-costly layers below. It can be seen that LA reduces the number of flows traversing the higher layers faster than RR and BF, while maximising the number of flows only communicating through the inexpensive ToR layer. It is worth noting that increasing the number of flows communicating through less expensive layers does not increase the link utilization of those layers since they were already used to carry the traffic to the higher layers before the migration occurred.

Figure 5b presents the LA algorithm's saving in overall communication cost with the layers' utilization visualized for each measurement (at least 3 measurements were used before a migration). The figure shows that the Core and Aggregation layers are not utilized after the 12th and 22nd migration out of the overall 26 migrations the algorithm took. Also, it can be seen that the first migrations significantly mitigate the use of the Core layer, while increasing the cheaper Aggregation layer's use. Once the Core layer's utilization is reduced to 0, the algorithm reduces the Aggregation layer.

D. Overall Communication Cost

The *overall communication cost*, as shown in Eq. 2, is the sum of all the individual communication costs that refer to all incoming and outgoing traffic from each VM. The costs are available in a *traffic matrix*, as described in Section III-C.

Figure 6 shows that LA reduces the overall communication cost faster than RR and BF. For the traffic generated between the 24 VMs, the optimal (minimal) overall communication

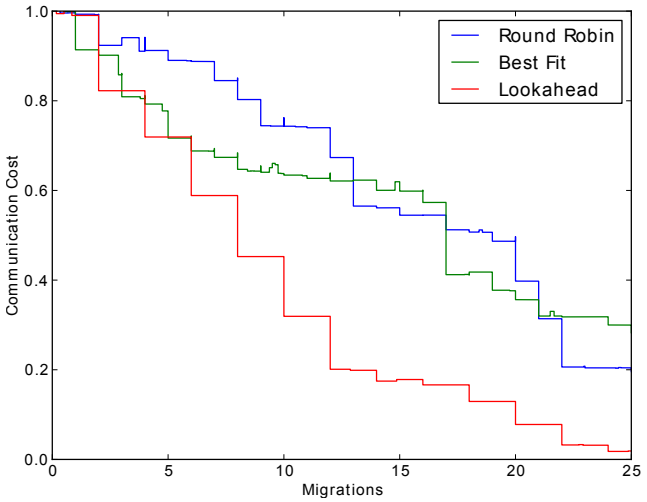


Fig. 6: Overall communication cost reduction: *Lookahead* reduces the overall cost faster, as it looks two migrations ahead before migrating.

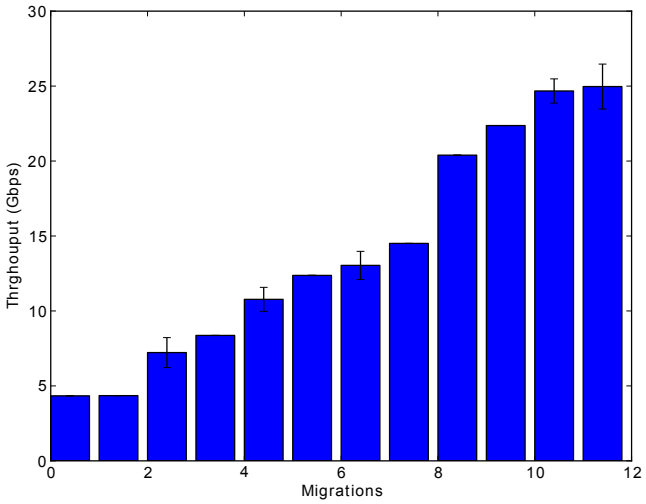


Fig. 7: Increase in aggregate throughput.

cost is zero, since communicating VMs can be colocated to hypervisors without outgoing communication. This does not mean that all 24 VMs are allocated to the same hypervisor, as the graph of communicating VMs (where nodes are VMs and edges represent communications) can be disconnected and only the connected subgraphs (VMs) need to be colocated to achieve zero communication cost. A correlation between link utilization and overall communication cost reduction can be seen in Figure 5b, as the first migration reduces the overall communication cost by approx. 20%, by reducing the utilization of the Core layer.

E. Aggregate Throughput

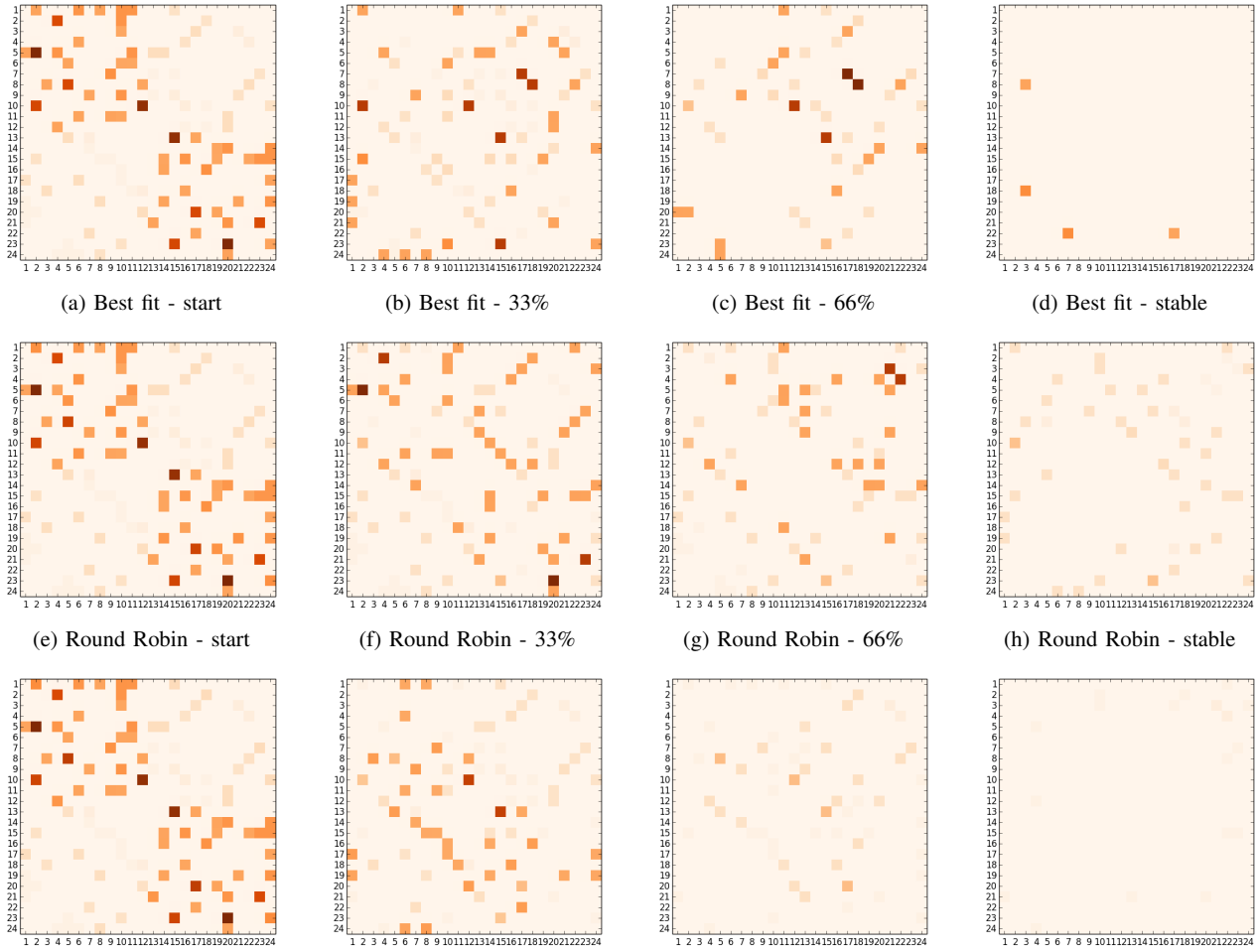
The aggregate throughput is measured by summing the maximum throughput achievable between all communicating VM pairs. In this experiment, we used *Iperf* in TCP mode. Without constraining packet size and rate, the network is always fully utilized due to the greedy nature of TCP. The traffic for VM pairs co-located on the same hypervisor has been limited to the maximum achievable bandwidth within the physical topology (1Gbps). Such limit is necessary as the transfer rate for co-located VMs through the local software switch would reach tens of Gbps, being limited only by the CPU of the hypervisor and not the network fabric, therefore biasing the overall aggregated fabric throughput. As shown in Figure 7, the aggregate throughput increases by over 6 times as a consequence of 12 live migrations.

F. VM-to-VM Communication Cost

Communication costs between VMs show the overall network load and the contribution of a particular VM pair to the overall cost. VM-to-VM costs are provided by the SDN controller as a *traffic matrix*, described in Section III-C. Figure 8 shows the reduction of the VM-to-VM costs in four heatmaps for each orchestration algorithm. The heatmaps present VM-to-VM communication cost at the beginning, 33%, 66%, and 100% of the migrations, when no further migration would reduce communication cost. As it can be seen, LA eliminates the expensive communications faster and more efficiently than RR and BF. It can be seen that, between Figures 8j and 8b, BF has 3 VM pairs with very high link cost (6 darker spots on its heatmap), while LA has only 1 VM pair after 33% of the migrations. Also, it is worth mentioning that even though our BF and RR algorithms end up with almost identical overall communication cost (as show in Section IV-D), BF leaves more dense hotspots behind. This can be explained with the behaviour of the algorithms. While BF aims for a local-optimum, greedy allocation as it always selects the VM for migration that saves the most in communication cost, while RR goes through them by order that results in a less dense utilization in our case.

TABLE IV: Number of migrations required for different orchestration algorithms

	Cost reduction (# migrations)		
Algorithms - VMs	**>20%**	**>50%**	**>70%**
Round Robin - 8	2 (21.9%)	4 (50.7%)	6 (90.8%)
Round Robin - 16	6 (23.7%)	12 (59.4%)	13 (72.6%)
Round Robin - 24	10 (21.1%)	21 (50.1%)	27 (79.0%)
Round Robin - 32	13 (23.8%)	26 (53.2%)	33 (77.9%)
Best Fit - 8	2 (29.6%)	3 (50.5%)	4 (70.5%)
Best Fit - 16	4 (20.3%)	8 (55.6%)	10 (74.9%)
Best Fit - 24	5 (25.8%)	9 (53.0%)	12 (72.2%)
Best Fit - 32	8 (22.5%)	14 (51.6%)	19 (78.0%)
Lookahead - 8	2 (35.2%)	4 (88.0%)	4 (88.0%)
Lookahead - 16	4 (36.9%)	6 (56.9%)	8 (76.1%)
Lookahead - 24	4 (34.7%)	6 (52.4%)	8 (70.4%)
Lookahead - 32	4 (23.9%)	12 (57.2%)	16 (77.9%)



(a) Best fit - start (b) Best fit - 33% (c) Best fit - 66% (d) Best fit - stable

(e) Round Robin - start (f) Round Robin - 33% (g) Round Robin - 66% (h) Round Robin - stable

(i) Lookahead - start (j) Lookahead - 33% (k) Lookahead - 66% (l) Lookahead - stable

Fig. 8: VM-to-VM communication cost for the *Best Fit*, *Round Robin* and *Lookahead* algorithms at the start, 33%, 66% of the migrations and at the end. Darker spots show higher communication cost between VM pairs.

G. Number of Migrations

We measured the number of migrations required for the different algorithms to achieve the highest cost reduction. Experiments were run with 8, 16, 24 and 32 VMs initiating 3 flows randomly to three other VMs. The traffic generated is similar to the one described in Section IV-B. The results are shown in Table IV with the number of migrations required to achieve a cost reduction by at least 20, 50 and 70 percent and the actual cost reduction in brackets. This table shows that the RR orchestration scheme does not scale well as the number of VMs increase, due to a large number of migration only marginally reducing the overall cost. RR requires 25% – 42% of the VMs to be migrated to achieve only 20% in cost reduction while BF and LA require less than 25% of VM migrations for the same 20% reduction in communication cost. Similarly, to reduce the overall cost by more than 70%, the RR algorithm had to migrate some VMs more than once with a population of 24 and 32 VMs, however BF and LA can achieve over 70% cost reduction by migrating less than 60% and 50% of the VMs, respectively. In order to present the performance of the communication cost reductions scheme in this experiment, VMs were migrated until communication cost could not be further decreased. Limiting the migrations for a particular VM can be achieved by either using a simple timestamp and ”freezing” the VM for a certain time or by adjusting the migration cost parameter (c_m) over time. In order to constrain the number of VM migrations globally, the consolidation algorithm can be executed at a slower interval.

H. Scalability

While we used a modest-scale topology in these experiments, in our previous work we have evaluated S-CORE's scale properties using large-scale simulations [18]. In these simulations, *ns-3* was used to construct canonical (2560 physical hosts; 128 ToR switches; 20 hosts per rack) and fat-tree (k=16; 1024 hosts) DC topologies (*cf.* Figure 1), and ran the simulation of the communication cost reduction algorithm. The results have shown that S-CORE achieves significant communication cost reduction (up to 87%), while incurs minimal overhead and VM downtime over large infrastructures. We would like to refer interested readers to the paper for detailed results.

It is also important to note that while a centralized SDN controller was used for our experiments, S-CORE can be distributed over multiple SDN controllers by the nature of the algorithm, as described in [18]. Multiple SDN controllers can be used with standard OpenFlow or through frameworks such as HyperFlow [31] that provides a logically centralized, but physically distributed control plane for OpenFlow.

V. Related Work

As Jennings and Stadler demonstrate in their survey [32], a comprehensive approach for joint optimization of Cloud resources is missing, despite such approach being crucial for the effective management of Cloud resources. In this work, our focus has been on the joint optimization of network and server resources by using SDN as the basis of such network-wide converged resource management system. In this Section, we discuss related work in unified Cloud resource management, topology-aware VM migration, and SDN-based resource management.

A. Unified Resource Management for the Cloud

Unified resource management involves acquiring a system-wide perspective on the allocation of the physical and virtualized resources that comprise a Cloud environment [32]. During the recent years there have been many proposals for systems and frameworks from both academia and industry. VMware's *Distributed Resource Scheduler (DRS)* [33] provides centralized control over a cluster of virtualized servers, while HP's *1000 islands architecture* [34] unifies three resource controllers that operate in different timescales. From the research community, most work has focused on the unified management of platform (e.g., power and thermal) and server (e.g., VMs, applications) resources [4] [35] [30] by extending VM management to take other objectives into account.

We took a different direction, presenting a resource management system built on top of an network-centric SDN controller and used the SDN controller as the underlying platform for a unified resource management system. By doing so, we managed to control the network infrastructure (e.g., end-to-end routing, flows, topology) and server (e.g., VMs, physical servers and applications) resources in synergy within a Cloud environment.

B. Topology-aware Virtual Machine Management

While today's VM management and consolidation algorithms typically focus on server-side resources (CPU, memory, local network interface, etc.) and do not consider the effect of migrations on the network infrastructure as a whole [36][37], few papers have addressed this issue. In [38], the authors formalize the Traffic-aware VM Placement Problem (TVMPP) that strives to reduce the aggregate network traffic by carefully selecting the optimal placement of the VMs for various topologies and traffic patterns. In [39], the authors jointly migrate VMs and manage routing in DCs to minimise traffic costs. They leverage and expand the technique of Markov approximation, provide an online algorithm and optimize the number of VM migrations required. In [40], Biran et al. focus on the satisfaction of the VM's traffic demands as well as the CPU and memory requirements by trying to derive a placement that satisfies a predicted communication demand and is also resilient to demand time-variations. Our previous work [18], presents S-CORE as underlying traffic and topology-aware VM management algorithm.

However, all these works are theoretical and provide simulated results. On the contrary, we present a real-world implementation and evaluation of such traffic and topology-aware VM management system. We also show how to utilize the SDN paradigm to manage different resources in a synergistic way.

C. SDN-based Resource Management

Remedy relies on SDN to monitor the state of network congestion by polling the per-port activity of every switch at regular time intervals [26]. Based on this aggregated information, it then estimates the cost of VM migration (number of packets to be migrated) to determine the benefits of migration in order to reduce the link utilization. Recent work by Cucinotta et al. optimizes initial VM placement for DC environments using a linear solver and SDN to gather network topology information used as a parameter in the optimization logic [41].

Despite being interoperable with SDN, the above works only exploit the logical centralization of the network's control plane for collecting flow statistics or implementing special network functions (e.g., routing, traffic engineering, and cost estimation for VM migration). On the contrary, our work in this paper exploits SDN to directly manage not only the network, but the VMs and hypervisors, therefore providing the basis for a unified and real-time resource admission and control framework for converged ICT environments.

VI. Conclusion

In this article, we presented a converged control-plane framework that integrates server and network resource management for Cloud Data Centers. We provide an SDN-based implementation for S-CORE, a scalable, topology-aware live migration algorithm that reduces the communication cost of pairwise VM traffic flows by exploiting collocation and network locality. SDN is an appropriate framework to capture network-wide state and compute utilization levels, and to disseminate them to the relevant VMs upon request.

We have extended the functionality of an SDN controller to provide flow utilisation measurement and aggregation, to expose network-wide state, and to assign weights to the links of the DC topology. For the purposes of this study, link weights reflect the bandwidth cost and the over-subscription ratio that increase when moving higher in a DC network hierarchy.

We have built a prototype system to allow flexible and platform-independent communication between the network infrastructure and the hypervisors hosting VMs in a DC. The proposed converged server-network interface has been evaluated over a scaled-down Cloud DC infrastructure, where real-world over-subscription ratios were maintained using a popular network topology and production VMs with real traffic. Our live VM management has been shown to reduce the network-wide communication cost, especially for the high-cost and congestion-prone aggregation and core layers of the DC. As a result, we show a significant reduction in congestion and an increase of overall throughput by over 6 times in our experiments, as well as, over 70% cost reduction by migrating less than 50% of the VMs.

ACKNOWLEDGMENTS

The work has been supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/L026015/1 and EP/L005255/1. The authors would like to thank Konstantinos Oikonomou from Ionian University, Greece, for his comments and suggestions on the analytical aspects of this work.

REFERENCES

[1] R. Miller, "http://www.datacenterknowledge.com/archives/2009/05/14/whos-got-the-most-web-servers/," 2013.

[2] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *USENIX NSDI'07*, 2007.

[3] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *USENIX NSDI'05*, 2005, pp. 273–286.

[4] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, vol. 28, no. 5, pp. 755 – 768, 2012, special Section: Energy efficiency in large-scale distributed systems.

[5] V. Mann, A. Kumar, P. Dutta, and S. Kalyanaraman, "VMFlow: Leveraging VM mobility to reduce network power costs in data centers," in *Proc. IFIP TC 6 Networking Conf.*, ser. LNCS, vol. 6640, pp. 198–211.

[6] G. Wang and T. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," in *Proc. IEEE INFOCOM'10*, Mar. 2010, pp. 1–9.

[7] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: comparing public cloud providers," in *Proc. ACM SIGCOMM Internet Measurement Conf. (IMC'10)*, 2010, pp. 1–14.

[8] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *Proc. ACM SIGCOMM'09*, 2009, pp. 51–62.

[9] O. N. Foundation, "Software-defined networking: The new norm for networks," Open Networking Foundation, Tech. Rep., 2012.

[10] B. N. Astuto, M. Mendonça, X. N. Nguyen, K. Obraczka, and T. Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," 2014, iEEE Communications Surveys & Tutorials.

[11] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *7th USENIX NSDI*, 2010, pp. 19–19.

[12] E. Keller, S. Ghorbani, M. Caesar, and J. Rexford, "Live migration of an entire network (and its hosts)," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM, 2012, pp. 109–114.

[13] D. Mattos, N. Fernandes, V. da Costa, L. Cardoso, M. Campista, L. H. M. K. Costa, and O. Duarte, "Omni: Openflow management infrastructure," in *Network of the Future (NOF), 2011 International Conference on the*, Nov 2011, pp. 52–56.

[14] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp, "OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks," in *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific*, 2012, pp. 1–8.

[15] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An api for application control of sdns," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, 2013, pp. 327–338.

[16] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM SIGCOMM Internet Measurement Conf. (IMC'10)*, 2010, pp. 267–280.

[17] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proc. ACM SIGCOMM Internet Measurement Conference (IMC'09)*, 2009, pp. 202–208.

[18] F. P. Tso, K. Oikonomou, E. Kavvadia, and D. P. Pezaros, "Scalable traffic-aware virtual machine management for cloud data centers," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, June 2014.

[19] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

[20] F. P. Tso, G. Hamilton, K. Oikonomou, and D. P. Pezaros, "Implementing scalable, network-aware virtual machine migration for cloud data centers," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, June 2013, pp. 557–564.

[21] 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*, vol. 38, no. 2, pp. 69–74, 2008.

[22] O. N. Foundation, "Openflow switch specification, version 1.4.0 (wire protocol 0x05)," Open Networking Foundation, Tech. Rep., October 2013.

[23] J. Hollander, *A Link Layer Discovery Protocol Fuzzer*. Computer Science Department, University of Texas at Austin, 2007.

[24] P. Phaal, S. Panchen, and N. McKee, "Inmon corporations sflow: A method for monitoring traffic in switched and routed networks," RFC 3176, Tech. Rep., 2001.

[25] B. Claise, "Cisco systems netflow services export version 9," 2004.

[26] V. Mann, A. Gupta, P. Dutta, A. Vishnoi, P. Bhattacharya, R. Poddar, and A. Iyer, "Remedy: Network-Aware Steady State VM Management for Data Centers," in *Proc. IFIP TC 6 Networking Conf.*, ser. LNCS, 2012, vol. 7289, pp. 190–204.

[27] "Openstack example architectures," http://docs.openstack.org/openstack-ops/content/example_architecture.html, last Accessed: 20-04-2015.

[28] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of virtual machine live migration in clouds: A performance evaluation," in *Cloud Computing*. Springer, 2009, pp. 254–265.

[29] R. Cziva, D. Stapleton, F. P. Tso, and D. P. Pezaros, "SDN-based Virtual Machine management for Cloud Data Centers," in *Cloud Networking*

(CloudNet), 2014 IEEE 3rd International Conference on, Oct 2014, pp. 388–394.

[30] W. Fang, X. Liang, S. Li, L. Chiaraviglio, and N. Xiong, "Vmplanner: Optimizing virtual machine placement and traffic flow routing to reduce network power costs in cloud data centers," *Computer Networks*, vol. 57, no. 1, pp. 179–196, 2013.

[31] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," ser. INM/WREN'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3.

[32] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, pp. 1–53, 2014.

[33] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu, "Vmware distributed resource management: Design, implementation, and lessons learned," *VMware Technical Journal*, vol. 1, no. 1, pp. 45–64, 2012.

[34] X. Zhu, D. Young, B. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova, "1000 islands: Integrated capacity and workload management for the next generation data center," in *ICAC '08*, June 2008, pp. 172–181.

[35] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan, "vManage: loosely coupled platform and virtualization management in data centers," in *Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 127–136.

[36] M. Wang, X. Meng, and L. Zhang, "Consolidating virtual machines with dynamic bandwidth demand in data centers," in *Proc. IEEE INFOCOM'11*, Apr. 2011, pp. 71 –75.

[37] A. Song, W. Fan, W. Wang, J. Luo, and Y. Mo, "Multi-objective virtual machine selection for migrating in virtualized data centers," in *Pervasive Computing and the Networked World*. Springer, 2013, pp. 426–438.

[38] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *INFOCOM, 2010 Proceedings IEEE*, March 2010, pp. 1–9.

[39] J. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang, "Joint vm placement and routing for data center traffic engineering," in *INFOCOM, 2012 Proceedings IEEE*, March 2012, pp. 2876–2880.

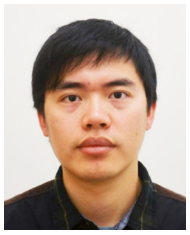
[40] O. Biran, A. Corradi, M. Fanelli, L. Foschini, A. Nus, D. Raz, and E. Silvera, "A stable network-aware vm placement for cloud systems," ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 498–506.

[41] T. Cucinotta, D. Lugones, D. Cherubini, and E. Jul, "Data centre optimisation enhanced by software defined networking," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 136–143.

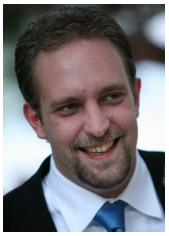
Richard Cziva (S'15) received the B.Sc. degree in Computer Engineering from the Budapest University of Technology and Economics, Hungary in 2013. He is currently a PhD student at the School of Computing Science, University of Glasgow where he previously worked as a Research Assistant. His research focuses on the efficient allocation of resources in Cloud Data Centre networks through the exploitation of converged network-server resource management and Software-Defined Networking (SDN). Recently, he has also been working on light-weight, container-based Network Function Virtualization (NFV) and increasing the programmability of next-generation networks.

Simon Jouet (S'13) received the M.Eng in Electronic and Software Engineering from the University of Glasgow in 2012. He is currently a Research Assistant at the School of Computing Science, University of Glasgow. His research focuses on the cross-layer benefits of centralized control in Cloud Data Centre in order to optimize resource utilisation, network and compute performance as well as energy efficiency. Current research focuses on the centralisation of network state, topology, routing and forwarding through Software Defined Networking (SDN) and orchestration through Network Function Virtualization (NFV).

David Stapleton received the B.Eng. degree in Electronic & Software Engineering from the University of Glasgow, UK (2014). Before his final year, he undertook an internship at Cisco Systems in Edinburgh, UK, where he worked on developing prototype onePK applications. After graduating, he worked on Cisco's XR routing platform at Reading, UK, before moving onto a new role at Brocade Communications Systems where he is now part of the team developing the next generation of the Brocade vRouter platform.



Fung Po Tso (S'09–M'11) received his BEng, MPhil and PhD degrees from City University of Hong Kong in 2006, 2007 and 2011 respectively. He is currently Lecturer in the School of Computer Science at the Liverpool John Moores University (LJMU). Prior to joining LJMU, he worked as SICSA Next Generation Internet Fellow at the School of Computing Science, University of Glasgow. His research interests include: network measurement and optimisation, cloud data centre resource management, data centre networking, software defined networking (SDN), virtualisation, distributed systems as well as mobile computing and system.



Dimitrios P. Pezaros (S'00–M'04–SM'14) received the B.Sc. (2000) and Ph.D. (2005) degrees in Computer Science from the University of Lancaster, UK. He is currently Senior Lecturer (Associate Professor) and director of the Networked Systems Research Laboratory (netlab) at the School of Computing Science, University of Glasgow, which he joined in 2009. His research is focusing on the resilient and efficient operation of future virtualised networked infrastructures through the exploitation of converged network-server resource management mechanisms, Software-Defined Networking (SDN), and Network Function Virtualisation (NFV). He has received significant funding for his research in the above areas from the UK Engineering and Physical Sciences Research Council (EPSRC), the University of Glasgow, the London Mathematical Society (LMS), and the industry. Previously, he has worked as a postdoctoral and senior research associate on a number of EPSRC and EU-funded projects in the areas of performance measurement and evaluation, network management, cross-layer optimisation, QoS analysis and modelling, and network resilience. Dimitrios has been a doctoral fellow of Agilent Technologies (2000-2004). He is a Chartered Engineer, and a Senior Member of the IEEE.