SDN-based Virtual Machine Management for Cloud Data Centers

Richard Cziva*, David Stapleton*, Fung Po Tso†, and Dimitrios P. Pezaros*
*School of Computing Science, University of Glasgow, G12 8QQ, UK
†School of Computing & Mathematical Sciences, Liverpool John Moores University, L3 3AF, UK
r.cziva.1@research.gla.ac.uk, 1004250S@student.gla.ac.uk, f.p.tso@ljmu.ac.uk, dimitrios.pezaros@glasgow.ac.uk

***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), even though network and server resources converge over the same infrastructure and typically under a single administrative entity, disjoint control mechanisms are used for their respective management.**

In this paper, we propose a unified server-network control mechanism for converged ICT environments. We present a SDN-based orchestration framework for live Virtual Machine (VM) management where server hypervisors exploit temporal network information to migrate VMs and minimize the network-wide communication cost of the resulting traffic dynamics. A prototype implementation is presented and Mininet is used to evaluate the impact of diverse orchestration algorithms.

Keywords*—*Virtual Machine Live Migration, Consolidation, Communication Cost, Data Center Network, Software Defined Networking, Mininet

I. Introduction

The advent of Cloud Computing has given rise to new and exciting prospects in the ICT world. Individuals, SMEs and large organisations can now flexibly lease processing, storage, and network resources on-demand according to their temporal needs. This has largely been made possible due to advances in virtualization technologies and in particular the Virtual Machine (VM) as a fundamental entity that encapsulates a running system and abstracts 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 paying customer, and at the same time efficiently utilize the provider's physical resources and alleviate bottlenecks. Live VM migration in particular [1] [2] is mainly employed to improve server-side resource usage (e.g., CPU, RAM, I/O) and reduce power consumption at runtime. Consolidation has also been suggested for reducing the number of network switches that need be powered on at any time [3]. However, the traffic dynamics resulting from the dynamic allocation of VMs are shown to create congestion and constitute network bandwidth as the most scarce resource in the DC [4] [5]. Network-aware VM placement research to date has only considered the original placement and typically disregards subsequent changes in traffic loads [6] [7].

At the same time, Software Defined Networking (SDN) is well-suited to operate in such highly dynamic environments as Cloud Data Centers, due to its network-wide abstraction of the control plane that can be exploited for fast service deployment and network virtualization [8] [9]. SDN aggregates the network-wide control logic into a logically centralized software component, thereby allowing for policies, configuration and network resource management to be programmed in short timescales. Most SDN controllers (*POX*, *NOX*, *FloodLight* and *Beacon*) expose APIs to configure network components, manage firewalls, get traffic counters, etc. They have also been widely utilized for different network-related projects such as, e.g., for complete network migration [10], new management interfaces [11], QoS management [12], and participatory networking [13]. However, SDN is network-centric and does not inter-operate with VMs or hypervisors to convey information of the temporal network state that could be then exploited for admitting sever resources without causing network-wide congestion and bandwidth bottlenecks [14] [4] [15].

In this paper, we propose a converged server-network control framework that exploits SDN to orchestrate 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 of a Cloud DC topology. We build on our previous work on S-CORE [16], a distributed, measurement-based live VM migration algorithm, and present a novel implementation that uses SDN to measure the temporal network load, to compute the end-to-end paths of pairwise VM flows, and to convey network-wide policies to individual VMs based on which migration decisions are made in short timescales. Instead of using proprietary interfaces, we extend the SDN framework to allow inter-operation and communication of network-wide parameters between the network infrastructure and instrumented hypervisors. We evaluate our implementation over a representative emulated Cloud DC topology in Mininet [17], and demonstrate that the algorithm can significantly reduce the topology-wide communication cost in short timescales.

The remainder of this paper is structured as follows: section II briefly describes the S-CORE VM migration algorithm and highlights the components that interface with SDN. Section III presents the system architecture and the implementation of the algorithm in Mininet. Section IV describes the experimental parameters and results, and discusses S-CORE's improvement in network-wide communication cost reduction and link utilisation. Finally, section V concludes the paper.

II. Distributed VM Migration

S-CORE [18] [16] is a scalable communication cost reduction scheme that exploits live VM migration to minimize the overall communication footprint of active traffic flows over a DC topology. 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 [14]. 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 utilizing 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 excahged over two links, i.e., over a Top-of-Rack (ToR) switch. 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 utilizing 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 $\mathcal{A}$,

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

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

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

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 minimized (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 centralized 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 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 [16] in which we have formulated and proved the S-CORE scheme.

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

B. SDN Dependencies

S-CORE's migration decision process is shown in algorithm 1. Although a fully distributed, server-only prototype implementation of the algorithm based solely on information available locally at each VM is possible, it would result in a static and non-extensible deployment that would not benefit operators nor would it take advantage of network resource virtualization. First, it would duplicate effort in measuring per-flow traffic load at each VM. Second, cost values against which each migration decision should be evaluated would have to be manually inputted 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 a SDN-enabled environment, all the above information is either readily available in-the-network or can be configured 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 for any collocated VM. Weights must be assigned to all links in the network to determine communication costs and hence determine whether migration is worthwhile. Instead of obtaining link weights (line #6) in a static manner relying on a table instantiated at startup, SDN can be used to programmatically adjust link weights when necessary. If a link was to fail, the ability to compensate for this by adjusting other relevant link weights accordingly to avoid other problems such as, e.g., logical over-subscription on other links is essential. Another drawback of a static lookup is the inability to account for new VMs. VMs are created on-demand in a DC and therefore new additions would not be included in the migration decision.

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**

Orchestration based on which individual VMs make a unilateral decision on whether to migrate at a particular run of the algorithm is a major 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 [16]. 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 centralized control plane can possess topology information allowing the link weight for any given path to be retrieved with minimal additional computation.

III. System Design

SDN has been widely deployed in DC environments, yet it is mainly used for network control and virtualization, especially in multi-tenant (e.g., to dynamically create segregated virtual networks across the DC) and location-agnostic (e.g., to create a WAN across DCs) networks [19]. In this paper, we have extended the SDN framework to support live VM management through server-network programmability. Enabling hosts to access a decoupled control plane through a logically centralized software controller gives a new rise to network-aware VM placement algorithms:

- The programmable nature of the network means it can dynamically adapt to changing traffic flows and therefore adjust the network in short timescales, such as by reassigning the network routes and link weights after the topology has changed.
- The logical centralization of the network control plane makes it much more simple to query the global state. For instance, the controller can query and aggregate port or flow statistics from any connected switch by sending a request to them.
- The complexity of networking 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 on the right port.

A. System Architecture

POX [20] is the controller development platform of choice due to its popularity, high amount of online support available and active development community. It has a modular, event-based architecture making it relatively easy to write custom modules to it in Python. Our system utilizes POX's publish / subscribe paradigm and relies on the standard OpenFlow protocol [21] [22] between switches and the controller. Table I summarizes the events our controller modules rely on.

TABLE I. POX Events Used by the S-CORE Prototype System

Event	Origin	Description
LinkEvent	Discovery module	Informs listeners that a link has been added or removed.
HostEvent	VM Tracker module	Informs listeners that a new VM has been found or a VM already known about has moved.
PacketIn	OpenFlow Switch	OpenFlow *asynchronous* message.
ConnectionUp	OpenFlow Switch	Occurs when new switch connects to controller.
FlowStatsReceived	OpenFlow Switch	Collects and maintains OpenFlow flow statistics.

B. Controller Modules

In this section, we describe the modules used by the prototype system to create an API for servers to access network information.

1) Topology Discovery: This module is bundled with POX and is used to construct the network topology. It works by utilizing the OpenFlow Discovery Protocol (OFDP), which uses the well-established Link Layer Discovery Protocol (LLDP) [23] with minor enhancements in order to forward the LLDP information on all other ports by OpenFlow switches.

2) Link Learner: Link learner's purpose is to assign weights to each link in the network. It depends on the discovery module's *LinkEvents*. When the network becomes stable, a graph traversal algorithm is used to assign incremental weights to layers 2 (ToR layer), 3 (aggregation layer) and 4 (core layer), respectively, as shown in figure 2. This weight assignment algorithm can be adjusted for assigning more complicated weightings to any other DC-network topology.

3) L2 Switching: A modified stock POX module is used to serve as the main control plane component in the network. The modification consists of a new method that takes the source and destination MAC addresses and the first and final port numbers and returns the path between them using the stored switching information in the module and the discovery module. This new functionality helps us to calculate link weightings for any given flow, which is done by the *Path monitor* module, described below.

4) Hypervisor Tracker: When a hypervisor comes online, this module stores its location within the network. Information obtained from this module is used by the network to orchestrate the decision process explained below.

5) VM Tracker: VM Tracker is used to pinpoint the physical location of any VM at any given time. Location refers to the hypervisor currently hosting the VM. In our simple model, this also gives us the ability to keep track of current hypervisor capacity in terms of the number of VMs currently being hosted.

6) Flow Statistics: This module provides flow statistics for two purposes: for the decision orchestration POX module (internal use from the network plane) and for the hypervisors to locally calculate the overall communication cost for their collocated VMs (external use from the server plane).

In order to collect flow statistics, OpenFlow's *Statistic Request* messages are periodically sent to edge switches which in turn reply with *FlowStatsReceived* events. Edge switches contain all the necessary flows (as all the VM-related flows must be installed on them), so it is needless to further propagate the query to subsequent switches in the network. A *flow stats* object contains the number of packets and bytes processed by the flow entry since the flow was installed. Building on these counters, the number of bytes transmitted every second is calculated for a given flow. Since the flow's byte count is crucial, OpenFlow hard timeouts have been disabled for all flows to avoid periodic counter resets. The hard timeout is the number of seconds after which the flow is removed from the flow table regardless of the number of matching packets. To avoid explosion of flow entries at the switches, flow tables are kept tidy by periodically removing unused flows by the controller.

7) Path Monitor: Path monitor is used to dynamically sum all link weights for any given path. It depends on *PathInstalled* events generated by the L2 switching module. When a *PathInstalled* event is triggered, the module receives the new path, traverses each link on it and obtains link weights from the Link learner module. All these values are summed, and then stored in a map. Note that only 'real' link weight costs are stored, that is, the link weight cost of flows that have been, or are currently active in the network. This is why the L2 switching module has been modified: so that a hypervisor can also retrieve the potential cost if its VM was to migrate. As previously described, the OpenFlow hard timeout has been disabled for all flows for the correct operation of the Flow Statistics module. Therefore *PathInstalled* events are received only when a new path is installed, for example when traffic generation has started on a new path or when migrations have changed the paths.

8) Migration Algorithm Orchestration: This module initializes the orchestration algorithms at the servers. Two orchestration algorithms have been implemented: *Round-Robin* and *Load-Aware*. The module heavily depends on the VM and Hypervisor Tracker modules for obtaining their locations.

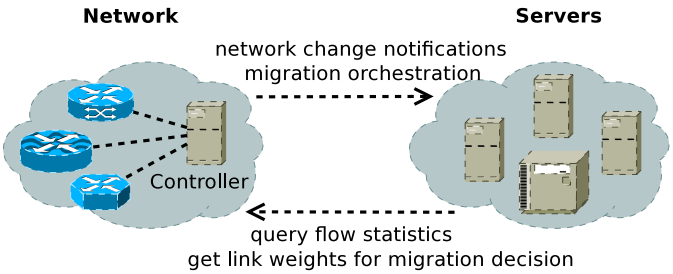

Fig. 1. Network-Server Communication: a typical communication can include decision orchestration from the network plane or gathering traffic counters from the servers.

a) Round-Robin (RR) Orchestration Scheme: RR is the simplest orchestration algorithm. It creates a list of all VMs by accessing the VM Tracker module. This list contains the MAC address of each learned VM which is then sorted in ascending order. For each MAC address, the corresponding IP address and the currently hosting hypervisor are retrieved by the *VM Tracker* and *Hypervisor Tracker* modules. The MAC addresses have been generated my Mininet from the name of the VMs, hence the order of the VMs is the ascending order of their names. Since the VMs are assigned to the hypervisors in order, RR goes from hypervisor to hypervisor ordered by their name. During the execution of the algorithm, the controller remotely invokes a method on the selected hypervisor, passing the VM's IP address as a parameter that triggers the hypervisor to execute the cost calculation for the passed IP and hence, to make the migration decision about a particular VM.

b) Load-Aware (LA) Orchestration Scheme: LA works in almost the same way as RR, except for the ordering of the VMs that are prompted to migrate. In the same way as RR, LA creates a new list of all VM MAC addresses with help from the *VM Tracker* module. However, this time, VM migration order is determined by which VM has the *highest communication cost*. This variable was formulated in section II. Since this must be calculated on a per VM basis, the controller must sequentially ask each hypervisor to calculate the communication cost for each of the VMs it hosts and return each of these values. Therefore, this module iteratively steps through each MAC address in the new list, obtains the associated IP address and hypervisor location (via the Hypervisor Tracker module) and prompts each hypervisor to calculate the total communication cost for the IP address passed as a parameter. This is achieved via Remote Method Invocation (RMI) in our prototype. After all total communication costs have been obtained, the VM with the maximum communication cost is found. Finally, the migration algorithm is remotely invoked on the hypervisor that decides about the migration of the particular VM the has the highest communication cost.

C. Communication

1) Server to Network Communication: Servers can access network information through the SDN controller using a platform-independent API. Examples of server to network communication include:

- A server (hypervisor) queries traffic usage information (both ingress and egress) for a particular VM. This is used in our current prototype where VMs query their traffic load from the SDN controller.
- A server (hypervisor) requests link costs. The most convenient way to store link cost information is at the SDN controller. This example is used to get costs for the migration decision algorithm running on the servers.
- A VM requests information about the other VMs it talks with. Since this knowledge is available in the network as flow entries at the switches, a SDN controller can collect it and serve these information for the VMs.

2) Network to Server Communication: In the current prototype, the network is responsible for initiating the migration algorithm on the hypervisors for a particular VM. The network retains all necessary information that is stored at the SDN controller to determine which VM to start the migration with, as it maintains topology, link and flow details and can retrieve traffic statistics from the switches. Using our proposed system, servers can also subscribe to any network event, such as, e.g., link change, link weight modification, etc. When the specified event occurs in the network, the SDN controller calls back the servers on a generic interface to notify that the event has happened.

IV. Evaluation

We have experimentally evaluated the SDN-based network-aware VM migration algorithm using our converged server-network management interface. As described in section III, two different orchestration algorithms have been implemented, the *Round-Robin* and the *Load-Aware*.

A. Network Modeling

Mininet [17] has been used to emulate a small-scale representative DC network infrastructure with three layers of switches and eight hypervisors. Our topology is based on the Cisco reference DC topology, as seen in figure 2. We run three virtual machines at each hypervisor and generate traffic between a subset of them. Table II contains the VMs that are used in traffic generation. We have initially allocated VMs in such a way so that pairwise traffic flows are routed through the higher layers of the topology, hence incurring high overall communication cost.

TABLE II. Initial traffic generation in our test setup.

Source VM	Source HV	Destination VM	Destination HV	Link cost
10.0.0.1	hv16	10.0.0.6	hv17	2
10.0.0.2	hv16	10.0.0.10	hv19	6
10.0.0.3	hv16	10.0.0.23	hv23	12
10.0.0.6	hv17	10.0.0.11	hv19	6
10.0.0.9	hv18	10.0.0.22	hv23	12
10.0.0.21	hv23	10.0.0.5	hv17	12

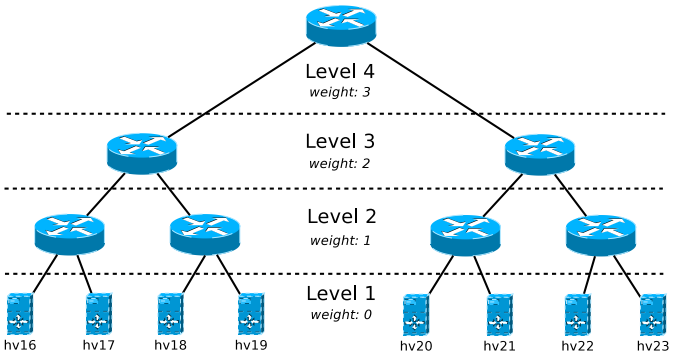

Fig. 2. Experimental network topology with four communication levels: core, aggregation, ToR, and hypervisors. Link weights are assigned by the link assigner module automatically.

B. Traffic Generation

We examined different tools and techniques for traffic generation. In our prior work, we have built a DC traffic generator [18] to create realistic DC workloads for large-scale simulation. For this emulated environment, we searched for a packet generator to create actual workloads and evaluate the SDN inter-operation, rather than the VM migration algorithm itself. We have considered *iPerf* and *Nping* and favored the latter since it does not require a running remote-side daemon and can be easily bound to a specific source port. The process generates 50-byte TCP packets at a 10 pps rate between pairs of VMs using a simple Python script, and running in the appropriate network namespace that is created by Mininet. As the emulated environment runs on a single Intel i7-3770 2.4GHz CPU, 16GB RAM server, the traffic generation rate has been kept relatively conservative without, however, influencing the validity of the experiments. In large infrastructures (such as in real DC environments), the communication between the hypervisors and a single SDN controller can stress out the network and the controller, so for these scenarios, we advocate the use of a distributed control plane for OpenFlow, such as HyperFlow [24] that can also be exploited for our system.

C. Experimental Results

Two main experiments were performed to evaluate our implementation. 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.

1) Link Utilization: To calculate link utilization, we exploit the OpenFlow *controller-to-switch port statistics request* message. This allows us to query specific switches about their ports and to gain knowledge of the received and transferred byte count. The experiments will only request this information from the switches the generated traffic is routed through. This is because at the end of each experiment, to make a comparison of both algorithms, the link utilizations are averaged out at each layer. Therefore, if the utilizations for all links were included, it would be averaging with zero values that are coming from the unused links. Furthermore, since there are more links at the lower layers of the network hierarchy (as seen

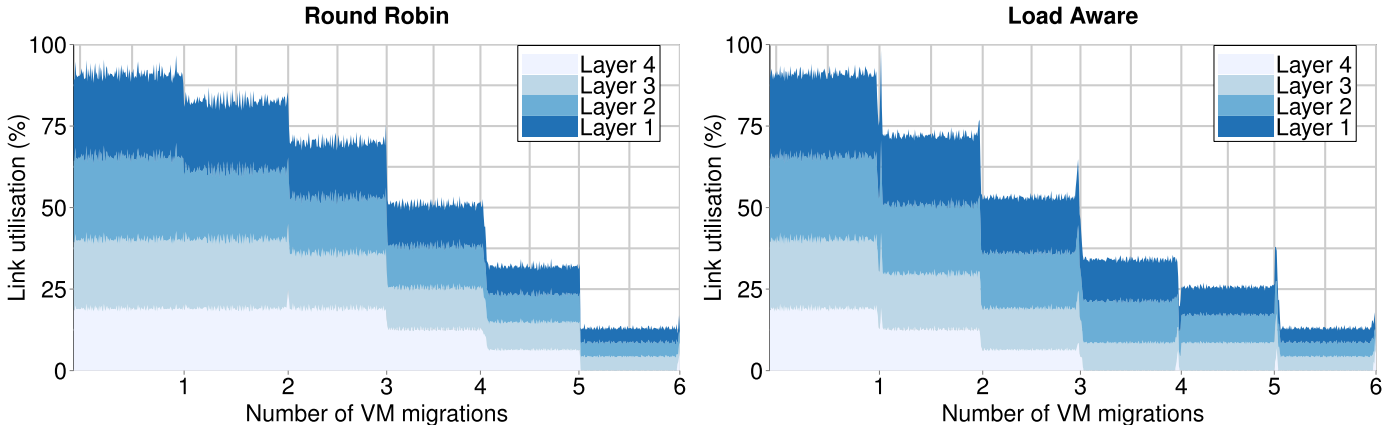

Fig. 3. Link utilization improvement during migration, using two different orchestration algorithms. *Load-Aware* orchestration reduces the utilization of the higher layers (layers 4 and 3) much faster than the *Round-Robin* does.

in figure 2), there will be more zero values to average with and hence these results would be further skewed. Link utilization also depends on the bandwidth of each link. We have used Mininet's *traffic congestion links* to specify a scaled 10Mb/s maximum bandwidth throughout the topology with adequate over-subscription ratios between the different layers of the network hierarchy. Figure 3 shows the results for the *Round-Robin (RR)* and the *Load-Aware (LA)* orchestration schemes. It can be seen that the RR policy reduces the utilization of the higher layers more slowly as it starts the migration with VMs in a load-agnostic order. Table III and IV show the step-by-step execution of the RR and LA policies, respectively. While RR takes the VMs ordered by their MAC / name (starting with VM1), LA picks them ordered by their communication load (starting with VM9 which involves flows routed via the core DC layers from hv18 to hv23, as seen in table II).

TABLE III. STEP-BY-STEP EXECUTION OF THE ROUND ROBIN POLICY

	VM1	**VM2**	**VM3**	**VM5**	**VM6**	**VM9**	**VM10**	**VM11**	**VM21**	**VM22**
	hv16	hv16	hv16	hv17	hv17	hv18	hv19	hv23	hv23	hv23
1	hv17									
2		hv19								
3			hv23							
4				hv23						
5								hv18		
6										hv19

TABLE IV. STEP-BY-STEP EXECUTION OF THE LOAD AWARE POLICY

	VM1	**VM2**	**VM3**	**VM5**	**VM6**	**VM9**	**VM10**	**VM11**	**VM21**	**VM22**
	hv16	hv16	hv16	hv17	hv17	hv18	hv19	hv23	hv23	hv23
1						hv23				
2			hv23							
3				hv22						
4					hv19					
5							hv16			
6	hv19									

2) Overall Communication Cost: The *overall communication cost* is the sum of all the individual communication costs, as shown in Eq. 2. An individual communication cost refers to all incoming and outgoing traffic from each VM. This is calculated by taking the number of bytes transferred per second per traffic flow, multiplied with the communication cost of the traffic flow. The communication cost depends on the level of the links each traffic flow is routed through (the links are assigned by our Link Learner POX module). Therefore, to calculate the overall communication cost, the controller must iteratively prompt each hypervisor via the API we installed on them to calculate the communication cost for each VM it is hosting. During the cost calculation, the hypervisors query the SDN controller to get their network traffic usage maintained by our *Flow Statistics* module. After the hypervisors complete the calculation, they return the cost to the controller that performs the simple task of summing each of these values at the decision orchestration module. Finally, the decision orchestration module triggers the migration on the selected hypervisor. In figure 4, it can be clearly seen that the LA algorithm reduces the overall communication cost faster than the RR. In our setup, the optimal (minimal) overall communication cost is zero, since the VMs that communicate with each other can be collocated on the same hypervisors. A correlation can be also seen between figures 4 and 3 that shows the reduction in link utilization. A successful migration reduces the overall communication cost by reducing utilization of the higher level links.

V. CONCLUSION

In this paper, we presented a converged control-plane framework that integrates VM and network resource management for Cloud Data Centers. We have provided a SDN-based implementation for S-CORE, a scalable and network-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 the POX SDN controller to provide flow utilization measurement and aggregation, to expose network-wide state, and to assign weights

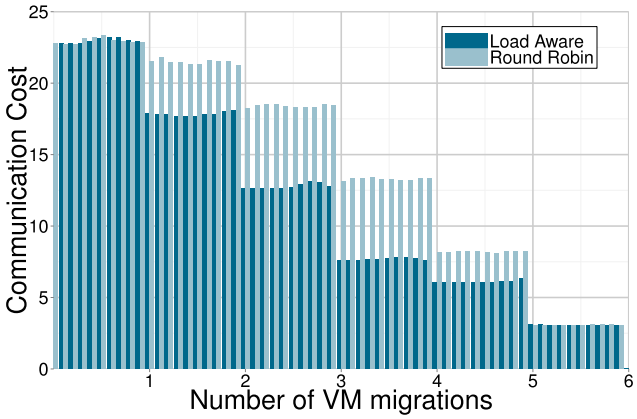

Fig. 4. Overall communication cost reduction. Two popular orchestration schemes have been compared: Round Robin and Load Aware. Load Aware reduces the overall cost faster, as it goes through the VMs ordered by the cost of their communications.

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 towards the core of a DC network hierarchy. However, link weights can be programmatically adjusted by a DC operator to reflect diverse traffic shaping policies.

We have built a prototype system to allow flexible and platform-independent communication between the network infrastructure and the servers hosting hypervisors and VMs in a DC topology. The proposed converged server-network interface has been evaluated over Mininet on a scaled-down Cloud DC network, using two different SDN-based orchestration algorithms. Live VM migration has been shown to reduce the network-wide communication cost as well as the overall link utilization, especially for the high-cost aggregation and core layers of the DC.

REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, and E. Snible, "Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement," in *Services Computing (SCC), 2011 IEEE International Conference on*, July 2011, pp. 72–79.

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

[9] 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, accepted in IEEE Communications Surveys & Tutorials To appear in IEEE Communications Surveys & Tutorials.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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 Sixth International Conference on*, June 2014.

[17] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[18] 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.

[19] M. Bari, R. Boutaba, R. Esteves, L. Granville, M. Podlesny, M. Rabbani, Q. Zhang, and M. Zhani, "Data center network virtualization: A survey," *Communications Surveys Tutorials, IEEE*, vol. 15, no. 2, pp. 909–928, Second 2013.

[20] "POX, A Python-based OpenFlow Controller," http://www.noxrepo.org/pox/about-pox/.

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

[22] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

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

[24] A. Tootoonchian and Y. Ganjali, "Hyperflow: a distributed control plane for openflow," in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*. USENIX Association, 2010, pp. 3–3.