Experimental Evaluation of SDN-Controlled, Joint Consolidation of Policies and Virtual Machines

Wajdi Hajji*, Fung Po Tso*, Lin Cui†, Dimitrios P. Pezaros‡

*Department of Computer Science, Liverpool John Moores University, UK
†Department of Computer Science, Jinan University, Guangzhou, China
‡School of Computing Science, University of Glasgow, Glasgow, UK
Email: w.hajji@2015.ljmu.ac.uk; p.tso@ljmu.ac.uk; tcuilin@jnu.edu.cn; dimitrios.pezaros@glasgow.ac.uk

***Abstract*—Middleboxes (MBs) are ubiquitous in modern data centre (DC) due to their crucial role in implementing network security, management and optimisation. In order to meet network policy's requirement on correct traversal of an ordered sequence of MBs, network administrators rely on static policy based routing or VLAN stitching to steer traffic flows. However, dynamic virtual server migration in virtual environment has greatly challenged such static traffic steering.**

In this paper, we design and implement *Sync*, an efficient and synergistic scheme to jointly consolidate network policies and virtual machines (VMs), in a readily deployable Mininet environment. We present the architecture of Sync framework and open source its code. We also extensively evaluate *Sync* over diverse workload and policies. Our results show that in an emulated DC of 686 servers, 10k VMs, 8k policies, and 100k flows, *Sync* processes a group of 900 VMs and 10 VMs in 634 seconds and 4 seconds respectively.

I. Introduction

DC management complexity has grown rapidly in recent years due to the increased virtualisation of both networking and server resources [1]. In addition, introduction of a variety of network functions or MBs as intermediaries for traffic control and shaping [2], their management has become critical since otherwise DC network security and performance can be jeopardised [3].

In such a dynamic environment, the aforementioned problems have been disjointedly addressed. For instance, several studies have focused on proposing a VM consolidation without consideration of the impact on policies or the governed traffic flows [4][5]. Others have tackled policy management issues through MB placement approaches [6]. To holistically remedy policy violation in dynamic VM migration environments, our previous work, *Sync*, a Synergistic policY and virtual machiNe Consolidation scheme jointly consolidate network policies and VMs [5] in DC environments. *Sync* has primarily focused on legacy hardware-based MBs as they are broadly adopted in today's data centres. It firstly models a network-wide communication cost with respect to MBs and VMs in tree-based topologies. Following that, it applies stable-matching approach to jointly migrate VMs and policies for reducing the network-wide communication cost while preventing policy violation.

In this paper, we aim, through a Mininet-based test-bed implementation[1], to evaluate *Sync* and understand which factors determine its performance in terms of execution time and resource consumption. Unlike ns-3 based *Sync* simulation in [5], Mininet based implementation gives realistic results and is readily deployable on real hardware[2].

The rest of the paper is organised as follows. In Sec. II, we introduce the principal algorithms that comprise its processing mechanism. Then, we present our system design in Sec. III. Particularly, we discuss the controller implementation and how SDN capabilities have been extended to reflect VMs, flows and policies characteristics. In Sec. IV, we describe our experiment set-up and evaluate *Sync* based on several criteria. We survey related works in Sec. V. Finally Sec. VI concludes the paper.

II. Sync Algorithm

Sync is a synergistic scheme for dynamic VM and policy consolidation runnable on top of an SDN-based environment. The problem formulation and the proposed model primarily deal with hardware-based MBs due to their popularity, better performance compared with their virtualised counterpart, and their flexibility and support for in-network policy and service deployment. In modelling the problem, we consider a multi-tier DC network, which is structured under a multi-root tree topology. Our experiments are running atop of k-ary fat-tree.

A. Get Communicating VM Groups

Handling all VM instances at the same time could incur an intolerable running time for Sync algorithms and it would hinder the scalability characteristics for the whole solution. In real data centres, several tenants share or own a set of VMs or resources, and there are groups of VMs that communicate between each other performing a logically similar operation. The algorithm partitions all VMs into isolated groups in which VMs do not communicate with a VM outside their group. These VM groups will be the input of other algorithms.

A group G is defined as the VMs that communicate between each other, and none has a connection/relationship with other VMs outside the group.

[1] Source code available on GitHub https://github.com/wajdihajji/sync.git

[2] https://mininet.org/

B. Policy Migration

This algorithm focuses on migrating the policies, in other words defining again the MBs; replace them with the same type of MBs as the deployed ones. In the meantime, it prepares for the VM migration by updating the *preference matrix* responsible for rating best candidate source and destination servers for VM pairs. Prior to policy migration, the algorithm should have a complete view on the *Cost Network* trees related to each flow and each policy. The *Cost Network* graphs will be the search space of the shortest paths related to policies.

The function responsible for getting the shortest path aims at reducing the Communication Cost through the migration of policies.

We define the *Communication Cost* of all traffic from VM v_i to v_j as

$$\begin{aligned} C(v_i,v_j) &= \sum_{p_k\in P(v_i,v_j)} f_k.rate \sum_{L_s\in R_k(v_i,v_j)} c_s \\ &= \sum_{p_k\in P(v_i,v_j)} \{C_k(v_i,p_k.in) \\ &\quad + \sum_{j=1}^{p_k.len-1} C_k(p_k.list[j],p_k.list[j+1]) \\ &\quad + C_k(p_k.out,v_j)\} \end{aligned} \tag{1}$$

where $C_k(v_i,p_k.in) = f_k.rate \sum_{L_s\in R(v_i,p_k.in)} c_s$ is the communication cost between v_i and $p_k.in$ for flows which matched p_k. Similarly, $C_k(p_k.out,v_j)$ is the communication cost between $p_k.out$ and v_j for p_k, and $C_k(p_k.list[j],p_k.list[j+1])$ is the communication cost between $p_k.list[j]$ and its successor MB in $p_k.list$. Note also that $R(n_i,n_j)$ is the routing path between nodes (i.e., servers, MBs or switches).

C. VM Migration

Each server has a preferred VM list (which is constructed in *Policy migration* algorithm) to host according to the corresponding *preference matrix and list*. In addition, VM migration incurs a *utility cost* depending on the server destination location. Besides, each server has a limited capacity, which determines whether it can host more VMs. These parameters are considered in the VM migration decision. Since VM and server preferences might be in some cases contradicting, a modified version of a Gale-Shapley algorithm has been adopted to address this challenge and guarantee a stable matching all the time.

The *utility* of migration $A(v_i) \rightarrow \hat{s}$ is defined to be the expected benefit through migration:

$$U(A(v_i)\rightarrow \hat{s}) = \mathcal{C}_i(A(v_i)) - \mathcal{C}_i(\hat{s}) - C_m(v_i) \tag{2}$$

where $C_m(v_i)$ is an estimated migration cost related to the VM, and $\mathcal{C}_i(s_j)$ is defined, in turn, as:

$$\mathcal{C}_i(s_j) = \sum_{p_k\in P(v_i,*)} C_k(v_i,p_k.in) + \sum_{p_k\in P(*,v_i)} C_k(v_i,p_k.out) \tag{3}$$

The VM migration algorithm, for a given VM group, initialises and obtains the preference list (where no policy violation or overused server capacity) of all servers. It sets all VMs as unmatched (no server yet chosen to migrate to). First, it starts with getting the most preferred server through calculating the migration utility and it subsequently checks that the selected server has enough capacity to host the VM. If that's the case then it moves to the next VM in the group, otherwise, it rejects less preferable VMs that were located to the server in question. Following that, it updates the *best_rejected* variable with the most preferred that has been rejected by the server. Lastly, it adds the server to the blacklists of all lower ranked VMs than *best_rejected*.

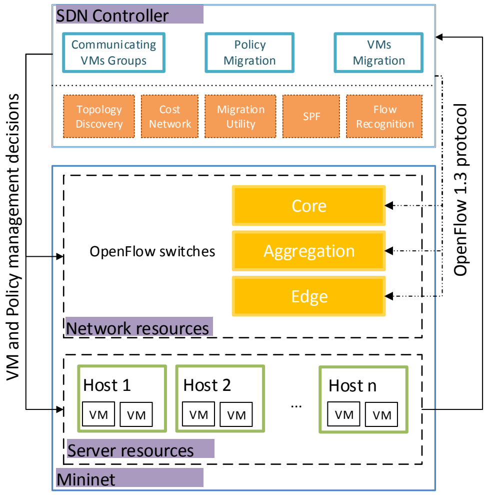


Fig. 1: Architecture design

III. System Design and Implementation

A. System Architecture

In Fig. 1, topology and the controller are running on separated environments, they communicate through OpenFlow to add rules to switches and via out-of-band control channel (network sockets) to exchange or update information related to flows, MB and VM placement, in case a migration decision is made. The controller is composed of mainly 8 modules that work collaboratively to identify VM groups, migrate VMs and policies. In Mininet, OpenFlow switches ensure the communication between VMs and servers in a Fat-tree topology.

We consider an MB as Mininet host attached to an aggregation switch. In the experiment, any type of MB only receives and forwards packets with no modification. So when a packet travels from source A to destination B through three MBs (e.g. mb1, mb2, and mb3), it only goes through and the forwarding rules are set, in advance, in the OpenFlow switches. We assume MBs are hardware-based and hence their positions are fixed. In addition, we have modelled the VM as a user process running on a Mininet host (a server in the topology), each process has an ID which is also considered as the ID of the VM. Upon creation or migration, the user process will be created or killed and instantiated accordingly. The policy is defined as a set of 3 MBs, each one governing one or many flows, which are in turn modelled as *Netperf*[3] traffic between VM pairs.

[3] http://www.netperf.org/netperf/

B. Controller Modules

In this section, we describe the main components of the controller, their roles, and the interactions between them.

1) Topology Discovery: It is a built-in feature in Ryu[4] controller. It keeps track of switches registration/de-registration and added/removed links. We construct the topology as a graph using Ryu *topology_api_app* module where vertices emulate switches and edges/links are the connections. A major limitation of this function is that it does not have a view of the instantiated VMs, flows, or policies. For this reason we have designed and created, in parallel, a communication channel to make the controller aware of the above information useful for Sync's usage.

2) Cost Network Construction: In order to make a decision of policy migration for a given flow, Sync needs to construct a *Cost Network* tree in which hosts and MBs are represented with links and corresponding weights.

For sake of improved performance, we build in advance all the *Cost Network* trees. This preliminary task is justified as the positions of MBs are meant to be fixed (hardware-based) and over a limited period of time, the flows characteristics are still unchanged. In addition, the weight of each edge can updated when used (in case flow rates are changed), and we can also prune the cost network (removing some nodes and edges) if some MBs are not available.

3) Shortest Path First (SPF): This module deals with the *Cost Network* tree of flows related to a given VM group. It gets the shortest path for a flow traversing a chain of MBs according to a specific policy. It returns the optimum positions of server source and server destination and the set of MBs in between.

4) Flow Recognition: In the controller, a flow database is built following the reception of information from the network regarding communicating VMs, therefore their IPs and the used protocol and ports are stored to match against entries in the policy database.

Flow information can be obtained by querying the network in which presumably we know in advance what traffic are initiated in a period of time, as the set of flows have been generated randomly in the experiment. In addition, Ryu can get real-time statistics by using the function "ofp_event.EventOFPFlowStatsReply".

5) Utility Of Migration: This module is essential to help with *VM Migration* decision, it aims to evaluate what is the impact on the *Communication Cost* when migrating a VM from source server to destination server. This module is used to get the *maximum utility of migration*, which helps in identifying the candidate servers for the VM to migrate to.

6) Get Communicating VM Groups: This module operates on a Python list of VMs and flows to output n VM groups, which are the input of *Policy Migration* and *VM Migration* modules.

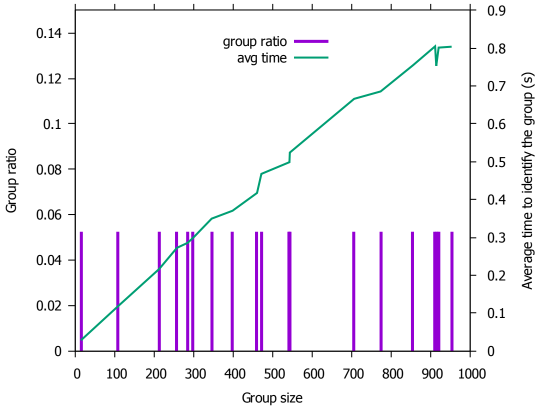


Fig. 2: Group distribution

7) Policy Migration: For a given VM group, this module works on a Python list of flows. Using output from SPF module, it migrates policies by updating the corresponding Python dictionary. Finally, it updates the *Preference Matrix* by incrementing the value that corresponds to the key (server, VM) in a Python dictionary as well.

8) VM Migration: It takes as input a VM group and outputs the new allocations for the VMs. It calls other subfunctions such as *"Get Maximum Utility"*, *"Initialise Black List"*, *"Check Server Capacity"*, *"Get Unprocessed VMs"*, and *"Obtain Preference List"*. For each VM in the group, it looks for an optimum location based on the *Utility Cost* and server capacity metrics. In the end, it constructs a Python dictionary that contains the new allocations of VMs and sends it to the topology environment via a Network Socket.

C. Communication

The communication between the topology and the controller is ensured by two channels, one via OpenFlow used by Ryu to get acknowledged of the switches and links introduced, updated, or removed, and the second one through Network Sockets used by Sync to get information on instantiated VMs, flows, MBs, and service chains (corresponding to policies).

IV. Experimental Evaluation

A. Experiment Set-up

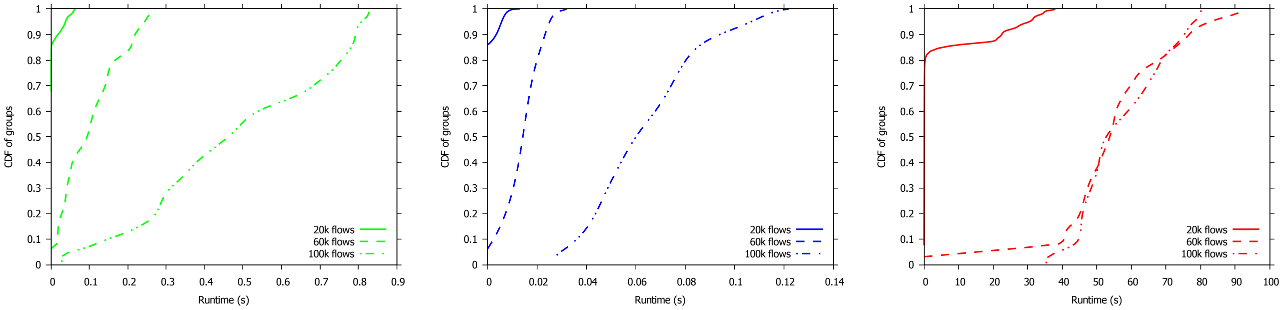
We ran our experiments on two identical servers (8 Cores/1.2Ghz and 8GB Memory). Ubuntu 14.04 is running atop of them and they belong to the same network and have directly a physical connection through a 1Gbps switch.

In server A, there are Mininet version 2.3.0d1, OpenFlow 1.3[5] and Python 2.7.6. Initially, we create and set OpenFlow switches and hosts, we also construct topology tree, VMs and flows database, which will be shared with the controller modules later on. In server B, we have installed and configured Ryu controller 4.10.

We run *Sync* with different combinations of VMs, flows, policies and MBs. We have fixed our topology size in every run with fat-tree's k=14. That means the number of edge switches equals to 98, the number of aggregation switches is 98, the

[4] https://osrg.github.io/ryu/

[5] https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf



(a) Get Communicating VMs algorithm

(b) Policy Migration

(c) VM Migration

Fig. 3: *Sync* performance evaluated with growing number of flows, group sizes in the three levels are 36, 31, 19, respectively.

number of core switches is 49, so the number of switches in total is 245, and the number of hosts is 686.

We have run all experiments 10 times to get average results so that we mitigate measurement irregularity and noisy statistical data. Variations in results can be caused by OS tasks running in background or logging processes executed to collect the results.

B. Group Formation

Sync is designed to operate on VM groups. In order to better understand the performance of *Sync*, it is important for us to show how groups are distributed in the topology. We will then show the efficiency of *Sync*'s *Getting Communicating VM Groups* algorithm for forming these groups. We particularly show results of 100k flows, 10k VMs, and 80 MBs, which is the most representative set-up in our experiment as it involves many VMs and consequently many groups.

In Fig. 2, almost every group represents 5% of the set of groups. The curve of average time to form the group evolves linearly with growing number of group sizes, that is expected since, as explained in Section II-A, the run time is affected by the group size. However, a slight dip appears for group size 915 which takes about 0.7534s to get identified. This change is due to the difference in order of appearance of groups. To explain this, we look at the group in question and its two neighbours in Fig. 2, whose sizes are 912, 915, 921 VMs, they take 0.805s, 0.7534s, 0.8017s, and their order of appearance are first, twelfth, and ninth, respectively. So group of 915 VMs appears lastly in the three groups, that means *Get Communicating VMs* operates on less number of VMs and flows at the order twelfth than at the first and ninth iterations. The aforementioned information are read from logs related to the experiment. Same explanation are still applicable on the two groups of sizes 912 and 921 VMs.

C. Overall Performance Results

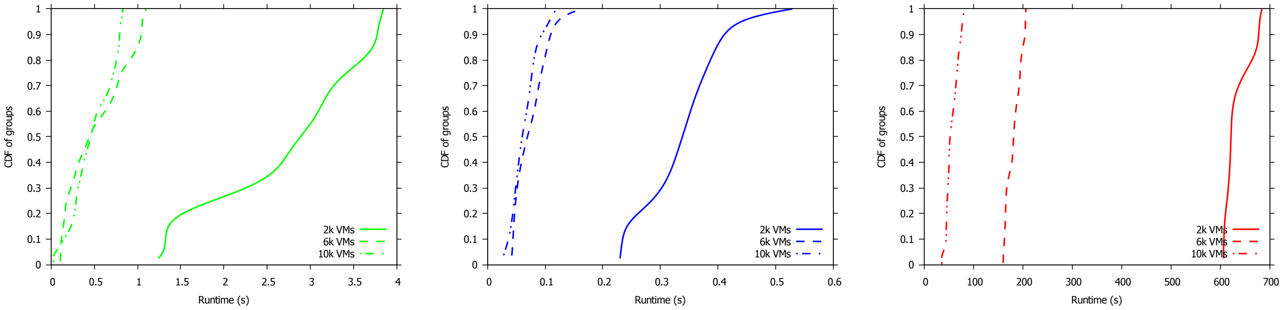
In this section, we study the impact of topology characteristics on *Sync* performance. However, we do not present the consumed network resources as *Sync* is mainly a workload intensive task, and the only network activity induced by it can be seen when sending VM and policy migration decisions to the Mininet topology.

Firstly, we fix the number of VMs and MBs (we set them at the maximum values of the experiment; 10k VMs and 80 MBs in a Fat-tree topology with k=14), at the same time, we change the number of flows starting from 20k to 100k flows. In each case, we measure the time taken for each group to run *Sync* algorithms, *Get communicating VM Groups*, *Policy migration*, and *VM migration*.

In Fig. 3, we observe how the growing number of flows causes longer runtime for *Sync* algorithms. For instance, where the number of flows is set at 20k, all groups finish in 0.08s, 0.01s, and 38s in the three algorithms respectively, at 60k flows, all of them finish in 0.28s, 0.035s, and 90s, and with 100k flows, the run-times of all groups reach nearly 0.75s, 0.12s, and 80s respectively. In *Sync* design, flows have always been involved in all algorithms. For example, in getting the communicating VMs, *Sync* looks for associated flows to each VM to conclude the relations between VMs and therefore recognise and define groups. This means that when the number of flows grows, the search space becomes larger and more importantly, the VM could have more associated flows. This also leads to an increase in the runtime of other algorithms. The discrepancy seen for VM migration when runtime is 80s for 100k flows, and 90s for 60k flows is due to the fact that the algorithm in question considers, besides the number of flows, the policy violation constraints. The latter depends on the MB positions which are initially set in a random way.

Secondly, we set the number of flows to 100k and vary the number of VMs. Fig. 4 demonstrates the results for this set of experiments. Surprisingly, we can observe that all three algorithms finish in less time for 10k VMs than for 2k VMs and 6k VMs. With 10k VMs, groups finish in 0.6s, 0.11s, and 135s for the three algorithms, respectively. In comparison, they take 3.8s, 0.51s, and 690s in 2k VMs settings. This is because when there is a large number of VMs, each VM will be source or destination for less flows than where there is a big number of flows and a small number of VMs. In the beginning of the experiment, we randomly allocate flows to VMs. This means, for example, for *Get communicating VM Groups* in the case of 2k VMs and 100k flows, *Sync* checks the flows related to a single VM, and then constructing one group will subsequently be more time-consuming.

This set of experiments has shown that the number of VMs has a measurable effect on *Get communicating VM Groups* and *Policy migration* on one hand, and *VM migration* on the other hand. In *Get Communication VM Groups* and *Policy migration*, for 6k and 10k VMs, the difference is not apparent, however, it becomes considerable in *VM Migration* algorithm.

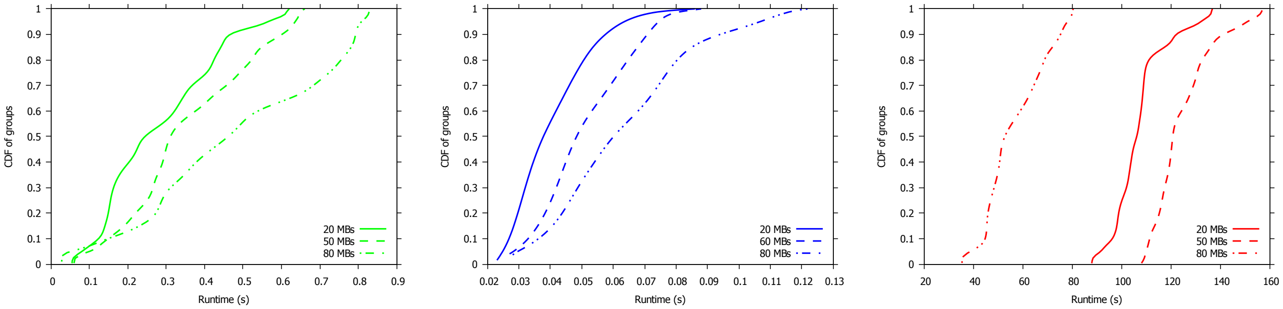


(a) Get Communicating VMs algorithm

(b) Policy Migration

(c) VM Migration

Fig. 4: *Sync* performance evaluated with growing number of VMs, group sizes in the three levels are 4, 14, 19, respectively.



(a) Get Communicating VMs algorithm

(b) Policy Migration

(c) VM Migration

Fig. 5: *Sync* performance evaluated with growing number of MBs, group sizes in the three levels are 25, 21, 19, respectively.

Thirdly, we fix the number of flows and VMs. In Fig. 5a and 5b, we observe how run time for *Get communicating VM Groups* and *Policy migration* evolves linearly with the number of MBs, albeit not too significantly. For example, *Get communicating VM Groups* finishes in 0.61s, 0.65s, and 0.81s for 20, 50, and 80 MBs, respectively. The same behaviour is recorded in *Policy Migration* algorithm. However, we do not see the same linear evolution of execution time in *VM migration*. In Fig. 5c, with 80MBs, it takes less time than for other number of MBs, and the difference is quite noticeable; 80s, 135s and 158s for 80, 20 and 50 MBs, respectively. Thus, the number of MBs have a considerable impact on all the three algorithms unlike the VMs and Flows factors. In *VM Migration* and *Policy Migration*, the number of MBs is involved directly in the processing, as in the former, it is needed to check the feasibility of the migration process, and in the latter, *Sync* will migrate MBs according to the output of *SPF* module described in section III-B3.

In addition, we have recorded the group average runtime, i.e., how much time on average groups take in each algorithm to finish processing under various settings. In Fig. 6, there are three histograms, each describes the evolution on run time based on one factor. As an example, in Fig. 6a, there are 9 boxes, the first three ones present the average runtime of a group in *Get Communicating VMs* when the number of flows evolves from 20k to 60k, to 80k flows (that correspond to the three levels level 0, level 1, and level 2). The second three boxes are for VM levels (2k, 6k, and 10k VMs), and the last three ones for MB levels (20, 50, and 80 MBs).

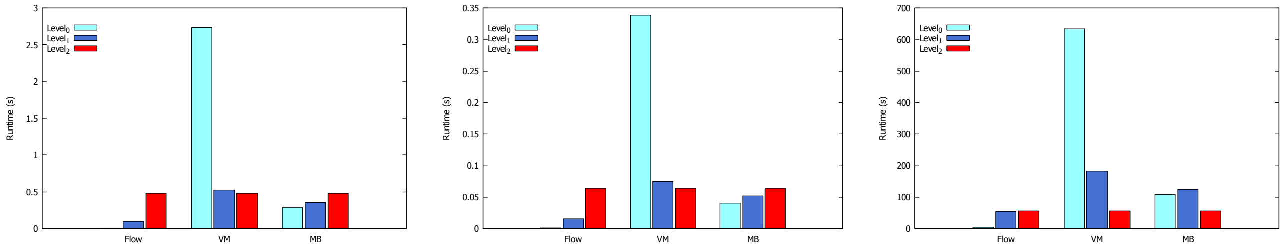
In *Get communicating VM Groups*, as shown in Fig. 6a, *Sync* is more sensitive to the number of flows that other factors, but in case the number of VMs is relatively small, the run time increases dramatically to reach 2.6s when the number of VMs, flows, and MBs are set to 2k, 100k, and 80, respectively. Otherwise, the execution time is at most at 0.5s in all other cases and it is, remarkably, at 0.00434s when the number of flows is at 20k.

In *VM Migration*, the number of VMs has a major effect on the runtime of a group, for instance, when the number of VMs as 20k, the algorithm takes nearly 600s, whereas, in case there are 10k VMs, the execution time falls dramatically to reach about 50s.

To conclude, the three factors have a different impact on the *Sync* algorithms, flows impacts more *Get Communicating VMs* and *Policy Migration* algorithms, while the number of VMs can alter significantly the time needed by *VM migration* algorithm. Lastly, the number of MBs has a known effect on *Get communicating VM Groups* and *Policy migration*, whereas, in *VM migration*, its impact becomes unpredictable because VM migration decision depends more on policy violation prevention strategy.

D. Resources Utilisation

In all experiments, CPU usage has been nearly at 13%, however, the memory consumption depends on the three input of *Sync* algorithms (number of flows, VMs, and MBs). The active memory increases linearly with the growing number of the aforementioned factors, for example, it reaches 3700 Mbytes in "extreme" set-ups (all values set at the maximum of the experiment). We also remark that memory usage grows



(a) Get Communicating VMs algorithm

(b) Policy Migration

(c) VM Migration

Fig. 6: Group average runtime measured with growing number of VMs, Flows, and MBs

significantly with increasing number of VMs, and this is explained by the fact that each VM possesses much information that comes with (e.g. associated flows). For other factors, the increase in memory is relatively limited (nearly 100 MBs). Active memory consumption raises with larger topology, but the CPU usage stands at the same level i.e. nearly 13%.

This means that *Sync* is very resource efficient and has room to scale to much bigger topologies. We also note that our implementation is a reference implementation that does not consider optimisation techniques such as parallelism with multi-controller paradigm, in which multiple controllers can process individual groups concurrently.

V. Related Work

VM and NF management has so far been broadly seen as two distinct problems treated separately. Each one of them has its benefits on the whole DC network performance. On the one hand, VMs placement has been thought of as one of DC management knobs that could improve general network performance. For instance, Net-Cohort [7] has aimed at reducing the bisection bandwidth utilisation by proposing VM ensembles detection and placement based on information collected about VM network interactions. On the other hand, Policy and MB management has gained a momentum in the last few years because of the emergence of SDN and NFV technologies. For example, in order to reduce the number of rules implemented on SDN switches responsible for traffic steering, [8] has proposed a MB placement algorithm, and this has led to reducing the ping-pong traffic.

Similar to our work, PLAN [9] proposed a joint policy and network-aware VM migration scheme but it does not deal with network policies. To prevent policy violation, PACE [10] presented an online algorithm for application mapping onto DC topology, however, it only considers one-off VM placement, which poses a problem of adaptability with dynamic workloads.

VI. Conclusion and Future Work

Sync has provided a novel approach to improving DC network performance by considering both VM and MB placement synergistic-ally. In this paper, we have designed, implemented and extensively evaluated Sync through a Mininet framework. We have found that *Sync*, which is composed of three key algorithms – Get Communicating VM Groups, Policy Migration, and VM Migration – is not only efficient but also has fractional system resource footprint. In the future work, we plan to improve *Sync*'s efficiency and performance by adopting multiple controllers in which master node will be responsible for forming VM groups and slave nodes will get fair share to continue on policy and VM migration concurrently.

VII. Acknowledgment

The work has been supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/1, EP/P004024/1, EP/L026015/1, and EP/N033957/1, Chinese National Research Fund (NSFC) Project No. 61402200, and by the European Cooperation in Science and Technology (COST) Action CA 15127: RECODIS – Resilient communication services protecting end-user applications from disaster-based failures.

References

[1] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska, "End-to-end performance isolation through virtual datacenters." in *OSDI*, 2014, pp. 233–248.

[2] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, "Stratos: A network-aware orchestration layer for middleboxes in the cloud," Technical Report, Tech. Rep., 2013.

[3] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

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

[5] L. Cui, R. Cziva, F. P. Tso, and D. P. Pezaros, "Synergistic policy and virtual machine consolidation in cloud data centers," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 2016, pp. 1–9.

[6] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," *ACM SIGCOMM computer communication review*, vol. 43, no. 4, pp. 27–38, 2013.

[7] L. Hu, K. Schwan, A. Gulati, J. Zhang, and C. Wang, "Net-cohort: Detecting and managing vm ensembles in virtualized data centers," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 3–12.

[8] A. Hirwe and K. Kataoka, "Lightchain: A lightweight optimisation of vnf placement for service chaining in nfv," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. IEEE, 2016, pp. 33–37.

[9] L. Cui, F. P. Tso, D. P. Pezaros, W. Jia, and W. Zhao, "Policy-aware virtual machine management in data center networks," in *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*. IEEE, 2015, pp. 730–731.

[10] L. E. Li, V. Liaghat, H. Zhao, M. Hajiaghayi, D. Li, G. Wilfong, Y. R. Yang, and C. Guo, "Pace: Policy-aware application cloud embedding," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 638–646.