TOR-ME: Reducing Controller Response Time Based on Rings in Software Defined Networks

Jianchun Liu
School of Computer Science and Technology
University of Science and Technology of China
Hefei, China
e-mail: lyl617@mail.ustc.edu.cn

Liusheng Huang
School of Computer Science and Technology
University of Science and Technology of China
Hefei, China
e-mail: lshuang@ustc.edu.cn

Chunming Qiao
Department of Computer Science and Engineering
State University of New York at Buffalo
NewYork, USA
e-mail: qiao@computer.org

Shishuang Wang
School of Computer Science and Technology
University of Science and Technology of China
Hefei, China
e-mail: sshuangw@mail.ustc.edu.cn

***Abstract*—In a software defined network (SDN), switches will send Packet-In messages of newly arrived flows to the controllers. With more and more flows arriving at a network, the controller load significantly increases, which may lead to control channel congestion and long controller response time. Meanwhile, to the best of our knowledge, part of these flows are elephant flows that can affect system performance (including control plane and data plane). To address this challenge, in this paper, we propose a novel Time-Optimized Ring with Minimizing Elephant flow (*TOR-ME*) scheme and present several algorithms for it. Our evaluation shows that *TOR-ME* can reduce controller response time by 54.8%, and improve the network throughput by 58.1%, when comparing with the existing solutions.**

Keywords-software defined network; ring; elephant flow; controller response time

I. INTRODUCTION

A typical Software-defined networking (SDN) is a new paradigm that separates the control and data planes on independent devices [1]. The controller monitors the network and provides centralized control by installing forwarding rules in the data plane. As specified in the OpenFlow standard [2], when a new flow arrives at a switch, the switch encapsulates the header packet into a Packet-In message and delivers it to a controller. Then, this controller determines the route of this flow, and installs flow entries on the switches along this path.

In recent years, the number of flows increases drastically in both cloud networks and Internet. On one hand, with the development of information technology, many novel network-based applications (*e.g.*, search [3] and content distribution [4]) are constantly emerging. On the other hand, some hot events will attract attention of many people through networks. For example, many audiences will watch the live broadcast of some attractive sport final through the Internet. With more new flows arriving at switches, these switches will send more Packet-In messages to the controllers in an SDN, which may lead to long and highly varied controller response time and poor user experience [5]. The controllers should respond to events such as shifts in traffic intensity, and new connection from hosts, by pushing forwarding rules to flow tables on the switches so as to achieve various performance requirements, such as load balancing. Thus, reducing controller response time help to significantly improve network performance and resource utilization.

There are there different ways to reduce the controller response time (or the controller load). The first or general way is to use multiple controllers with the static controller assignment mechanism (*e.g.*, Onix [6] and NVP [7]). Specifically, the control plane is implemented as a cluster of distributed controllers, and each switch is only connected with one controller. The switches deliver the Packet-In messages to different controllers, which helps to reduce the controller load compared with the single-controller framework. However, since the traffic in the network dynamically fluctuates in space and time, some controllers may still be heavy-loaded, or even congested [5].

The second method is the dynamic controller assignment mechanism [5] [8], which permits each switch to dynamically associate from a “heavy-loaded" controller to a “light-loaded" one, so that the maximum controller load can be reduced. For example, the authors of [5] [8] presented a near-optimal Nash stable solution for dynamic controller assignment. However, the dynamic controller assignment scheme brings some disadvantages on reassignment communication delay and control plane overhead. Specifically, when a switch is re-assigned from one controller to another, the communication delay is unavoidable for building secure connection with the newly assigned controller.

The final method is to pre-install entries for *all* flows in a network, also called *proactive* routing scheme [9]. Since each flow can match at least one pre-installed entry when arriving at a switch, the switch will not deliver any Packet-In message to the controller. Thus, the controller load is very light and the controller response time is low. However, the *proactive*

978-1-7281-2184-0/19/$31.00 ©2019 IEEE

routing can not efficiently deal with traffic and management policy dynamics. Due to dynamic traffic intensity, it may result in transient congestion on some data links and lead to packet dropping and throughput reduction.

Meanwhile, the centralized and fine-grained design on SDN flow control, however, results in serious performance and scalability bottlenecks of SDN. Some researches [10] [11] [12] show that flows in DCN (Data Center Network) can be classified as elephant flows and mice flows. The former is long-lived while the latter typically lasts less than 10s due to its small size ($\leq$10KB). Generally, elephant flows usually transfer significant amount of data in the network, such as ftp, data backup and VMotion, *e.g.*, which may tend to cause network congestion and impact the network performance. Therefore, The detection and processing of elephant flow are also crucial in efficient network management.

In this paper, we are committed to a scheme that can avoid network congestion as much as possible, which also can reduce controller response time. To achieve this goal, we propose a framework and present several algorithms for it. The main contributions of this paper are summarized as follows.

- We propose the novel Time-Optimized Ring with Minimizing Elephant flow (*TOR-ME*) framework, which can effectively overcome the above shortcomings.
- We present several algorithms for *TOR-ME*, including *Loop Seeking* algorithm, *Loop Selection* algorithm and *Overall Running* algorithm. In the *Loop Selection*, we formulate it as an ILP problem, which is NP-Hard and use an approximation algorithm of greedy heuristic to solve it.
- The simulation results show that *TOR-ME* help to improve system performance significantly, such as reducing the controller response time by 54.8% and improving the controller load ratio by 37.8% compared with the existing solutions, *etc.*

The remainder of this paper is organized as follows. We review related work in Section II and design the *TOR-ME* framework in Section III. Section IV for several appropriate algorithms of *TOR-ME*. We report our simulation results in Section V and conclude the paper in Section VI.

II. Related Works

Blocking (Control channel or Data plane) is an important issue in an SDN. To prevent it, some previous solutions about flow processing have been proposed, elephant flow especially. PMCE [13] is a parameter minimum cross entropy algorithm to find the optimal switch allocation policy for each elephant flow. Mahout [14] computes the optimal path for new flows more than 100MB by the *Global First Fit* (GFF) algorithm. Hedera [15] uses the *Simulated Annealing* (SA) algorithm to recalculate paths of elephant flows periodically. All of these approaches have their own advantages, however, the complexity of PMCE is high and the convergence speed is low and the result of the GFF algorithm depends on the arrival order of flows completely, may leading to new hotspots. The results of the SA algorithm are completely random in different periods, the controller needs to issue a large number of flow tables to the switches to adjust elephant flows' paths, increasing the additional load for the network. The common problems of above solutions based on SDN is that the massive control traffic between controllers and switches will cause network congestion.

The most related work, Hu [16] has proposed a mechanism to reduce congestion, which pre-computes a loop path for each flow. When the controller has installed forwarding rules in the switches, the flow leaves the loop and is forwarded according to these rules. Since the flows will not wait, until flow entries have been installed, it can obviously reduce congestion of network. However, resource (*e.g.*,**SRAM** and **DRAM**) constraints should be considered for the model. The SRAM size is usually limited on some commodity switches. Even in the high-end Trinder2 switch with forwarding capacity of 960GB, its SRAM size is only about 16MB, which will be further shared with routing/firewall/filter/measurement [17]. For example, it expects to store forwarding information database (FIB) with 10MB in practice [18]. In this paper, we will study how to reduce congestion and controller response time of network while conquering the above challenges and disadvantages.

III. Preliminaries

TABLE I.
Key Notations.

Symbol	Semantics
U	a cluster of SDN controllers
V	a set of switches
E	a set of network links
F	a feasible loop set
$c(e)$	the capacity of link e
α_u	the processing capacity of controller u
$\zeta_v(t)$	the number of Packet-In by switch v at time t
$\theta_u(t)$	the load on controller u at time t
$\vartheta_u(t)$	the response time of controller u at time t

A. Network Model

An SDN typically divides into the control plane and the data plane, which consists of two device sets: a cluster of controllers, $U = \{u_1, ..., u_m\}$ with $m = |U|$ and a set of SDN switches, $V = \{v_1, ..., v_n\}$, with $n = |V|$. The controllers are responsible for management of the whole network, including route selection for all flows and traffic measurement distribution. Each controller u_j has a processing capacity in terms of the requests it can handle in one unit time, denoted by α_j. The network topology from a view of the data plane can be modeled by $G = (V, E)$, where E is the set of links connecting switches. For each link $e \in E$, the capacity of link e is denoted by $c(e)$.

B. Elephant Flow Detection Mechanism

Prior works such as [9] [19] [20] hava used SDN (OpenFlow and/or P4) for elephant flow monitoring. Wang [21] provided a survey of the elephant flow detection in SDN that approaches can be classified into two kinds: detection in switch and

detection in end-host. We pay attention to detection in switch and it also can be classified into two kinds: flow-statistics based and flow-characteristics based elephant flow detection. Both of them will be used in the mechanism, which is in the judicious combination of flexible software for packet-header processing and scalable hardware for flow-counter monitoring. The software tracks all the flows uniquely identified by a 5-tuple. It maintains its internal data structures that contain the duration and volume of individual flows. Using a built-in event mechanism, the software is able to detect an elephant flow, which also can dynamically balance the load between software and hardware.

C. Controller Response Time Model [5]

We consider a discrete time model where the length of each time slot matches the timescale at which Packet-In requests of each switch can be precisely recorded. In an SDN, coordination among multiple controller is necessary to install this path. The set of its connected switches is denoted as $\mathcal{S}_u$. The number of Packet-In messages (one for each newly arrived flow), which will be delivered to the associated controller by switch v in slot t, is denoted as $\zeta_v(t)$. Then, the load of controller u is $\theta_u(t) = \sum_{v \in \mathcal{S}_u} \zeta_v(t)$.

By applying the Little's law [5] [22], the average sojourn time on controller u is $\frac{1}{\alpha_u - \theta_u(t)}$. Given that the time for computing a single-source route is $O(n^2)$ [23], where n is the number of switches in an SDN, the average response time of controller u can be expressed as:

$$\vartheta_u(t) = \frac{1}{\alpha_u - \theta_u(t)} \cdot O(n^2) \tag{1}$$

D. Design Of TOR-ME

We now present the design of *TOR-ME*. Its idea is simple: when the flow arrives at the ingress switch and the controller load has exceeded threshold δ, which is set by default, the flow will not wait, to avoid blocking. In contrast, keep it looping in the switches along the pre-computed paths and leave the loop when the routing rules are installed in the switches. At the same time, try to minimize the number of elephant flows in the loop because it will increase the load of links, compared to mice flows.

For easy understanding, we give an example, in Fig. 1, suppose that a packet from $h1$ to $h2$. Switch $v1$ will keep the packet in its buffer and forward it to the controller if buffer overflows, according to the OpenFlow standard. At the same time, $v1$ has detected whether it is elephant flow or mice flow. If it is mice flow and the controller load has exceeded δ, $v1$ just sends the first a few bytes of the packet to the controller, and directly forwards the entire packets to a pre-computed loop path, *e.g.*, $v1 \rightarrow v3 \rightarrow v4 \rightarrow v1$.

While the flow is looping, the controller issues new routing rules to update corresponding flow tables in the switches (*e.g.*, $v1$ and $v2$ in this case). The flow will be forwarded to $h2$ next time it backs to $v1$ through the path $v1 \rightarrow v2 \rightarrow h2$. This solution optimizes the controller load and response time by trading off two types of resources: (1) the bandwidth in the loop path against the bandwidth in the control channel while waiting for the new rules, and (2) the computing overhead of ingress switch that used to detect elephant flow.

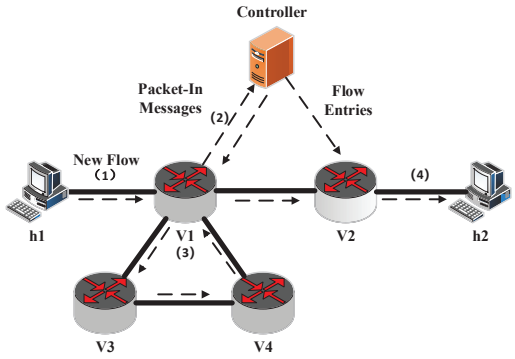

Figure 1. Basic idea of *TOR-ME*.

In addition, the *TOR-ME* is also controllable. *First*, the looping is controllable because each flow packet will eventually leave the loop path. It can stop looping and leave at any switch in the loop path. *Second*, the bandwidth occupied by the *TOR-ME* is also controllable, because of an independent traffic control queue with limited bandwidth is set in each output port of loop paths for the flow packets. The main problem we should solve is to detect whether the flow is elephant flow or not and find the paths for looping. The former will be addressed by the mechanism in Section III-B and we focus on the latter.

IV. Algorithm for *TOR-ME*

In this section, we present several algorithms used in the framework, such as *Loop Seeking* (Section IV-A), *Loop Selection* algorithms (Section IV-B) and *Overall Running* algorithm (Section IV-C).

A. Loop Seeking

As mentioned in Section III-A, the network topology is denoted as $G = (V, E)$ and we then denote a loop $l \in N^+$ is a subgraph of G which satisfies

$$V_l = \{V_{l,e}\} \subset V, e = 1, ..., |V_l|, \tag{2}$$

$$E_l = \{(v_{l,1}, v_{l,2}), ..., (v_{l,e-1}, v_{l,e}), (v_{l,e}, v_{l,1})\} \subset E, \tag{3}$$

where $(v_{l,e-1}, v_{l,e})$ denotes that a link from $v_{l,e-1}$ to $v_{l,e}$.

Johnson [24] proposed an algorithm for identifying all loops in a directed graph. It mainly uses *Depth First Search (DFS)* to find each vertex. However, this method cannot be used in our framework because the graph in the framework is undirected. However, we can borrow the basic idea of Johnson's method and use *DFS* biconnected components instead of strongly connected components. It should be noted that there may be cases where the loop does not cover some switches, so we introduce a virtual loop to cover these nodes, which is a simple palindrome path $v1 \rightarrow v2 \rightarrow ... \rightarrow v2 \rightarrow v1$.

The time complexity of the method is as large as $O((n+e)(c+1))$, where n, e, c are the number of switches, edges and loops. The complete *Loop Seeking* method is described in Algorithm. 1.

Algorithm 1: Loop Seeking Algorithm

Input: $G = (V, E), LEN_{MIN}, LEN_{MAX}$
Step 1: Get Biconnected Set;
$B \leftarrow getBiconnectedSet(G)$;
Step 2: Get Loop Set **for** *each* $B_i \in B$ **do**
  for *each* $root \in Switch(B_i)$ **do**
    RandomShuffleSwitch($B_i - \{root\}$);
    DFS(root, root, **new** Stack());
Function DFS(vertex v, root, Stack Path) : **do**;
for *each* $w \in B_i[v]$ **do**
  if $w == root \bigwedge \parallel Path \parallel +1 \geq LEN_{MIN}$ **then**
    $F \leftarrow F \bigcup \{Path \bigcup v\}$;
    if *The number of loops reaches 100* **then**
      stop DFS;
  else
    if $\parallel Path \parallel < LEN_{MAX}$ **then**
      DFS(w, root, $Path \bigcup v$);
Output: F - Feasible Loop Set for Loop Selection

A large and complex network can lead to unacceptable time to find loops, so we make **Graph Partition**. Spectral [25] proposed a method of dividing the network topology into k subgraphs, where k is the controllers' CPU cores, which will be borrowed in our framework. However, this operation will also delete some links and may impact the coverage rate of the loops. We use the switches that is not covered as the root to do re-search.

We also need to control **Loop Length** and **Loop Scale**. If the length of the loop is too short, the flow will traverse a link several times in the loop, consuming too much link bandwidth. If the length is too long, the packet still in the loop which causes extra link consumption, even if the flow entry is already installed in the switch. Therefore, it is important to control the length of the loop within a reasonable range.

It is almost impossible for the loop to cover all the switches, so we have to limit the number of loops. In order to achieve this goal, we add restrictions in DFS. Each switch will be searched as a root, and we introduce a random shuffle algorithm to increase the diversity of the loop. Under the above strict conditions, we only get 100 loops for each root search, and the experimental results show that there is little difference between our strategy and full loops searching.

B. Loop Selection

In this section, we select loops from the output of Algorithm. 1, which needs to meet several constraints and objectives. We formulate the problem as follows:

$$S.t. \begin{cases} V_1 \bigcup V_2 \bigcup, ..., \bigcup V_l = V, & \\ L_l = \sum\limits_m t_{ml} I_{ml}, & \forall l \\ C_i = \sum\limits_l a_l I_{ml}, & \forall i \\ N_i = \sum\limits_l I_{ml}, & \forall i \\ 0 \leq a_l L_l - T \leq \alpha, & \forall l \\ I_{ml} \in \{0, 1\}, & \forall m, l \end{cases} \tag{4}$$

The first set of equations is a primary requirement, which secures all the switches in the network topology being covered. Virtual loops will play an important role if physical links are not sufficient. The second set of equations expresses the time traversing the whole loop, where t_{ml} is the delay of switch m to the next hop. The third set of equations denotes that bandwidth cost on the link which packet traverses. The fourth set of equations expresses the number of the extra flow entries because loop paths should be proactively installed in the switches and it requires one more flow entry installed into the switch. The fifth set of inequalities expresses extra delay of the framework, where T represent the time interval between the switch sending the Packet-In message and the switch being updated the new flow entries from controller. Extra delay must not exceed upper bound α and should be minimized. T and t_{ml} can be measured directly and we use I_{ml} to indicate whether loop l contains switch m or not.

Algorithm 2: Loop Selection Algorithm

Input: F - Feasible Loop Set from Loop Seeking
V - Switch Set of Graph G
Step 1: Initialize Variables;
S $\leftarrow \emptyset$, *Cover* $\leftarrow \emptyset$;
Step 2: Loop Selection;
while Cover != V **do**
  selected $\leftarrow \emptyset$;
  for *each loop* $\in$ *F* **do**
    loop.*IE*$\leftarrow \frac{\|\{v|v \in loop\} - Covering\|}{\|loop\|}$;
    if *loop.*IE $>$ *selected.*IE **then**
      selected $\leftarrow$ loop;
    else
      if *loop.*IE == *selected.*IE **then**
        selected $\leftarrow$ argMin($Delay_{loop}$,$Delay_{selected}$);
  if *selected.*IE == *0* **then**
    break;
  S $\leftarrow$ S$\bigcup${selected},
  Cover$\leftarrow$*Cover*$\bigcup${v|v $\in$ selected};
Output: S - Selected Loop Set to Cover Switches

The above formula can be simplified as a weighted set cover problem, which is NP-hard. We use an approximation algorithm of greedy heuristic to solve it and get an approx-

imate solution [26]. The complete algorithm is described in Algorithm 2. *Cover* represents the switches covered by the selected loops and *Increase Effect (IE)* represents the effect can be obtained from adding a new loop to the selected set. The time complexity of *Loop Selection* is $O(\parallel F \parallel \times min(\parallel F \parallel, \parallel V \parallel))$, where $\parallel F \parallel$ is the number of loops found in Algorithm. 1 and $\parallel V \parallel$ is number of switches in the network topology. The approximation ratio of this greedy algorithm is $O(log(n))$ and it is optimal to solve set cover problem in polynomial time.

C. Overall Running

The overall running of *TOR-ME* based on the techniques we have discussed in previous sections and detailed algorithm is described in Algorithm. 3. Without loss of generality, we assume that each switch in the network topology has flow tables specifying regular matching fields (*e.g.*, Src/Dst IP Address/port, VLAN, MPLS), actions (*e.g.*, forwarding, to_controller, meter), and supporting rule priorities.

Algorithm 3: Overall Running

```
   Input: G = (V, E), Flow Packets
1  Step 1: Elephant Flow Detection;
2  if Controller load ≥ δ then
3      if Not Elephant Flow then
4          Enter Loop Path (Step 2);
5      else
6          Wait until Flow Entries Installed;
7  Step 2: Loop Seeking (IV-A);
8  Step 3: Loop Selection (IV-B);
9  Step 4: Install Loop Paths;
10 while Not Flow Entries Installed do
11     if TTL > 0 then
12         Keep Looping in the Path;
13     else
14         Stop Looping;
15 Forwarded Directly According to Flow Entries;
```

In order to keep the flow packet from looping endlessly, we add a *TTL* field to it. Looping will stop when routing rules have not been installed on the switches and *TTL* turns to be zero. This field will be reset when the flow leaves the loop. Overall Running algorithm will also be re-executed while the network topology changes.

V. Performance Evaluation

In this section, we evaluate our *TOR-ME* framework and algorithms with network simulator. First, we introduce the metrics and benchmarks for performance comparison (Section V-A). Then we compare with the previous methods by running extensive simulations (Section V-B).

A. Performance Metrics and Benchmarks

This paper studies how to reduce controller response time (or controller load) by trading off part of data plane resources, which can avoid network congestion. We adopt five main metrics for performance evaluation. (1) the maximum controller response time; (2) the maximum load ratio of any controller; (3) the maximum load ratio of any link; (4) the network throughput; (5) the running time. We compute the maximum controller response time by Eq. 1. The second metric measures the maximum number of Packet-In messages per controller divided by the controller processing capacity during the simulation. The load ratio of a link is the traffic load divided by the link capacity. As we continuously increase the number of flows, we measure the maximum throughput that the network can support. The running time is measured when packet arrives at the ingress switch, until the packet leaves the egress switch.

To evaluate how well our proposed solution performs, we compare with other three benchmarks. The first benchmark is called the *proactive* routing scheme, in which the controller pre-installs wildcard entries for all flows. This method can achieve the lowest controller load. The second one is the *dynamic* routing scheme. When each individual flow arrives at a switch, the controller will dynamically determine its route path. The last benchmark is *SoftRing* [16], which also can reduce the control channel congestion by trading off data plane resources.

B. Simulation Evaluation

1) Simulation Settings: In the simulations, as running examples, we select two typical and practical topologies for data center networks. The first topology, called VL2[21], contains 240 switches (including 200 edge switches, 20 aggregation switches, and 20 core switches) and 1000 terminals. The second one is the fat-tree topology[23], which has been widely used in many data center networks. The fat-tree topology has total 320 switches (including 128 edge switches, 128 aggregation switches, and 64 core switches) and 1024 terminals. The parameter δ and the controller capacity are set as 0.5 and 180K, respectively. We use Packet Generator (PktGen) [27] to generate network traffic, which is a powerful tool also used by [28] [29]. Through PktGen, we can generate various sized and pattern flows, and we can collect throughput information through PktGen API. In this experiment, we generate DCTCP pattern flows and execute each simulation 100 times, then take the average of the numerical results.

2) Simulation Results: We run three groups of simulations to check the effectiveness of the proposed *TOR-ME* framework and related algorithms.

The first set of three simulations observes the performance (*e.g.*, Controller Load Ratio and Controller Response Time) of Controller. Fig. 2 shows the controller load ratio by changing the number of flows from 200K to 1M. We observe that it is almost linearly increasing with more flows arrival in the network and our framework can achieve lighter controller load than *Reactive* and *SoftRing*. For example, given 600K flows

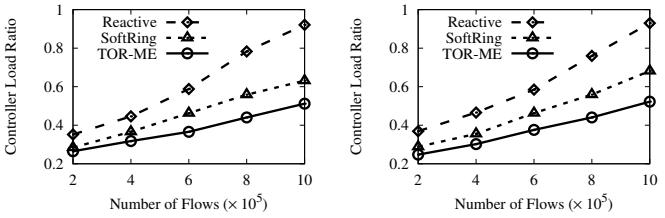

Figure 2. Controller Load Ratio vs. Number of Flows. *Left plot*: VL2; *right plot*: Fat-tree.

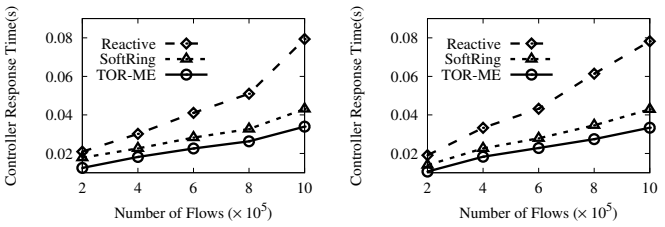

Figure 3. Controller Response Time vs. Number of Flows. *Left plot*: VL2; *right plot*: Fat-tree.

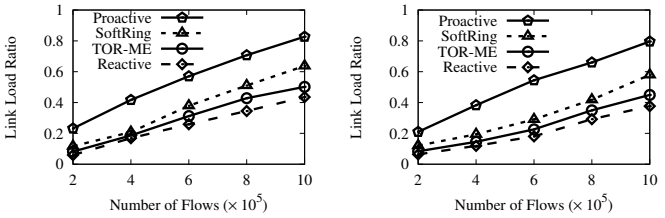

Figure 4. Link Load Ratio vs. Number of Flows. *Left plot*: VL2; *right plot*: Fat-tree.

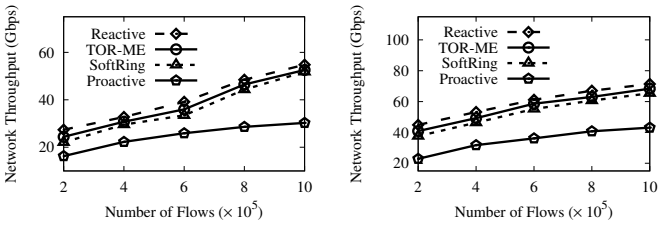

Figure 5. Network Throughput vs. Number of Flows. *Left plot*: VL2; *right plot*: Fat-tree.

in the network, the controller load ratio will reach 0.58 and 0.46 by *Reactive* and *SoftRing*, respectively. In other words, our proposed framework can reduce the controller load ratio by about 38% compared with *Reactive*. Heavier controller load also leads to longer controller response time, which is also validated by Fig. 3. This figure shows that our proposed framework can reduce the controller response time about 55% compared with *Reactive* and also significantly better than *SoftRing*.

The second set of simulations observes the routing performance (*e.g.*, Link Load Ratio or Network Throughput). Fig. 4 shows that the link load ratio is increasing with more flows in a network. Since *Proactive* can not dynamically adjust the routes for new arrivals, so its routing performance is worst

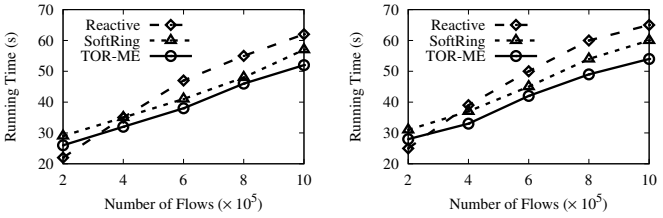

Figure 6. Running Time vs. Number of Flows. *Left plot*: VL2; *right plot*: Fat-tree.

among these solutions. Our proposed *TOR-ME* is very close to the *Reactive*, which has best routing performance. Fig. 5 shows that the network throughput will increase when there are more and more flows in the network. However, the increasing ratio is slower with more flows. *TOR-ME* can improve the network throughput by about 60% compared with *Proactive*, also optimize it of *SoftRing* and is close to *Reactive*.

The last set of simulations observes the running time of these solutions, including *SoftRing*, *TOR-ME* and *Reactive*. Fig. 6 shows that running time of *Reactive* is shorter than other two solutions at the beginning, but the advantage of *TOR-ME* becomes obvious, with more and more flows in the network. That is to say, our proposed framework can achieve ideal performance of running time.

VI. Conclusions

In this paper, we focus on the optimization of controller load and response time in SDNs. We proposed a novel *TOR-ME* framework, and present several algorithms for it. Our evaluations demonstrate that the proposed solutions can achieve much lower controller response time and running time compared to existing solutions. In the future, we will observe the impact of traffic dynamics and looping overhead.

References

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

[2] B. Pfaff *et al.*, "Openflow switch specification v1.3.0," 2012.

[3] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The google cluster architecture," *IEEE micro*, no. 2, pp. 22–28, 2003.

[4] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM computing surveys (CSUR)*, vol. 36, no. 4, pp. 335–371, 2004.

[5] T. Wang, F. Liu, J. Guo, and H. Xu, "Dynamic sdn controller assignment in data center networks: Stable matching with transfers," in *IEEE INFOCOM*, 2016.

[6] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *OSDI*, vol. 10, 2010, pp. 1–6.

[7] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. J. Jackson *et al.*, "Network virtualization in multi-tenant datacenters." in *NSDI*, vol. 14, 2014, pp. 203–216.

[8] T. Wang, F. Liu, and H. Xu, "An efficient online algorithm for dynamic sdn controller assignment in data center networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2788–2801, 2017.

[9] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 254–265.

[10] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 267–280.

[11] 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 *ACM SIGCOMM computer communication review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[12] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*. ACM, 2009, pp. 202–208.

[13] Z. Liu, D. Gao, Y. Liu, and H. Zhang, "An enhanced scheduling mechanism for elephant flows in sdn-based data center," in *Vehicular Technology Conference (VTC-Fall), 2016 IEEE 84th*. IEEE, 2016, pp. 1–5.

[14] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *INFOCOM, 2011 Proceedings IEEE*. IEEE, 2011, pp. 1629–1637.

[15] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks." in *Nsdi*, vol. 10, no. 8, 2010, pp. 89–92.

[16] C. Hu, K. Hou, H. Li, R. Wang, P. Zheng, P. Zhang, and H. Wang, "Softring: Taming the reactive model for software defined networks," in *Network Protocols (ICNP), 2017 IEEE 25th International Conference on*. IEEE, 2017, pp. 1–10.

[17] "Resource monitoring usage computation overview." https://www.juniper.net/documentation/en US/junos/topics/concept/ resource- monitoring- usage- calculation.html.

[18] J. Scudder, "Resource monitoring usage computation overview." http..//www.arin.net/meetings/minutes/ARIN XX/PDF/wednesday/SolutionSpace Scudder.pdf 2009-07-281.

[19] Y. Afek, A. Bremler-Barr, S. Landau Feibish, and L. Schiff, "Sampling and large flow detection in sdn," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 345–346.

[20] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*. ACM, 2017, pp. 164–176.

[21] J. S. Binfeng Wang, "A survey of elephant flow detection in sdn," *International Symposium on Digital Forensic and Security (ISDFS)*, pp. 1–6, 2018.

[22] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Presented as part of the USENIX Hot-ICE*, 2012.

[23] S. Skiena, "Dijkstra¡¯s algorithm," *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pp. 225–227, 1990.

[24] D. Johnson, "Finding all the elementary circuits of a directed graph." *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975.

[25] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," *SIAM Journal on Scientific Computing*, vol. 16, no. 2, pp. 452–469, 1995.

[26] L. l. Xindong zhang, "An approximation algorithm for solving weighted set cover problem." *Journal of Wenzhou University: Natural Science Edition*, vol. 6, no. 3, pp. 46–48, 2008.

[27] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ecn in multi-service multi-queue data centers." in *NSDI*, 2016, pp. 537–549.

[28] G. Chen, Y. Lu, Y. Meng, B. Li, K. Tan, D. Pei, P. Cheng, L. Luo, Y. Xiong, X. Wang *et al.*, "Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers." in *USENIX Annual Technical Conference*, 2016, pp. 29–42.

[29] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 1–14.