CheetahFlow: Towards Low Latency Software-Defined Network

Paper #683

***Abstract*—Software defined networking (SDN), which enables programmability, has the advantage of global visibility and high flexibility. However, when forwarding new flows in SDN, the interaction between the switch and the controller imposes extra latency such as round-trip time and routing path search time. Even though such latency is acceptable for elephant flows since it only takes limited ratio of total transmission time of elephant flows, it is an overkill to pay certain overheads for mice flows due to the their short transmission time. Moreover, the controller is frequently invoked by the mice flows since the number of mice flows accounts for a large portion of the total number of flows. Hence, the frequent controller invocation is mainly responsible for the controller performance degradation, and thus increasing the flow setup latency significantly. To solve this problem, we propose CheetahFlow, a novel scheme to predict frequent communication pairs via support vector machine and proactively setup wildcard rules to reduce flow setup latency. Particularly, in order to avoid congestion along a fixed path, elephant flows are detected and rerouted to the non-congestion path efficiently by applying blocking island paradigm. Extensive experiments show that CheetahFlow prominently reduces latency without any loss of flexibility of SDN.**

I. Introduction

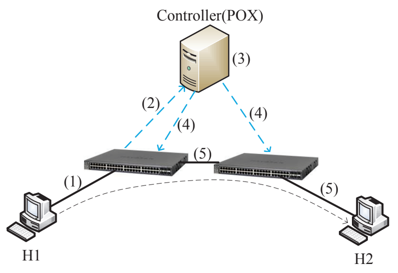
Software defined networking (SDN) has emerged as an active field in both industry and academia. The most significant feature of SDN is to provide centralized control and global view of the network. SDN enables network programmability which is infeasible in traditional network. Each switch in the network communicates with the controller through a secure channel via OpenFlow [1] protocol. Controller maintains all the switches states in the network, calculates routing path and setups rules along the path to forward packets. In contrast, routing in traditional network architecture is accomplished by all switches running distance vector or link-state algorithms. Such fixed routing path is not traffic-aware and thus not optimal. In SDN, the routing and management policy is determined by the administrator. This kind of flexibility not only eliminates many issues in traditional network, but also brings up fast innovation to develop new protocols.

However, the global view of the network yields both opportunities and challenges. Maintaining switch information and making forwarding decisions in the controller brings unprecedented flexibility, but high computation load on the controller increases the flow latency compared with traditional routing scheme. Many literatures [2]–[4] target at improving the controller performance. Upgrading controller to a more powerful server only slightly mitigates the problem. Furthermore, compared with traditional network, we have to pay extra round-trip time (RTT) and routing path search time for installing the flow entries. According to previous measurements [5], [6] in datacenter, it is stated that 80% of the flows are smaller than 10KB and last under a few milliseconds while top 10% elephant flows account for most of the traffic volume. Similar results were found in operational network as well [7]. The aforementioned traffic pattern indicates that short flows play an important role in network. However, the challenge for SDN is that a large number of small flows invoke controller too frequently because the switches always regard them as new flows. High workload in controller drags down its performance and thus increases response time. In the extreme case, one mice flow consisting of only one packet pays excessively price to consult the controller. Making the matter worse, the limited process rate of the controller further increases the forwarding latency.

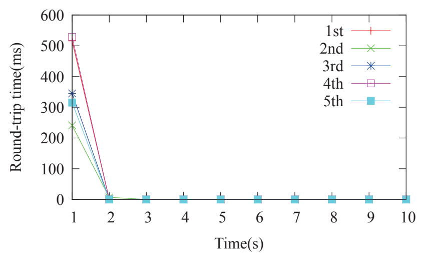
The state-of-the-art approaches only focus on improving the performance of the controller or taking advantage of high flexibility of SDN. Little work aimed at decreasing the extra latency in SDN. To address this issue, we present CheetahFlow, a novel approach to install flow entries proactively by predicting the future communication patterns according to historical data. In terms of OpenFlow specification, the controller can modify flow entries on the switches both reactively and proactively. Most existing approaches utilize the former method to respond to OFPT_PACKET_IN message which cannot meet the needs for high-throughput, low-latency network. On the contrary, we leverage proactive flow setup function to develop a more intelligent hybrid controller, which detects frequent communication pairs via support vector machine (SVM) [8]. The forwarding rules are installed along the shortest paths between frequent pairs in advance. Blocking island paradigm [9] is also applied to CheetahFlow to make traffic adaptive routing for elephant flows. The advantage of this scheme is that most of the flows are forwarded in data plane without asking the controller to make decisions in real time. Directly forwarding packets in the data plane completely eliminates the extra RTT and decision making time.

The main contributions of this paper are as follows:

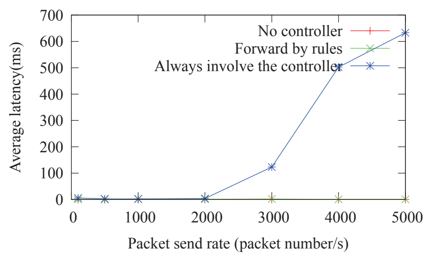
- We analyze and demonstrate the increased latency and performance loss, which are caused by the phenomenon of a large number of mice flows invoking the controller.
- We propose CheetahFlow, a novel hybrid controller which leverages SVM and blocking island paradigm to detect frequent communication pairs and non-congestion paths respectively to pre-install flow forwarding rules.
- To the best of our knowledge, CheetahFlow is the first practical scheme that reduces extra latency imposed by



(a) Simple two hosts topology



(b) RTT for two hosts. We measure the RTT between two hosts for 10 seconds. The measurement is repeated for 5 times.



(c) Average latency for different flow arrive rates. Each point is the average latency of all arrived packets.

Fig. 1. Motivation examples

interaction between the control plane and data plane without loss of flexibility.

- We conduct a wide spectrum of experiments. The experimental results show that CheetahFlow significantly reduces the extra latency between frequent communication pairs.

The rest of this paper is organized as follows. Section II illustrates the motivation of this paper by some simulation experiments. Section III presents how we mining the traffic pattern and install the flow entries. Section IV discusses the implementation of the system and experiment results. Finally, Section V elaborates related work and Section VI concludes the paper.

II. Motivation

In this section, we analyze the overheads induced by the phenomenon of a large amount of mice flows enquiring the controller in SDN and show some motivation examples.

In real world SDN deployments, the extra forwarding latency compared to traditional network can be obtained by:

$$L = t_{\text{first packet}} + t_{\text{routing path search}} + \max\{t_{\text{distribute}}\} \qquad (1)$$

Latency calculation Formula (1) could be demonstrated by Figure 1(a) which consists of 5 phases. The first term $t_{\text{first packet}}$ refers to phase 2 denoting the transmission delay from the edge switch to the controller, when a new flow arrives and matches no rule in the edge switch. The second term $t_{\text{routing path search}}$ refers to phase 3 which is the path searching time in the controller. Such latency does not exist in traditional network as the routing path is obtained by link-state or distance vector algorithm ahead of time. Path searching time accounts for a large ratio of the total latency in general. The computing complexity of the shortest-path algorithm is $O((|V|+|E|)log|V|)$. As a result, the path searching time increases a lot for a large network. The third term $\max\{t_{\text{distribute}}\}$ refers to phase 4 representing the longest delay when the controller distributes the flow modification message OFPT_FLOW_MOD to all the switches along the path. The forwarding latencies in phases 1 and 5 are not included in this formula as it exists in both SDN and traditional network environments.

Previous experiments [10], [11] show that an OpenFlow switch could only setup about 275 flows per second in practice, but such setup speed is far from enough for contemporary network. In order to find out the latency caused by mice flows consulting the controller, we build a simple topology in which two hosts is connected to a single switch. We compare the TCP throughput by iperf in two scenarios: invoking controller once and setup the rules for this flow; invoking controller for each packet without installing any forwarding rule to the switch. The throughputs for these two cases are 2.01Gbps and 14.0Mbps respectively. Apparently, there is a large gap between them. Invoking controller every time has only 0.68% throughput of directly forwarding in data plane.

Figure 1(b) illustrates the ping latency of two hosts in the topology of Figure 1(a). Apparently, the first ICMP packet has a higher latency than the following packets. The reason is that the first packet requires the switch to communicate with the controller and setup the flow entries back to the switches along the routing path. This process imposes extra latency to forward packet to the destination. Our result matches previous measurements in [10].

To further explore the problem, we generate flows to find out the relation between the flow arrive rates and latency. The simulation experiments are conducted in Mininet2.0 [12] by sending packets from one host to another via UDP in different scenarios. The latency is measured on the server side once it receives the packet. "No controller" means traffic is sent and received on the local host which is the baseline of UDP transmission latency for comparison. "Forward by rules" means the packets are forwarded by the rules on the switches. "Always involve the controller" means the controller is configured to process every packet to emulate high flow arrive rate network, namely each packet is treated as a *new flow* to be processed by the controller. The result is shown in Figure 1(c). For high flow arrive rates, the latency increases from ~5ms to ~600ms. The latency by forwarding rules is almost equal to that of forwarding inside the same host.

In summary, from the experiments above, we learn that the frequent controller invocation lowers the performance and increases the latency in SDN. Installing forwarding rules to the switches in advance is a promising approach to reduce the latency.

III. System Design

A. Overview

To address the latency and performance issues in SDN, we propose a novel scheme named CheetahFlow that leverages a data mining algorithm to setup rules on switches in advance. The design principle is that the approach must be compatible with current OpenFlow specification which needs no extra

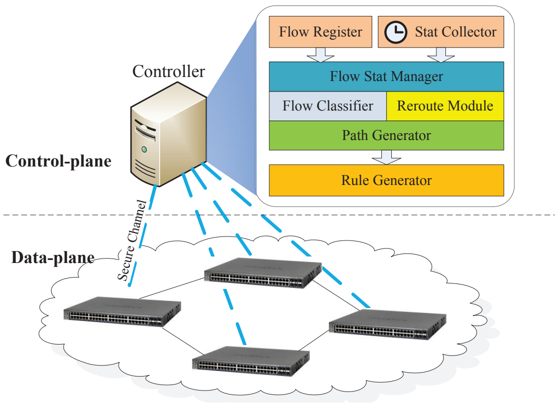


Fig. 2. CheetahFlow architecture

modification to switch hardware, OpenFlow protocol and end hosts. The above principle guarantees our solution is practical with little extra effort to upgrade. However, there are three challenges to resolve: first, how to find the most frequent communication pairs to setup rules in advance? Second, how to prevent mice flows from congesting along a pre-installed path? Third, how to tackle the flow table overflow problem due to the limited size of OpenFlow switch flow table?

The underlying logic of our solution is as follows: finding out the hotspot communication pairs and pre-installing the forwarding rules. When elephant flow appears, reroute these flows to a non-congestion path. To address the challenges, we propose a three-layer architecture illustrated in Figure 2. The top layer is the module for inputing flow information. There are two ways to collect flow information: push based and pull based. The controller learns host communication pattern by flow registration which is a pull based method. The controller gathers flow statistics to flow stat manager which is a push based method. The second layer, named flow stat manager, is the core component of CheetahFlow. It consists of three parts: flow classifier, reroute module and path generator. This layer collects flow information and makes further decisions to install forwarding rule to the switches. Reroute module is triggered by flow classifier notification and it migrates elephant flow to a high bandwidth path. The bottom layer is rule generator which makes trade-off between flows match missing and flow table size.

The workflow of CheetahFlow is as follows:

1) In the beginning, when a new flow comes, CheetahFlow updates the flow information in the flow stat manager and installs the exact match rules reactively.
2) Flow statistics is collected from the switches periodically and updated in the flow stat manager.
3) Flow classifier checks if new frequent communication pairs are detected. If so, path generator searches shortest path for these pairs and passes them to rule generator.
4) Flow classifier checks if new elephant flows are detected. If so, reroute module is triggered and path generator searches path by heuristic algorithm and passes it to the rule generator.
5) Rule generator produces wildcard rules for frequent communication pairs and exact match rules for elephant flows.

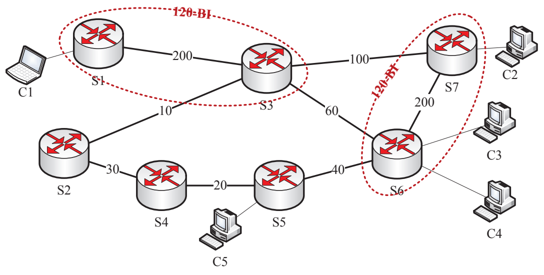


Fig. 3. Blocking island graph

From the above workflow description, it is easy to note that CheetahFlow works as a reactive controller initially. After the convergence time is reached, i.e. enough flow statistics is gathered from the switches, it works as a hybrid controller that pre-installs forwarding rules to the switches. Therefore, CheetahFlow keeps the advantage of flexibility of SDN.

B. Frequent Communication Pair & Elephant Flow Detection

As we want to setup flow entries according to network traffic communication patterns, it is necessary to track communication information to obtain the frequent communication pairs. Frequent pair information is maintained in flow stat manager and stored in a hash table, which maps src, dst pair to an information structure. The hash table structure is listed as below:

$$\{src, dst\} \mapsto \{\text{flownum, transmitted bytes, aging time}\}$$

Upon receiving a packet which doesn't match any rule, related information is updated in the structure. Flow classifier uses SVM to classify frequent communication pairs. We use flow number between src and dst, the transmitted bytes and flow aging time (past time since last matched packet) as SVM features to train and classify frequent pairs. Experiments demonstrate that these parameters are effective to pick frequent communication pairs. Detailed evaluation results are listed in Section IV.

Flow stat manager detects elephant flow by sampling and sliding window on the basis of its heavy tailed distribution and short interval time [13]. However, collecting statistics from switches is costly due to the limited bandwidth between the control plane and data plane. CheetahFlow further reduces the cost by the following heuristic: flow stat manager only gathers statistics from edge switches instead of all the switches. It is reasonable to use this heuristic because flow statistics is the same for a specific flow among all the switches along its routing path.

C. Routing Heuristics

In order to reduce latency in SDN, customized routing scheme for different type of flows is necessary. Mixed routing path search algorithms are used in CheetahFlow.

1) Blocking Island Paradigm: The key idea of blocking island (BI) [9] is transforming the original network graph into a hierarchy tree which contains available bandwidth information. The benefits of applying blocking island for path searching is obvious: it reduces the routing path search space significantly with bandwidth guarantee.

Algorithm 1 Construct β-BI

```
1: function CONSTRUCTBI($N = \{V, E\}, \beta$)
2:     $L \leftarrow \{\emptyset\}$                                    ▷ $L$: Result $\beta$-BI list
3:     for all $v$ in $V$ do
4:         if not visited($v$) then
5:             $I \leftarrow ConstructBIFromNode(N, \beta, v)$
6:             $L \leftarrow L.Add(I)$
7:         end if
8:     end for
9:     return $L$
10: end function

11: function CONSTRUCTBIFROMNODE($N = \{V, E\}, \beta, x$)
12:     $I \leftarrow \{x\}$                                           ▷ $I$: Result $\beta$-BI
13:     $S \leftarrow \{$links incident to $x$ and weight $\geq \beta\}$          ▷ $S$: stack
14:     while $S \neq \emptyset$ do
15:         $l \leftarrow pop(S)$
16:         $e \leftarrow$ another endpoint of $l$
17:         if $e \notin I$ and $weight(l) \geq \beta$ then
18:             $I \leftarrow I \cup \{e\}$
19:             $S \leftarrow S \cup \{$links incident to $e$ and weight $\geq \beta\}$
20:         end if
21:     end while
22:     return $I$
23: end function
```

For a given network N, it can be denoted as a weighted graph $N = (V, E)$, where V denotes nodes (switches) in the graph and E denotes link set. The weight of the link in E is the available bandwidth. A bandwidth request d from source src to destination dst with bandwidth B can be denoted as 3-tuple $d = (src, dst, B)$. A β-blocking island (β-BI) for a node x is a set of nodes (including x) which can be reached from x with weight of at least β, all the links between these nodes also belong to this β-BI. A β blocking island graph (β-BIG) is a graph which is clustered into groups. Each group is a β-BI. For example, in Figure 3, switch nodes in original graph are divided into two groups by bandwidth requirement 120: $\{S1, S3\}$ and $\{S6, S7\}$ [1].

Blocking island has many interesting properties. The main properties that are useful for bandwidth-aware routing are listed below.

- Uniqueness. For $\forall\beta$, there is one and only one β-BI for a certain node.
- Routing existence. Given a request $d = (src, dst, B)$, if src and dst belong to the same β-BI, there are at least one routing path exist in this BI. Conversely, if there is a path from src to dst, all the links along this path belong to this BI.
- Inclusion. If $\beta_i < \beta_j$, β_j-BI is a subset of β_i-BI.

To explain how blocking island works, we show some examples in Figure 3. If we want to find a path with 120 bandwidth from $C1$ to $C4$, we can get the answer "route not exist" immediately since $C1$ and $C4$ are not in the same 120-BI. If we want to find a path with 120 bandwidth from $C2$ to $C4$, the path search space is reduced from the whole graph to only $\{S6, S7\}$, which speeds up response time a lot. Based on inclusion property, a h-level blocking island hierarchy can be constructed via a series of β_i-BI which $\beta_i < \beta_{i-1}, i = 0, 1, \ldots, h$.

According to the definition, β-BI can be constructed by greedy Algorithm 1. It is easy to get the construction BI

[1]According to the definition, single-node sets $\{S2\}$, $\{S4\}$ and $\{S5\}$ are β-BI as well. For the sake of brevity, single-node groups are not circled.

Algorithm 2 Find Path for Elephant Flow

```
1: function FINDELPTFLOWPATH($src, dst, I$)
2:     $i \leftarrow$ Layer number
3:     while $i \geq 0$ do
4:         if $src \in I[i]$ and $dst \in I[i]$ then break
5:     end while
6:     $path \leftarrow SearchPath(src, dst, I[i])$
7:     return $path$
8: end function
```

complexity is $O(|E|)$. When a new path is allocated, the updating of BIG is needed. The complexity of updating BI is also $O(|E|)$. The complexity of querying two nodes in the same BI is $O(1)$ as only two hash operations are required.

2) Routing Path Search: Shortest-path algorithm is the default routing path search scheme for frequent communication pairs. However, after installing the forwarding rules to switches on the path, every flow will be routed through the same path. It is not fair for mice flows. If elephant flows and mice flows go through the same path in a short period of time, incast problem [14] may occur and lead to long latency. To avoid this problem, when the elephant flow is found by flow stat manager, reroute module will be triggered. The module places the elephant flow to another "non-congested" path.

Blocking island paradigm is suitable for finding "non-congested" path in the network. Blocking island graph is updated in the flow stat manager after a path allocated to a specific flow. For efficiency consideration, CheetahFlow constructs a three-layer blocking island hierarchy H consisting of three blocking island graphs $I[1]$, $I[2]$ and $I[3]$. $I[0]$ is a 0-BI, i.e. the whole graph G. Given the max bandwidth of the network B, $I[1]$, $I[2]$ and $I[3]$ are $\frac{1}{3}B$-BI, $\frac{2}{3}B$-BI, B-BI respectively. The heuristic for routing elephant flows is to find the maximum available bandwidth path from src to dst. It is more effective if the two paths are disjoint. With blocking island hierarchy constructed before, this can be done by Algorithm 2. If no disjoint path can be found or there exists only one path, the shortest path will be returned.

D. Wildcard Rule

CheetahFlow cares only source and destination of a flow. However, for a network having n hosts, the total flow entries number $E = H * F * \frac{n(n-1)}{2}$, where H is the average hops between source and destination, and F is the average flow numbers of a pair of hosts. E is proportional to n^2 which is a large number even in a middle-sized network. Pre-installing all these entries to switches is impractical due to the limited flow table size. The flow table size of OpenFlow switch is about 1K to 4K [10]. As a result, wildcard rule is the requisite for aggregating many mice flow forwarding rules. Once a frequent communication pair is detected, a wildcard rule matching only src and dst will be installed along the shortest path. Applying wildcard rules not only eliminates the latency induced by consulting controller, but also compresses the rule space significantly. Here is an example of exact match rule and wildcard rule between two hosts:

```
idle_timeout=10,hard_timeout=30,priority=65535,tcp,in_port=3,
vlan_tci=0x0000,dl_src=3a:07:ce:24:87:3c,dl_dst=ea:0d:62:a4:
e3:c3,nw_src=10.0.0.2,nw_dst=10.0.0.1,nw_tos=0,tp_src=56467,
tp_dst=7000 actions=output:1

idle_timeout=100,priority=50000,ip,nw_src=10.0.0.1,nw_dst=10.
```

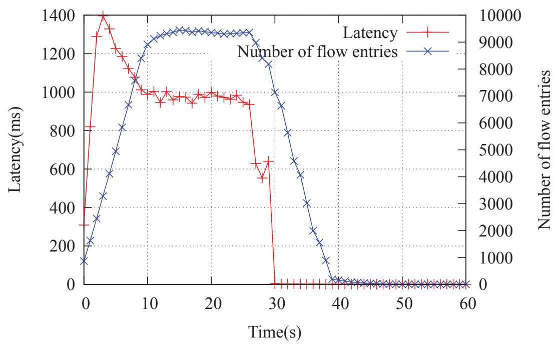


Fig. 4. Latency trends before/after wildcard rule setup

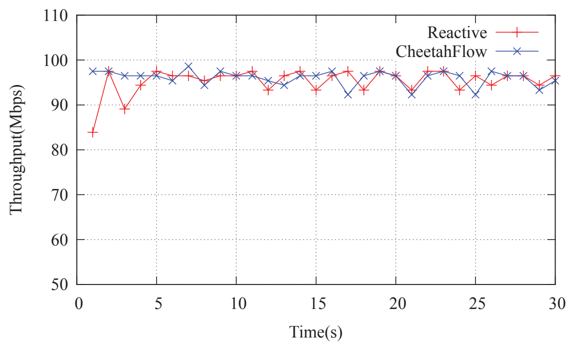


Fig. 5. Throughput for a frequent communication pair

```
0.0.3 actions=output:2
```

It is easy to see that the wildcard rule has no hard_timeout and the idle_timeout is much longer than that of the original exact match rule. We set longer idle_timeout because these wildcard rules are for frequent communication pairs. The priority of wildcard rules is higher than the default exact match rules. Therefore, at most by idle_timeout, the old exact rules will be kicked out from the switches and flow table size will shrink noticeably. Flow stat manager takes responsibility to remove the wildcard rules proactively to free the flow table space if the frequent communication pairs are not qualified.

IV. Evaluation

In this section, we introduce the prototype implementation based on an emulated hierarchy topology. Then we evaluate the performance of CheetahFlow from different aspects.

A. Implementation

CheetahFlow is implemented on top of POX controller [15] and Mininet2.0 [12] in Python. We build a testbed with $k = 4$ fat-tree topology based on RipL-POX [16]. Blocking island module and path searching algorithm are written in mixed C++ and Python. We use LIBSVM [8] which is a widely used open-source implementation of SVM to train and classify frequent communication pairs. Experiments are conducted on a computer with Intel i5-650 3.20 GHz (4 cores) processor and 4G RAM.

B. Experimental Results

1) Flow Classifier: We first conduct experiments to find out the accuracy of flow classifier which is crucial for mining frequent communication pairs. SVM is trained by real communication data of two hosts. We use radial basis function (RBF) kernel with cross-validation training sets to obtain our final classifier. Experiments illustrate that the flow classifier achieves accuracy up to 76.9%. The false positive rate and false negative rate is about 97% and 40% respectively. This means that most of the frequent communication pairs are detected and wildcard forwarding rules are installed ahead of time. Some non-frequent communication pairs also benefit from pre-installed rules. These false positive judgments lead to a waste of flow table space. However, compared with the saved space of wildcard rules, the side effect is negligible.

2) Latency in CheetahFlow: We simulate empirical data mining workload which is obtained from production datacenters [6] on the testbed. We measure the latency and the number of flow entries per second between one frequent communication pair which has five hops distance from each other. From Figure 4, it is easy to note that before the frequent communication pair is detected(around $0 \sim 25s$), the latency between this pair is about $1000ms$. After the wildcard forwarding rules are installed along the path, the latency drops below $10ms$. The trend for flow entry number on the edge switch is also interesting. Before $25s$, the flow entry number increases monotonically as many new flows arrive. After the rule is installed proactively with a higher priority, the forwarding packets begin to match the new rule instead of the old one. Therefore, the forwarding rule number decreases smoothly as time passed.

3) Throughput: Throughput test is conducted during one frequent communication pair. According to Figure 5, there is not much difference between reactive approach and Cheetah-Flow. Only at the beginning of the communication, the reactive method has a slightly lower throughput loss due to flow setup latency.

4) Blocking Island Efficiency and Overheads: Blocking island reduces the path searching time very much especially in a large network. Due to the limitation of simulation network size, we conduct extensive experiments on a large network to demonstrate its efficiency. Table I compares direct search path time and BI search path time for four requests in an Erdős-Rényi graph with 1000 nodes and 75124 edges. For the first three cases, BI is efficient for searching path with bandwidth guarantee since it shrinks the search space significantly. But for the last case, the src and dst belong to the same BI which is the whole graph in fact. Consequently, it is reasonable that BI doesn't have remarkable improvement in this scenario. In general, path searching with BI is much efficient than direct search in most cases.

The construction overheads of BI are shown in Figure 6, we build two graphs with up to 5000 nodes by Erdős-Rényi graph model and Waxman graph model. The BI construction time is in proportion to graph edge numbers as expected. Even in a network with 5000 nodes, the three-layer BI construction time is within $4s$ which is less than flow statistic gathering time.

V. Related Work

SDN separates the control plane and data plane to achieve high flexibility. In this emerging research field, many research

TABLE I
BI SEARCH PATH TIME VS. DIRECT SEARCH PATH TIME

Request	1	2	3	4
Direct search time(ms)	0.3	10.0	33.1	52.0
BI search time(ms)	0.1	0.1	12.6	49.8
BI search space(nodes/edges)[a]	5/4	71/415	592/26493	1000/75124

[a]Direct search space(node/edges) for all the requests are always 1000/75124.

papers focus on how to improve the performance of SDN and find a better trade-off between flexibility and performance.

As mentioned before, the controller is the major bottleneck of the system. Therefore, many controllers such as NOX [17], NOX-MT [2], Maestro [18], Floodlight [19] and Beacon [20] with different specialties have been proposed. Syed Abdullah et al. [21] compares all these controllers performance from different aspects and provides guidelines to design new controllers with better scalability. Besides, some distributed controllers [3], [4] are presented, which try to increase the computing power of centralized control. Although these controllers have short response time and good throughput, the extra RTT and path searching time are not taken into consideration.

DevoFlow [10] is the most related work to this paper. The basic idea of DevoFlow is devolving control over most of flows back to the switches. Rule cloning, local actions and sampling method are applied to SDN. Such operations reduce interactions between switch and controller and number of TCAM entries. However, the scheme proposed in DevoFlow is not completely compatible with current switch design so that it requires hardware modifications. DIFANE [22] generates and distributes rules to many authority switches, which enables handling traffic in fast datapath. It compresses rule numbers by supporting wildcard rules as well. But DIFANE needs changes to switches which are of high price. Kandoo [23] proposed a hierarchical controller architecture which consists of one root controller and many local controllers to limit the overhead of frequent events on control plane. Low level controller makes local decisions such as elephant flow detection while root controller reroutes the flow by global network view. [24] presents the controller placement problem, which try to minimize the propagation latency by finding an optimal placement of the controller in wide area network. Additionally, many data mining techniques such as SVM [8], frequent pattern mining [25], [26] have been widely used in networking problems.

VI. Conclusion

In this paper, we present a novel system called CheetahFlow to reduce the extra latency in SDN while preserving its flexibility. We investigate the impact of mice flows bringing to the controller and examine the extra latency in SDN. We propose a novel approach to install flow entries to the switches proactively on the basis of historical communication preferences. We leverage SVM to identify frequent communication pairs with high accuracy. Heuristic routing algorithm and blocking island paradigm are applied to make traffic-aware routing efficiently. Extensive experimental results demonstrate that our scheme significantly reduces forwarding latency in SDN. CheetahFlow paves the way for applying SDN to latency sensitive systems in both data center and operational network.

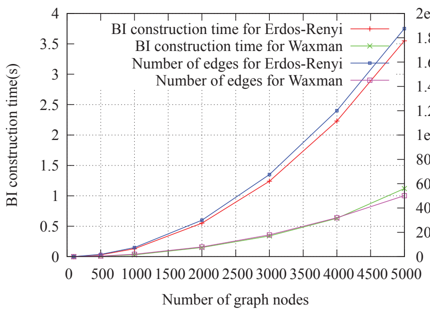


Fig. 6. BI construction time

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," 2008.

[2] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *HotICE*, 2012.

[3] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: a distributed control platform for large-scale production networks," in *OSDI*, 2010.

[4] A. Tootoonchian and Y. Ganjali, "HyperFlow: a distributed control plane for openflow," in *INM/WREN*, 2010.

[5] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC*, 2010.

[6] 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 *SIGCOMM*, 2009.

[7] B.-Y. Choi, J. Park, and Z.-L. Zhang, "Adaptive packet sampling for accurate and scalable flow measurement," in *GLOBECOM*, 2004.

[8] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," 2011.

[9] C. Frei and B. Faltings, "Abstraction and constraint satisfaction techniques for planning bandwidth allocation," in *INFOCOM*, 2000.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *SIGCOMM*, 2011.

[11] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *OSDI*, 2010.

[12] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *CoNEXT*, 2012.

[13] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, "Identifying elephant flows through periodically sampled packets," in *SIGCOMM*, 2004.

[14] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *SIGCOMM*, 2009.

[15] "POX controller," http://www.noxrepo.org/pox/about-pox/.

[16] "Ripcord-Lite for POX: A simple network controller for openflow-based data centers," https://github.com/brandonheller/riplpox.

[17] "NOX controller," http://www.noxrepo.org/nox/about-nox/.

[18] C. Zheng, A. L. Cox, and T. S. E. Ng, "Maestro: Balancing fairness, latency and throughput in the openflow control plane," Rice University, Tech. Rep., 2010.

[19] "Floodlight openflow controller," http://www.projectfloodlight.org/floodlight/.

[20] "Beacon," https://openflow.stanford.edu/display/Beacon/Home.

[21] S. Syed Abdullah, F. Jannet, F. Maham, S. Aamir, and M. Syed Akbar, "An architectural evaluation of SDN controllers," in *ICC*, 2013.

[22] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *SIGCOMM*, 2010.

[23] S. H. Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *HotSDN*, 2012.

[24] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *HotSDN*, 2012.

[25] T. Yongxin, C. Lei, C. Yurong, and P. S. Yu, "Mining Frequent Itemsets over Uncertain Databases," 2012.

[26] T. Yongxin, C. Lei, and D. Bolin, "Discovering Threshold-based Frequent Closed Itemsets over Probabilistic Data," in *ICDE*, 2012.