SoftRing: Taming the Reactive Model for Software Defined Networks

Chengchen Hu, Kaiyu Hou, Hao Li, Ruilong Wang, Peng Zheng, Peng Zhang, Huanzhao Wang
MOE KLINNS Lab
Department of Computer Science and Technology
Xi'an Jiaotong University

***Abstract*—The reactive model of Software Defined Networking (SDN) invokes controller to dynamically determine the behaviors of a new flow without any pre-knowledge in the data plane. However, the reactive events raised by such flexible model meanwhile consume lots of the bottleneck resources of the fast memory in switch and bandwidth between controller and switches. To address this problem, we propose SoftRing with the motivation to mitigate the overhead to handle a reactive event. In fact, the reactive packets are not necessarily stored in the switch or sent to the controller; instead, they are forwarded to traverse a pre-defined loop path. The packets will finally leave the loop path after the switch rules related to the packet flow being updated to switches in the loop with fewer flow entries. We have implemented a SoftRing system that integrates the controller and software/hardware SDN switches. The results show that SoftRing can eliminate the fast memory requirement for reactive packets and reduce the control channel bandwidth consumption up to 80%, with the cost of less than 5% data plane bandwidth, an average of three extra flow entries in each switch, and minor extra latency for the flow forwarding.**

I. Introduction

In the context of Software-Defined Networking (SDN), the controller independently computes the "knowledge" on processing the packets and issues it to the switches in a *Match-Action* abstraction, *aka*, flow rules/entries. To push the flow rules into switches, a *proactive mode* attempts to anticipate the network issues in advance, while a *reactive model* will invoke the controller on detecting a network event.

One common trigger of the reactive model is the so-called "table-miss" event: an incoming packet does not match any forwarding rule in switch's flow table. And it can also actively program the rule action to invoke the controller. The reactive model could be leveraged to flexibly build up the SDN applications, however, its reliance on the exchanging messages between controller and switches would easily fulfill the control/data channel with complex applications or large networks, and even be abused to launch a Denial of Service (DoS) attacks [29]. The objective of this paper is to ensure the reactive model at an affordable cost.

Achieving this goal is not trivial. First, though the proactive-only model completely eliminates the reactive overhead, it cannot serve all applications that require flexible flow management (Section II). Second, simple utilization of existing memory (DRAM, SRAM) in the switches is far from sufficient to handle many simultaneous reactive events, which would further congest the control/data channel (Section III). However, while the bandwidth between two planes is in great demand, the bandwidth in data plane is quite underutilized: the average link utilization in Internet is lower than 50% and the utilization in date center seldom exceeds 25% [7]. Based on this observation, we propose *SoftRing* resorting to the unused data plane bandwidth to handle overhead.

Let us draw an analogy to explain the idea. A popular restaurant also has a "table-miss" problem, which might not have enough tables to serve every customer on their arrival during the rush hours. In this case, a waiter would arrange a waiting list recording the customers' names. It is fine for the waiting customers to walk away and come back later checking the seat availability. Also, the customers might go to another nearby restaurant that has available tables.

This inspires us that the packets triggering the reactive events (reactive packets) can also "walk around": an SDN switch forwards the reactive packets to a pre-computed "waiting-loop" path in the data plane instead of having it in the arriving switch's buffer or sending to the controller. A corresponding rule would be later installed to one switch in the loop, which makes the packet leave the loop and be properly forwarded/operated.

The design of SoftRing is presented in Section IV followed by three sections addressing the main issues related to SoftRing: a two-stage mechanism generating the waiting-loop paths in Section V, switch enforcement method to enable the reactive packets forwarding along the waiting-loop path in Section VI and the strategies to handle the network dynamics in Section VII. We implement the SoftRing prototype and evaluate it in Section VIII. Finally, in Section IX we conclude the paper.

II. Why Not Proactive Model Only

It is no doubt that the proactive model largely cuts the overhead between controller and switches but the reactive model in many scenarios is worth preserving due to the following limitations with only proactive model.

Limited forwarding memory in switch. An SDN switch can provide approximately thousands of flow entries, which

This work is supported by the National Key Research and Development Program of China (2017YFB0801703), the NSFC (No.61672425, 61702407, 61772412, 61402357), the Microsoft Research Asia Collaborative Research Program (2016JM6066) and State Grid Corporation of China (DZ71-16-030).

978-1-5090-6501-1/17/$31.00 ©2017 IEEE

may be enough to install proactively for only possible flow forwarding. However, such critical constraints on the flow tables stifle the innovation of SDN, obstructing the deployment of applications that rely on more flexible or finer-grained flow management. For example, an ISP can prioritize the traffic of World-of-Warcraft (WoW) for the subscribed players to provide better game experience. An SDN solution for this service requires massive flow rules considering the large population of WoW players in order to detect the WoW traffic from the subscribers and then redirect to a prioritized link or network. In literature, there are two broad ways to fill the large rule set into small flow tables: aggregating the rules, or predicting the traffic. The first solution works well for routing prefix aggregation, while it becomes challenging when tens of fields are involved in SDN/OpenFlow context [28], or the field values are scattered randomly, *e.g.*, it is impractical to aggregate the randomly subscribed WoW players' IP addresses. The second method combines the proactive model with a predictive approach, leveraging a time division multiplexing mechanism on flow tables. However, the network is hard to predict timely, and the reliable prediction is only with simple coarse-grained scenarios, *e.g.*, it is impossible to perfectly predict when the subscribers will be online and issue the flow rules beforehand.

Things get easy if the reactive model is enabled. Still consider the above WoW service, the switch is proactively installed with only one rule which notifies the controller once detecting the newly incoming WoW traffic. If the traffic is from a service subscriber, a further rule indicating the corresponding forwarding path with higher priority will be used to configure the switch. Since it is not likely to have all players online simultaneously, the reactive model that only installs rules for online players, while providing the same service level. For the un-subscribers, it is easy to use a bloom filter to aggregate them into a single forwarding rule.

Unawareness of data plane. The proactive-only model assumes that the data plane correctly executes all the policies from the control plane, which is not the case in the real network, raising potential reliability issues [28], [19]. For example, the flow rules can be mistakenly removed due to the bugs or attacks on both controller and switches, resulting in the wrong forwarding behaviors in the data plane, *e.g.*, black hole, loop, *etc*. In these cases, the switches will continuously drop packets. Because the controller cannot be aware of such failures with only proactive model. In other words, though proactive-only model eliminates the single-point failure on the controller, it also loses control over the failure on the switches. To this end, we argue that the reactive model triggering a packet-in message provides awareness of data plane, bringing better reliability of the system.

Generally, we believe the reactive model should coexist with proactive model as a complement, as our objective mentioned in Section I. The proactive-only model is similar to "refusing to eat for fear of choking". And in this paper, we explore the possibilities to "keep eating while reducing the risk of choking".

III. Reactive Model Overheads

A. When Does Reactive Event Happen?

In the real network, table-miss is the most common cause triggering a reactive event. The main causes are two-fold: (1) the flow table size is small, so only a small portion of rules could be accommodated [16], and (2) the small flows that contributes only 20% traffic occupies 80% portion of the total, which means the switch needs to handle more new flows in unit time [7].

Even worse, the frequency of table-miss events will increase if the flow table is full, since a replacement method, *e.g.*, the Least Recently Used (LRU) policy, will potentially raise the flow table thrashing, bringing more table-miss events. For this reason, when we say a "new flow" packet triggering a table-miss event, it does not only mean the first packet of the flow, *e.g.*, TCP SYN packet, but also possibly a subsequent packet of the flow missing a match on the flow table due to the limitation of table capacity. In addition, multiple connections in HTTP/1.1 also bring several table-miss events before the establishment of the connections.

Besides the table-miss event that only reacts to the missing flow entries, operators may actively enable the reactive model to collect data plane status [12], perform high-level processing [6], and adopt fine-grained flow control as the WoW case, *etc*., which has analogy with a CPU interrupt in PC system.

B. Overhead in the Design Space

The reliance on the controller slows down the processing in the reactive model. The mismatch of the processing speed requires to move the reactive packets out of the processing pipeline in switch and buffer them somewhere till the controller replies.

A hardware SDN switch usually consists of a *Control Engine (CE)* and a *Forwarding Engine (FE)*. The former coordinates with the controller and the latter provides line-rate processing (parsing, matching, metering, *etc*.) on each packet by the hardware configurations. Based on this architecture, we have several design options with off-the-shelf controller and switches: controller armed with disks and DRAMs has the largest storage space, CE is usually equipped with a DRAM and the FE has a fast but small SRAM.

Keep in switch's FE. To keep pace with the processing speed, SRAM as the memory for FE is quite small. As shown in Table I, the buffer size of a typical commercial OpenFlow switch (OFS) is 4MB-16MB. Let us approximately calculate the packet rate (R_o) of the new flows filling the buffer by $R_o = \frac{M}{TP}$, where M is the buffer size, P is the switch throughput and T is the time to activate a new rule in switch after the table-miss event. A recent work showed in [24] that the time to activate a new rule in switch could be with a median value of 12ms and an upper bound (90% confidence probability) of 15ms. Let us choose 15 ms for T and the specification of Pica8 switch for M and P, it overflows the buffer if the new flow rate exceeds 0.26% of the switch capacity.

TABLE I
SPECIFICATION OF COMMERCIAL OFS

Brand	Model	Port	Buffer	Control BW
Pica8	AS7712-32X	100GbE*32	16MB	1000Mbps
Brocade	ICX7750-26Q	40GbE*26	12.2MB	1000Mbps
Dell	Z9100-ON	100GbE*32	16MB	1000Mbps
Huawei	CE8860	100GbE*32	16MB	N/A
Netgear	M5300-52G	48+10GbE*4	4MB	10Gbps

Buffer in switch's CE. Besides a dedicated FE silicon, *i.e.*, switching chip or network processor (NP) or Fields Programmable Gate Array (FPGA), a general-purpose CPU is adopted as the CE in a switch. The CE acts as a bridge between the controller and FE, which encapsulates/decapsulates the messages to/from controller. In general, a DRAM is used as the CPU's memory. The size of DRAM could be as large as several gigabytes. However, the bandwidth of the bus connecting FE and CE in a switch becomes the bottleneck, which is usually only 1Gbps (For reference, HP ProCurve 5406zl has only 80Mbps bandwidth [11]).

Send to controller. The controller has enough space to store the reactive packets and these packets can be sent though FE directly to bypass the bottleneck bus between FE and CE. However, all the switches share the single channel connecting to the controller. The test on HP ProCurve 5406zl shows that only 17Mb/s bandwidth is available between controller and switch in [11]. Let us assume a network with 20 switches, each of which is with 3.2Tbps capacity. Even with a controller providing a 40Gbps or 4 x10Gbps bandwidth and processing capacity (almost today's limit), only in average 0.06% of the traffic being sent to the control channel would consume all the control bandwidth.

Combination design: the OpenFlow's choice. The de facto SDN South-Bound Interface (SBI), *i.e.*, OpenFlow, utilizes the combination of the fast memory in FE and the bandwidth of control/data channel. To be specific, once a reactive event occurs, the OFS buffers the reactive packet in FE's memory like the "keep in switch's FE" solution, and sends a packet-in message to the controller, carrying only the first 128Bytes of that packet, which would be sufficient for controller to determine the corresponding flow rules in most cases. If the buffer in FE's memory overflows, OpenFlow degrades to the "send packets to controller" solution, embedding the entire packet into the packet-in message to avoid packet loss. However, as we demonstrated, the fast memory of FE can be easily fulfilled and the control channel is not efficient as well.

C. Other Prior Works

Besides the extreme point simply abandoning reactive model and the straightforward solutions buffering a very limited number of reactive packets (Section III-B), there are broadly three kinds of works indirectly handling the overhead.

Firstly, a great number of papers attempted to increase the processing capacity of controller by utilizing pipeline, concurrency and cache, which meanwhile supports more reactive events [23]. The processing capacity can also be improved by employing hierarchical or distributed models for multi-controller [8].

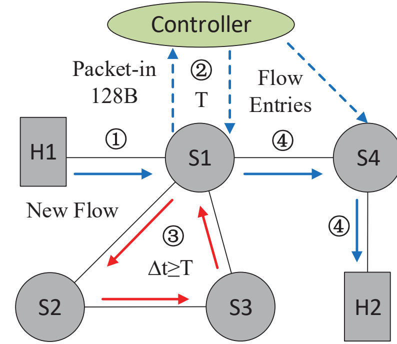

Fig. 1. Basic idea of SoftRing

The second way decreases the chances to reactively involve the controller by improving the forwarding efficiency of switch. For example, DevoFlow [11] delegated some processing tasks to switches. FAST [18] empowered switches to support dynamic actions based on local information. DIFANE [26] proposed authoritative switches for installing rules on the remaining switches.

Thirdly, like treating a cache, effectively decreasing the table-miss rate lowers the probability to involve controller. CAB [25] separates the field space into logical buckets, which are bucked along with all the associated rules. CacheFlow [16] breaks long dependency chains to cache a number of small groups of rules holding equivalent semantics.

The proposed SoftRing is a fundamentally different approach that attempts to decrease the overhead handling each reactive event, which is complementary to the prior works improving the processing ability (the 1st category) or reducing the number of reactive events (the 2nd and the 3rd category).

In the context of optical communication, a long fiber forming a loop path is used as the "buffer" [17], [20], which is not relevant to the study in this paper.

IV. THE DESIGN OF SOFTRING

We now present the design of SoftRing, which ensures sufficient available reactive connections between controller and switches. Its idea is simple: instead of storing the reactive packets in a single switch or sending to the controller, it keeps them in the switches along the pre-computed waiting-loop paths.

For example, in Fig. 1, suppose that a packet from H1 to H2 raises a reactive event in S1 (①). With OpenFlow, S1 first keeps all the reactive packets in its buffer and then forwards them to controller if buffer overflows. In contrast with SoftRing, S1 sends the first a few bytes of a reactive packet to the controller (②), and directly forwards the entire packets to a pre-computed waiting-loop path, *i.e.*, S1→S2→S3→S1 (③). While the packet looping, the controller issues new flow rules to update corresponding flow tables (S1 and S4 in this case). And when the rules are successfully installed, the packet will

be properly forwarded to H2 next time it backs to S1 through the path S1→S4→H2 (④).

This loop-path solution saves the control bandwidth and avoids the potential buffer overflow by trading off two critical resources: (1) the bandwidth in the loop path against the bandwidth in the control channel while waiting for the new rules, and (2) the fast memories of all switches in the loop path instead of the one in a single switch.

As we utilize the extant network loop to temporally store a reactive packet, it does not mean this system will be frangible. First, the loop routing is controllable because each packet can eventually leave the loop path. Though in the above case the reactive packet leaves the loop path where the loop starts, it is possible for it to stop loop routing at any switch along the waiting-loop path, depending on the controller. Second, the bandwidth occupied by SoftRing is controllable as an independent traffic control queue with limited bandwidth is set in each output port of waiting-loop paths for loop packets.

The conceptual simplicity of the loop-path idea poses three significant challenges as below and we will address them in the following three sections

- How to select waiting-loop paths that ensure reactive model without undue overhead?
- How to enforce the path configurations in the switches?
- How to handle the network dynamics and adjust the waiting-loop paths?

V. Waiting-Loop Paths Generation

In this section, we explore the way to find loops in the network topology graph to form the waiting-loop paths.

The network topology is denoted as $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, where $\mathbb{V} = \{v_i | i \in \mathbb{N}^+, i \leq |\mathbb{V}|\}$ is the set of the vertexes and $\mathbb{E} = \{e_k | k \in \mathbb{N}^+, k \leq |\mathbb{E}|\}$ is the set of edges ($\mathbb{N}^+$ means positive integer). A loop $j \in \mathbb{N}^+$ is a subgraph of $\mathbb{G}$ and is denoted as $\mathbb{G}_j = (\mathbb{V}_j, \mathbb{E}_j)$ satisfying

$$\mathbb{V}_j = \{v_{j,l}\} \subset \mathbb{V}, l = 1, ..., |\mathbb{V}_j|, \tag{1}$$

$$\mathbb{E}_j = \{(v_{j,1}, v_{j,2}), ..., (v_{j,l-1}, v_{j,l}), (v_{j,l}, v_{j,1})\} \subset \mathbb{E}, \tag{2}$$

where $(v_{j,l-1}, v_{j,l})$ denotes a link/edge from $v_{j,l-1}$to $v_{j,l}$. We use I_{ij} to indicate whether loop j contains vertex i or not.

$$I_{ij} = \begin{cases} 1 & v_i \in \mathbb{V}_j \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The problem we addressed in this section is to solve I_{ij}, which guarantees that the loops (specified by I_{ij}) 1) cover all the switches, 2) minimize the latency, and 3) control the bandwidth occupation in the data plane. We accomplish the objective in two steps: collect enough loops from the network topology as the candidates (Section V-A); and then select a subset of proper loops to meet the requirements (Section V-B).

A. Seeking for Loop Candidates

In literature, Johnson presented an algorithm to identify all loops of a directed graph [14]. The basic idea is to first traverse all the strongly connected components[2] in a directed graph, and then activate Depth First Searching (DFS) to each vertex in strongly connected components with block/unblock operation.

While this algorithm cannot be directly used in our case, where the graph is undirected, we borrow the basic idea to develop our preliminary. The difference is that we perform DFS on the biconnected components[3] instead of strongly connected components (not existed for undirected graph). It was demonstrated in [9] that elementary loops are existed in a biconnected components, which can be obtained using Tarjan's algorithm [22]. Notice that it is possible that some vertexes are not be covered by any extant elementary loop and in this case, we introduce the concept of *virtual loop* to cover such vertexes, which is a simple palindromic path $v_1 \to v_2 \to ... \to v_2 \to v_1$.

The time complexity of the above method is as large as $O((n + e)(c + 1))$, where n, e, c are the number of vertexes, edges and the elementary loops. We develop the following pruning methods to accelerate the search and the complete seeking method is described in Algorithm 1.

Graph partition. A super large and complex network topology may lead to an unacceptable computation time of seeking loops. We use Spectral Method [13] to divide the large topology into k subgraphs. In our experiments, k is set as the number of the controllers' CPU cores, so each subgraph can perform the proposed Algorithm 1 simultaneously. However, this operation meanwhile deletes some links in the topology, which may impact the coverage rate of the waiting-loops. We use the uncovered vertexes to be $root$ vertex and re-search for them in the integrated topology.

Loop length control. If the length of a waiting-loop is too short, the reactive packets would traverse a link in the loop several times consuming too much bandwidth. But a simple blind enlargement of the loop length introduces larger extra delay: the packets keep traveling along the waiting-loop path even the rules from controller are installed in the switches. To speed up, there is no need to seek loops of arbitrary length. The details on how to determine the trade-off will be discussed in Section V-B and we control the searching of the loops in a reasonable length range.

Loop scale control. As few waiting-loops can cover all the switches, we restrict the number of found waiting-loops. We meet this goal by adding a restrict condition in DFS. Each vertex is set to root in turn for a round of searching. In each round, we introduce random shuffle algorithm to remainder vertexes, which can efficiently increase the loop diversity. And

[2]Strongly connected component: a subgraph of a directed graph $\mathbb{G}$ that is strongly connected, and no additional edges or vertexes from $\mathbb{G}$ can be included in the subgraph without breaking its property of being strongly connected. Strongly connected: there is a path in each direction between each pair of vertexes in a directed graph $\mathbb{G}$.

[3]Biconnected graph: no articulation points in its vertex set. Articulation point: a vertex such that the number of connected components in $\mathbb{G}$ increases when removed.

Algorithm 1: Loop Seeking

```
   inputs : 𝔾 = (𝕍, 𝔼), LEN_MIN, LEN_MAX
   output: 𝔽 - feasible loop set for Loop Selection
 1 B ← getBi-connectedSet(𝔾);
 2 foreach B_i ∈ B do
 3     foreach root ∈ Vertex(B_i) do
 4         RandomShuffleVertex(B_i − {root});
 5         DFS(root, root, new Stack( ));
 6 Function DFS(vertex v, root, Stack Path) : do
 7     foreach w ∈ B_i[v] do
 8         if w == root ∧ ||Path|| + 1 ≥ LEN_MIN then
 9             𝔽 ← 𝔽 ∪ {Path ∪ v};
10             if found 100 loops then stop DFS;
11         else if ||Path|| < LEN_MAX then
12             DFS(w, root, Path ∪ v);
```

Algorithm 2: Loop Selection

```
   inputs : 𝔽 - feasible loop set from Loop Seeking
            𝕍 - vertex set of graph 𝔾
   output: 𝕊 - selected loop set to cover vertexes
 1 𝕊 ← Ø, Cover ← Ø;
 2 while Cover != 𝕍 do
 3     selected ← Ø;
 4     foreach loop ∈ 𝔽 do
 5         loop.IE ← ||{v|v ∈ loop} − Covering|| / ||loop||;
 6         if loop.IE > selected.IE then selected ← loop ;
 7         else if loop.IE == selected.IE then
 8             selected ← argMin(Delay_loop, Delay_selected);
 9     if selected.IE == 0 then break;
10     𝕊 ← 𝕊 ∪ {selected}, Cover ← Cover ∪ {v|v ∈ selected};
```

we search only k loops in one root. Here is an example for a rough impression. There are 1,521,072 loops (with different start vertexes) in a 6-pod FatTree topology. But only 7 of them are enough to cover all these 45 vertexes. Under the restrict condition, we search only 100 loops in one root and our experiments show that the result of our strategy has no difference with that of full loop searching.

B. Loop Selection Algorithm

In this section, we select waiting-loops ($\mathbb{S} = \{\mathbb{G}_j\}$) from the output of Algorithm 1 $\mathbb{F}$ considering several constraints and objectives. A primary requirement, as listed in (4), is to secure all the vertexes (switches) in the topology being covered (with the help of virtual loops if physical links are not sufficient).

$$\cup \mathbb{V}_j = \mathbb{V}. \tag{4}$$

Let T represent the time interval between the switch sending the packet-in message and the switch being updated the new flow entries from controller, t_{ij} be the delay of switch i to the next hop in loop j, and L_j be the time traversing the whole waiting-loop j. Since SoftRing allows packets to traverse the loop $a_j \geq 1$ times, the extra delay introduced by SoftRing can be formulated in (5) and (6), where α is the extra delay upper bound to be minimized. T and t_{ij} can be measured (see Section VI for details), while the values of a_j, I_{ij} are to be resolved.

$$0 \leq a_j L_j - T \leq \alpha, \tag{5}$$

$$L_j = \sum_i t_{ij} I_{ij}. \tag{6}$$

We use the times that a reactive packet traverses a link i as an indication of bandwidth cost on the link as shown in (7). SoftRing puts a cap on the bandwidth used for the waiting-loop paths using a meter to shape/police the use of the bandwidth for SoftRing, which will be presented in Section VI.

$$C_i = \sum_j a_j I_{ij}. \tag{7}$$

To enable SoftRing, the waiting-loop paths should be proactively installed in switches. As we will mention later in Section VI, it requires one more flow entry installed into the switch for each wait-loop traversing it, *i.e.*, the number of the extra flow entries N_i can be expressed as,

$$N_i = \sum_j I_{ij}. \tag{8}$$

Considering the C_i and N_i, this problem can be addressed to a weighted set cover problem, which is NP-hard [15]. We employ a greedy heuristic algorithm to solve it. The complete Loop Selection algorithm in illustrated in Algorithm 2, where *Cover* represents the vertexes covered by the selected loops, *Increase Effect* (IE) represents the effect can be obtained from adding a new loop to the selected set.

It is easy to prove that there is no other loop set that can cover more vertexes than the output subset $\mathbb{S}$. The time bound of Loop Selection algorithm is $O(\|\mathbb{F}\| \times min(\|\mathbb{F}\|, \|\mathbb{V}\|))$, where $\|\mathbb{F}\|$ is the number of loops found in Algorithm 1 and $\|\mathbb{V}\|$ is number of vertexes in the topology. As we eliminate lots of loops by the loop length requirement, Loop Selection is much faster than Loop Seeking. This greedy algorithm provides an approximation ratio of $O(log(n))$, which is optimal to solve the set cover problem in polynomial time [21].

VI. Switch Enforcement

The enforcement is inherently related to the switch specifications. Without loss of generality, we only assume that each switch 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. In other words, the design should at least support off-the-shelf SDN proposals including OpenFlow, POF, P4, *etc*. Especially, we discuss the following design issues related to enforce the waiting-loop path in switches.

A. How to design flow rules enabling the waiting-loop?

First, a reactive rule with the lowest priority to handle the packets that need to be further processed by the controller. The match fields of that rule depend on the kind of reactive

TABLE II
FLOW RULE TO IMPLEMENT SOFTRING

	Priority	Match	Action	Timeout
ES	0	reactive	Send 128 Bytes to Controller Push VLAN = 1 Set TTL = β Send to next switch by $queue_s$	0
LS	1	VLAN = 1 `in_port` = pre-switch	Decrement TTL Send to next switch by $queue_s$	0

TABLE III
CONTROLLER-TO-SWITCH MESSAGE IN SOFTRING

Type	Priority	Match Field	Action	Timeout
`Flow-add`	> 1	VLAN = None `ip_src ip_dst` etc.	Forwarding	normal
`Flow-add`	> 1	VLAN = 1 `ip_src ip_dst` etc.	Pop VLAN Reset TTL Forwarding	short

event to be handled, and here we demonstrate the flow rule construction of the table-miss event in a single switch s. The reactive rule of table-miss event is with the lowest priority (0) and matches wildcard (*) on all fields, so a table-miss packet will hit this rule. The action of this reactive rule consists of the following operations: (1) send a packet-in message carrying the first 128 bytes of the packet to the controller, (2) append a $LOOP_ID$ to the VLAN field of the packet as a waiting-loop ID, which would be removed when leaving the loop, and (3) forward the packet to the next hop of the pre-computed waiting-loop. Notice that this reactive rule in s can only be matched by the reactive packets triggered in s, called *entrance switch (ES) rule*. Additional rules with different $LOOP_ID$ on VLAN field will handle the reactive packets triggered by other switches that traverse s. Such *loop switch (LS) rules* have higher priority than the ES rule.

B. Is the enforcement robust with switch/controller malfunctioning?

We append Time To Live (TTL) to the packets in the waiting-loop as a guarantee to avoid endless loop routing when errors occur, *e.g.*, the actions from controller are not properly installed or are just lost. TTL of a packet in the loop path decreases in each hop and it will be dropped when TTL turns to be zero. TTL is initialized to a tolerable loop times, denoted as β, and would be reset when leaving the waiting-loop. As some Ethernet packets do not have TTL field, we use MPLS tag to replace VLAN tag for bearing $LOOP_ID$ and use MPLS TTL to avoid endless loop.

C. How many flow table rules are needed in each switch?

A switch traversed by n loops needs n LS flow rules and one ES rule, which are illustrated in Table II. The *Timeout* is set to 0, which means "never be expired", in order to permanently keep the reactive rules. Once controller replies the reactive request, one flow rule is added to handle the subsequent packets of the flow, as well as a temporal rule to handle the reactive packets looping in the waiting-loop path. The temporal rule, which is safe to be timeout shortly (*e.g.*, 1s$\sim$the shortest timeout if being with OpenFlow), pops the added VLAN tag, resets the TTL to its former value and forwards the packets correctly. It is worth noting that these two newly updated rules are prioritized making the packets not to trigger the LS/ES rules anymore. Also, it is not necessarily send back the rule to the original switch triggering the reactive request. In fact, any switch in the loop is capable to be the egress switch. The trade-off about how to choose considers the available flow table rules in each switch and the routing hop length.

D. How many messages are there between controller and switches?

We compare the messages using OpenFlow and SoftRing. An OpenFlow controller will send two messages to entrance switch: a *packet-out* message with buffer id (to fetch the reactive packet buffered in switch) or the entire reactive packet (when buffer overflows); a *flow-add* message to install flow rule (rules) on operating the subsequent packets of the flow. In SoftRing, controller does not send the *packet-out* message, which is very likely to be the entire reactive packet. Instead, one more *flow-add* message is sent to restore the reactive packets in the waiting-loop by removing VLAN tag, resetting TTL and forwarding to the right next hop (Table III). Recall that the reactive packets are not only SYN but also normal packets in a TCP flow, this additional *flow-add* message is expected to be much smaller than the reactive packet, so as to reduce the overhead in control/data channel.

E. How much bandwidth would be consumed in the data plane for the waiting-loop path?

Without any buffer and transmission control in switches along the waiting-loop path, the bandwidth would be consumed by the reactive packets traversing the loop again and again. However, by fully utilizing the available memory and metering/shaping function supported in the switches along the waiting-loop, we can control the bandwidth consumed. Especially, the packets traversing the waiting-loop will be buffered in a dedicated queue under a meter in each switch, therefore only a specified proportion of the bandwidth in the meter would be used for the waiting-loop routing.

VII. HANDLING THE DYNAMICS

In this section, we discuss the adjustment of the waiting-loop paths due to two kinds of network dynamics. First, if a link becomes heavily loaded or even congested, it should not be used for the waiting-loop paths. Second, the dynamics of t_{ij} and T have an important impact on deciding loop length, which is used to accelerate the loop generation progress and to guide the trade-off between consumed data plane bandwidth and processing latency.

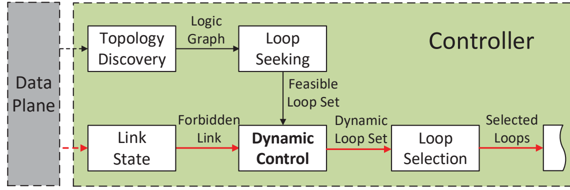

Fig. 2. Dynamic flow control

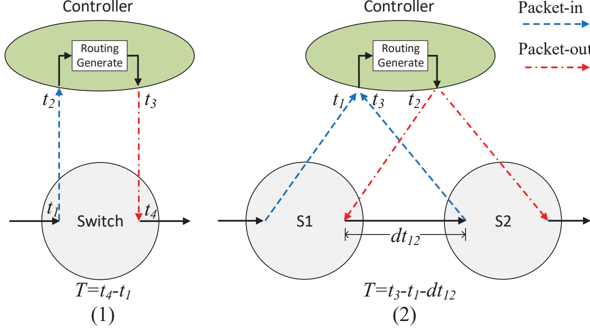

Fig. 3. How to measure the T in SoftRing

A. Traffic Load Dynamics

SoftRing periodically collects the link status (connectivity, utilization), and the overloaded or failed links will be marked as the *forbidden links*. All the loops that cover such links will be removed from the loop set obtained by Algorithm 1, so as to avoid the link congestion or black hole along the loop path.

The change of the forbidden links will refine the loop selection process, while the loop seeking algorithm will be re-executed only when the topology changes. The experiments show that the change of forbidden links brings minor time overhead (<1s), which is fast enough to meet the frequency of the traffic dynamics.

B. Delay Dynamics

As we mentioned before, the length of the waiting-loop path should be carefully estimated as one of the input parameters to the loop seeking algorithm. To estimate the length based on (5) and (6), we need to monitor the delay dynamics.

For t_{ij}, SDN/OpenFlow provides various statistics in the controller to query the switch's running status including the latency dt_{ij} between any two switches i and j through the controller link discover module.

For T, assume that a switch sends packet-in message at t_1, controller receives the packet-in at t_2, after generating rule for the missed packet/flow, the controller sends packet-out at t_3, and the switch receives the packet-out at t_4. In this situation, $T = t_4 - t_1$, but recording t_1 and t_4 requires modifications on switches. As depicted in Fig. 3(2), we estimate the latency in this way: a probe packet is sent to trigger the table-miss in switch S1. Controller receives packet-in from S1 at t_1 and send the packet-out at t_2. Then this probe packet is forwarded to S2 after dt_{12} and triggers the table-miss in S2, too. Controller receives the packet-in from S2 at t_3. Since we have knowledge about dt_{12}, the T is equal to $t_3 - t_1 - dt_{12}$. In this method, we only need to record t_1 and t_3, which are easy to get from the system time in controller. The controller can periodically compute and update the estimated latency and re-execute the loop seeking algorithm on demand.

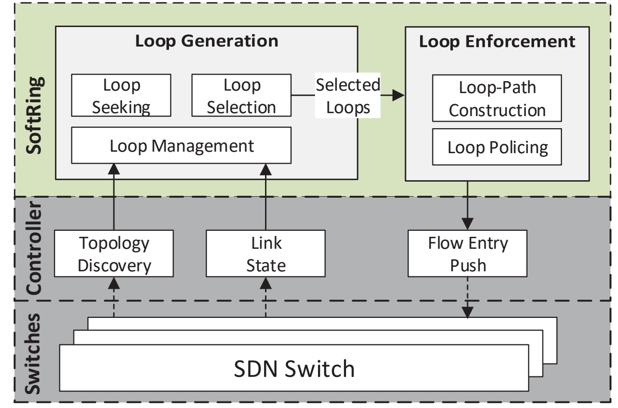

Fig. 4. SoftRing implementation

Notice that both the dt_{ij} and T are not precise in the above measurement. However, the gap is tolerable and the precision is enough for SoftRing as indicated in our evaluation.

VIII. Implementation and Evaluation

A. Implementation

Fig. 4 depicts the prototype system of SoftRing based on the techniques we have discussed in previous sections. In the first stage, a *Loop Seeking* module computes enough feasible loops in the topology (Section V-A), and then a *Loop Selection* module selects a subset of loops to efficiently cover all switches (Section V-B). Notice that the traffic is changing frequently, a *Loop Management* module is therefore developed to adjust the selected loops for controlling the load dynamics (Section VII). In the next phase, a *Loop Enforcement* module enforces the waiting-loops in the data plane (Section VI). To be specific, it can be further separated into a *Loop-Path Construction* module constructing the loop-paths using real rules and a *Loop Policing* module metering the bandwidth utilized in the data plane. SoftRing leverages only the basic information from the controller, thus can be implemented as a built-in component or an individual application. SDN switches with normal abilities (to_controller, meter, forwarding) can support SoftRing with little modification, as buffering in waiting-loop action can be separated into those three actions. We have enforced SoftRing on software SDN switches (Open vSwitch [4]) and ONetSwitch [3], which is a data plane programmable SDN hardware switch.

In order to mitigate those control channel overhead, we implement a *flow-add* action in ONetSwitch. When the first packet triggers the ES entry in SoftRing (Section VI), it can install a new flow entry with higher priority than ES (Flow-ES). The subsequent packets in this flow will match the Flow-ES entry and then send to waiting-loop, thus will not send the same packet-in for this flow. Flow-ES has the shortest timeout, which will be removed immediately after the controller replies the reactive request. Such a modification with the hardware

TABLE IV
TESTING TOPOLOGY

Type	#Switch	#Link	Type	#Switch	#Link
BCube(1,4)	24	32	FatTree(4)	20	32
BCube(2,6)	324	648	FatTree(8)	80	256
BCube(3,8)	6,144	16,384	FatTree(32)	1,280	16,384
DCell(1,4)	25	30	Stanford	26	46
DCell(2,6)	2,107	3,612	CERNET	41	59
DCell(2,8)	5,913	10,512	KDL	754	899
			CAIDA	10,827	37,734

switch on FE is not applied to the software switch in our experiment since the FE capacity is weak in software switch.

B. Experiment Settings

Testbed. We use 4 computers and 20 switches (ONetSwitch) to build two kinds of testbed for evaluation. 4 computers are numbered as H0, H1, H2 and H3 and are with the same configuration (4-core 3.4 GHz Intel CPU, 8 GB memory and Linux kernel 3.13). The first testbed is software switch based. It is built on Mininet [2] and Open vSwitch 2.3.1 [4] to emulate the data plane supporting OpenFlow 1.3, where H0 is equipped with the latest Floodlight controller as well as the SoftRing function running over the controller and H3 is the host of Mininet. We have a second hardware-based testbed based on ONetSwitches [3]. H1 and H2 act as the sender/receiver to exam the performance of SoftRing on hardware switches. A dedicated cable (1Gbps) is employed to connect the controller (H0) and the data plane (ONetSwitches). Each link between any two switches is also 1Gbps.

Metrics. First, we measure the calculation time and the memory occupation to obtain the waiting-loop paths. This metric indicates how effective the proposed algorithms in Section V are. Next, we quantify the benefit achieved by SoftRing on mitigating the reactive overhead, *i.e.*, buffer size in switch and control/data channel bandwidth can be reduced. At last, we exam the cost to deploy SoftRing including the consumed data plane bandwidth, the extra latency, as well as the additional flow table entries used in switches.

C. Efficiency of waiting-loop calculation

Two topology categories are used. One is the Data Center Networking (DCN) topologies including BCube, DCell, FatTree [10]. And the other falls in the Internet topology group obtained from CAIDA [1], Stanford backbone [27], Chinese Education and Research NETwork (CERNET) and Kentucky Datalink (KDL) topologies from the Internet topology zoo [5]. See Table IV for details.

The calculation of the waiting-loop is performed on H0 and Table V illustrates the performance statistics. All the switches are 100% covered by the extant waiting-loops (virtual loops are employed to cover all the switches for the last three topologies in the table). For topologies with less than 100 switches, the proposed algorithms use less than one second and require less than 10MB memory. On handling larger topologies, the calculation could be also accomplished in seconds with quite small memory consumption. The CAIDA topology is the most time consuming one and the computation on it lasts for 36 minutes. The computing time is still acceptable since it can be pre-computed. When we involve dynamic control as mentioned in Section VII-A only calling the loop selection algorithm, the adjustment has been completed within about 1 minute for the same CAIDA topology.

TABLE V
WAITING-LOOP GENERATION RESULTS

Type	Time(s)		Memory(MB)		#	Ave
	Seek	Cover	Seek	Cover	Loop	Entries
BCubea(1,4)	0.00	0.00	2.8	2.4	4	2.3
BCube(2,6)	0.81	0.43	3.6	10.5	58	2.4
BCube(3,8)	122	81.3	90.4	329	1077	2.4
DCell(1,4)	0.00	0.00	2.8	4.2	4	2.4
DCell(2,6)	2.02	3.11	10.0	86.9	304	2.3
DCell(2,8)	12.0	32.3	45.6	139	816	2.3
FatTree(4)	0.00	0.01	2.4	2.9	4	2.6
FatTree(8)	0.03	0.01	3.3	7.8	14	2.3
FatTree(32)	6.49	2.89	128	111	208	2.5
Stanford	0.01	0.00	3.7	3.7	5	2.3
Cernet*	0.00	0.00	3.2	3.2	3	2.7
KDL*	0.12	0.02	7.7	24.2	62	3.3
CAIDA*	2,160	84.8	1,026	733	4,097	4.6

*: Virtual loops are used to cover all the switches.

TABLE VI
THE TIME AND FLOW ENTRIES USAGE WHEN USING SEARCH SCALE CONTROL (SSC)

		BCube (3,8)	DCell (2,6)	FatTree (8)	FatTree (32)
Time	None	1474s	6.06s	230s	$>$1h
	SSC	203s	5.13s	0.04s	9.38s
Ave Entries	None	2.37	2.28	2.32	N/A
	SSC	2.40	2.28	2.32	2.50

In Section V-A, we have introduced many accelerating attempts to the primary loop seeking algorithm. Due to the space limitation, we show the efficiency of the acceleration from “Search Scale Control (SSC)” as an instance: the searching time is significantly reduced while slightly increases the flow table entries to enforce the waiting-loops. The results are demonstrated in Table VI. Although the change for DCell is tiny, the searching improvement on the other three topologies are huge. In addition, the narrowing the search space has only a neglectable impact on the average entries needed.

D. Benefit of SoftRing

In either the software switch or the hardware switch testbed, a full FatTree(4) topology is built with 20 virtual switches (Open vSwitch) or 20 hardware switches (ONetSwitch).

Since each waiting-loop path works independently after enforcing the flow table rules, the results collected on a single loop are representative. It is depicted in Fig. 5 that we test the communication between H1 and H2 triggering reactive packets in switch S1. The bold line in Fig. 5 highlights the waiting-loop tested.

In this evaluation, we do not consider the possible performance degradation due to congestion and only exam the performance differences between original OpenFlow and SoftRing on processing the reactive model. The related works in

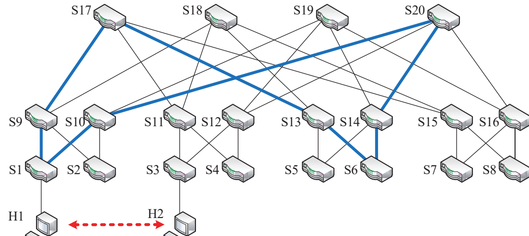

Fig. 5. Waiting-Loop and Connection Hosts in FatTree(4)

TABLE VII
REACTIVE PACKET RATE VS. BUFFER OVERFLOW PROBABILITY IN DEFAULT OPEN VSWITCH

Pkt/s	<750	1000	1250	1500	1750	2000	2250	2500
$P_{overflow}$	0%	2%	4%	7%	14%	38%	64%	91%

Section III-C do not conflict with SoftRing and can work together with SoftRing. We aim to check the benefit achieved by SoftRing alone thus these related works are not involved in the experiments. Table VII lists the buffer overflow probability under different reactive packet rates (number of reactive packet per second). Buffer starts to overflow when the reactive rate reaches 750 packets per second and the buffer overflow probability keeps enlarging with the increment of the table-miss rate. It could be as large as 91% for a rate up to 2500 reactive packets per second.

We have performed a set of experiments to check the control/data channel bandwidth consumption. First, we generate a number of reactive flows, each of which has only one packet, to OpenFlow switch (with and without buffer) and to our software/hardware switch testbed. Fig. 6 depicts the control/data channel bandwidth consumption versus the reactive packet rate with only one switch. SoftRing needs much less bandwidth on handling the reactive model since the size of each packet-in message is 128 Bytes plus the SBI message header (42 Bytes if employing OpenFlow), while OpenFlow has to send the whole packet to the control after buffer overflows, which could be as large as 1500+42 Bytes for each packet-in message. In the worst case, OpenFlow consumes about 5 times control/data channel bandwidth as that is required by SoftRing. When the reactive rate is low, SoftRing occupied very slightly more bandwidth than OpenFlow because SoftRing's flow-add message is a little bit larger than the packet-out message in OpenFlow (Please check Section VI for details). But in that case, the control/data channel bandwidth is not the bottleneck and can afford the messages from SoftRing.

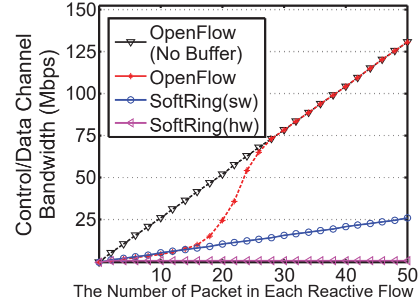

Fig. 7. Control/data channel bandwidth consumed vs. the number of packets in each reactive flow. The total number of the reactive flow generated in one second is 100.

In the second experiment related to control/data channel bandwidth consumption, we modify the injected traffic: we constantly send 100 reactive flows but change the number of packets (1500Bytes for the packet size) in each reactive flow. With the fully programmable data plane of ONetSwitch as we mentioned before, we enable an action in switch to modify the flow tables on detecting the first reactive packet. A Flow-ES entry is appended to the flow table so the subsequent packets for the reactive flow are no longer sent to controller but directly forwarded to the waiting-loop. This entry would be removed later according to the replied flow-mod message from controller. In this way, control/data channel bandwidth overhead can be further mitigated with sending only once the packet-in message to controller for each reactive event. Fig. 7 demonstrates the benefit as we expected: the results achieved by SoftRing with software switch are similar to the previous one in Fig. 6, while the SoftRing on the optimized hardware switch shows a flat curve consuming only 1.7KB/s bandwidth. OpenFlow Design cannot handle such flows in our testbed, we just show its speculative value.

E. Cost of SoftRing

With regarding to the extra cost introduced by SoftRing, we first check how much bandwidth in data plane is reserved for the waiting-loop path. As discussed in Section VI, SoftRing shapes a dedicated queue to send the traffic for SoftRing. In our experiment, the bandwidth used by SoftRing is always less

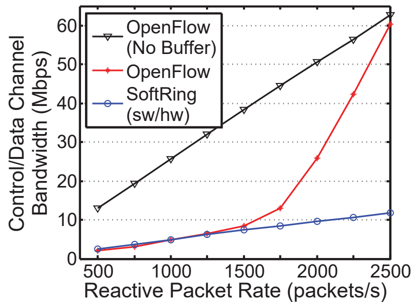

Fig. 6. Control/data channel bandwidth used vs. reactive packet rates. Each reactive flow has only one packet (1500Bytes).

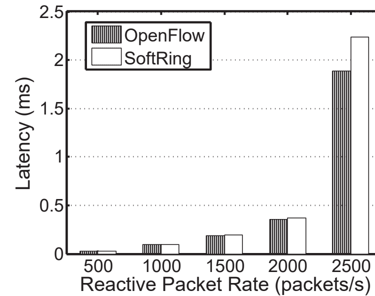

Fig. 8. Different table-miss packet in flow vs. packet Latency (ms) in Software Switch

than 50Mbps (5%) of the link capacity as the control parameter we have configured.

To measure the extra latency of SoftRing compared with OpenFlow, we use the same input traffic in Fig. 6 and SoftRing implementation with the software switch. The results are depicted in Fig. 8. We observe that SoftRing does increase the latency, however the difference is quite small.

Furthermore, Fig. 9 shows the flow table memory used to build the waiting-loop paths. In the DCN topologies and Stanford topology, more than 80% switches contain only 2 or 3 extra static flow entries. The use of virtual links needs more flow table entries like the last three topologies indicated. But in all the topologies, less than 12% switches are required to equip more than 5 flow table entries. The average number of the added flow entries in each experiment with different topologies is around three.

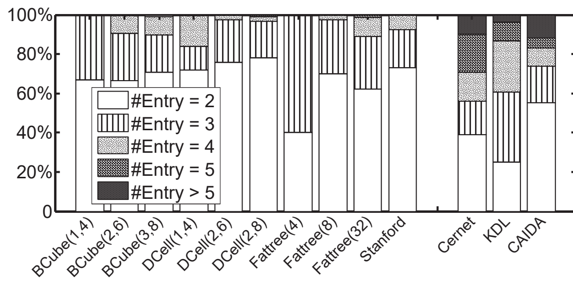

Fig. 9. Accumulated number of static flow entries in one switch

IX. Conclusion

We advocate the coexistence of proactive model and the reactive model: the prior should be adopted for the network issues in expectation; while the latter takes care of the table-miss event and the active involvement of the controller.

We have proposed SoftRing to handle the overhead: a reactive packet goes through a pre-computed waiting-loop path before being correctly operated by the corresponding rules from controller. SoftRing reduced $5\times$ control channel bandwidth consumption compared with OpenFlow.

SoftRing is easy to deploy. It only relies on the basic information from controller and can be a built-in controller component or an individual application. In addition, SoftRing has little requirement on the underlying SDN switch. If the switch could be further optimized for the SoftRing processing, as we have done with the ONetSwitch in the paper, the overhead can be further reduced. We believe an even more powerful programmable data plane device would be soon available with the coming BareFoot switching chip inside P4 production switch. SoftRing slightly increases the processing delay of the reactive model, which is a future direction. A possible way is to combine with the solutions off-loading controller functions to the switches so the latency on the reactive packet would be smaller.

References

[1] CAIDA. "http://www.caida.org/home/".
[2] Mininet. "http://mininet.org/".
[3] ONetSwitch30 specification . "http://onetswitch.org/hardware.html".
[4] Open vSwitch. "http://openvswitch.org/".
[5] The Internet Topology Zoo. "http://www.topology-zoo.org/".
[6] White Paper: DPI & Traffic Analysis in Networks Based on NFV and SDN. "http://www.qosmos.com/wp-content/uploads/2014/01/Heavy-Reading_Qosmos_DPI-SDN-NFV_White-Paper_Jan2014.pdf".
[7] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of ACM SIGCOMM IMC* (2010), ACM, pp. 267–280.
[8] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., et al. ONOS: towards an open, distributed SDN OS. In *Proceedings of HotSDN* (2014), ACM, pp. 1–6.
[9] Birmelé, E., Ferreira, R., Grossi, R., Marino, A., Pisanti, N., Rizzi, R., and Sacomoto, G. Optimal listing of cycles and st-paths in undirected graphs. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms* (2013), pp. 1884–1896.
[10] Chen, K., Hu, C., Zhang, X., Zheng, K., Chen, Y., and Vasilakos, T. Routing in data centers: Insights and future directions. In *IEEE Network Magazine - Special Issue on Cloud* (2011).
[11] Curtis, A. R., Mogul, J. C., Tourrilhes, J., Yalagandula, P., Sharma, P., and Banerjee, S. Devoflow: scaling flow management for high-performance networks. *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 254–265.
[12] Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., and McKeown, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *USENIX NSDI* (2014).
[13] Hendrickson, B., and Leland, R. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing 16*, 2 (1995), 452–469.
[14] Johnson, D. B. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing 4*, 1 (1975), 77–84.
[15] Karp, R. M. *Reducibility among combinatorial problems*. 1972.
[16] Katta, N., Alipourfard, O., Rexford, J., and Walker, D. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of SOSR* (2016).
[17] Langenhorst, R., Eiselt, M., Pieper, W., Grosskopf, G., Ludwig, R., Kuller, L., Dietrich, E., and Weber, H. Fiber loop optical buffer. *Journal of Lightwave Technology 14*, 3 (1996), 324–335.
[18] Moshref, M., Bhargava, A., Gupta, A., Yu, M., and Govindan, R. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of HotSDN* (2014), ACM, pp. 61–66.
[19] Perešíni, P., Kuźniar, M., and Kostić, D. Monocle: Dynamic, fine-grained data plane monitoring. In *Proceedings of CoNEXT 2015* (2015), pp. 32:1–32:13.
[20] Sakamoto, T., Okada, A., Moriwaki, O., Matsuoka, M., and Kikuchi, K. Performance analysis of variable optical delay circuit using highly nonlinear fiber parametric wavelength converters. *Journal of lightwave technology 22*, 3 (2004), 874.
[21] Slavík, P. A tight analysis of the greedy algorithm for set cover. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (1996), ACM, pp. 435–441.
[22] Tarjan, R. Depth-first search and linear graph algorithms. *SIAM journal on computing 1*, 2 (1972), 146–160.
[23] Voellmy, A., Wang, J., Yang, Y. R., Ford, B., and Hudak, P. Maple: simplifying sdn programming using algorithmic policies. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 87–98.
[24] Wen, X., Yang, B., Chen, Y., Li, L. E., Bu, K., Zheng, P., Yang, Y., and Hu, C. RuleTris: Minimizing rule update latency for tcam-based sdn switches. In *IEEE ICDCS* (2016), pp. 179–188.
[25] Yan, B., Xu, Y., Xing, H., Xi, K., and Chao, H. J. CAB: A reactive wildcard rule caching system for software-defined networks. In *Proceedings of HotSDN* (2014), ACM, pp. 163–168.
[26] Yu, M., Rexford, J., Freedman, M. J., and Wang, J. Scalable flow-based networking with difane. *ACM SIGCOMM Computer Communication Review 40*, 4 (2010), 351–362.
[27] Zeng, H., Kazemian, P., Varghese, G., and McKeown, N. Automatic test packet generation. In *ACM CoNEXT* (2012).
[28] Zhang, P., Li, H., Hu, C., Hu, L., Xiong, L., Wang, R., and Zhang, Y. Mind the gap: Monitoring the control-data plane consistency in software defined networks. In *ACM CoNEXT* (2016).
[29] Zhang, P., Wang, H., Hu, C., and Lin, C. On denial of service attacks in software defined networks. *IEEE Network 30*, 6 (Nov. 2016), 28–33.