HiFi: Hybrid Rule Placement for Fine-Grained Flow Management in SDNs

Gongming Zhao[1,3] Hongli Xu*[1,3] Jingyuan Fan[2] Liusheng Huang[1,3] Chunming Qiao[2]
[1]School of Computer Science and Technology, University of Science and Technology of China
[2]Department of Computer Science and Engineering, University at Buffalo, The State University of New York
[3]Suzhou Institute for Advanced Study, University of Science and Technology of China

***Abstract*—Fine-grained flow management is useful in many practical applications, *e.g.*, resource allocation, anomaly detection and traffic engineering. However, it is difficult to provide fine-grained management for a large number of flows in SDNs due to switches' limited flow table capacity. While using wildcard rules can reduce the number of flow entries needed, it cannot fully ensure fine-grained management for all the flows without degrading application performance. In this paper, we design and implement *HiFi*, a system that achieves fine-grained management with a minimal number of flow entries. To this end, *HiFi* takes a two-step approach: wildcard entry installment and application-specific exact-match entry installment. How to optimally install wildcard and exact-match flow entries, however, is intractable. Therefore, we design approximation algorithms with bounded factors to solve these problems. We consider how to achieve network-wide load balancing via fine-grained flow management as a case study. Both experimental and simulation results show that *HiFi* can reduce the number of required flow entries by about 45%-69% and reduce the control overhead by 28%-50% compared with the state-of-the-art approaches for achieving fine-grained flow management.**

I. Introduction

Compared with coarse-grained flow management, fine-grained flow management has irreplaceable advantages for some important applications in a network [1] [2]. For example, researchers have shown that it can improve the success ratio of portscan detection by about 35% through management of small (mice) flows or implementing fine-grained flow management [3]. It is also useful for resource allocation [4], anomaly detection [3] [5], traffic engineering [6] [7], and application identification [8], as well as load-balancing [9].

SDN offers a great opportunity for fine-grained flow management [2] [10]. In an SDN, when a packet arrives at an SDN switch, the switch will match this packet to all rules in the flow table. If there is a matched rule, the switch will forward this packet according to the rule's operation field. Otherwise, the switch will report the packet header to the controller, which determines the route for this flow and installs rules on the switch and on the other ones along the path. In this way, the controller can perform fine-grained management of the flow by deploying per-flow rule (*e.g.*, identified by the 5-tuple, we call it exact-match rule) along its forwarding path. [11]. *In fact, if there is one exact-match rule along its route path, the controller has a chance to control this individual flow by modifying this exact-match rule.*

However, it is far from trivial to achieve fine-grained management with SDNs in practice. R. Cohen *et al.* [12] propose a method that installs exact-match entries on all switches along the forwarding path (to be referred to as *ER* here after) so as to achieve fine-grained flow management. Similar methods have been also used in [4] [13]. However, ER will consume a huge number of flow entries. The problem is exacerbated due to the fact that Ternary Content Addressable Memory, commonly used in commercial SDN switches for storing flow tables/rules, is usually expensive, power hungry and therefore size-limited (*e.g.*, 16,000 entries on high-end Broadcom Trident2 switches [12] or even 1,500 entries on HP 5406zl switches [14]). Thus, the limited flow-table brings a critical challenge for fine-grained management in an SDN network.

An alternative solution to save the number of flow entries is to combine the default path and exact-match rules (to be referred to as *DER* here after), as in OFFICER [15] and HS [16]. Specifically, DER first deploys default paths for all flows, and then installs exact-match rules for some (or all) flows to implement fine-grained management. However, when there is a default path from source to destination, *all matching flows will be directly forwarded to the destination without notifying the controller*, which makes the fine-grained flow management difficult and increases the risk of network attacks [5]. To derive the information of each (individual) flow for fine-grained management, it requires additional devices (*e.g.*, monitor [17]) or software (*e.g.*, statistical modular [18]) to be deployed in the network, which inevitably increases the system setup and maintenance cost.

To overcome the shortcomings of the existing approaches, in this paper, we build *HiFi*, a system targeted at providing fine-grained management for all flows in SDNs while minimizing the number of flow entries that need to be installed on switches without additional hardware and/or software. In other words, the goal of *HiFi* is two-fold: 1) ensuring that each flow will be forwarded by matching at least one exact-match rule along the path from source to destination; and 2) minimizing the number of flow entries needed on switches. *HiFi* enables this by taking a two-step approach: wildcard entry installment and exact-match entry installment. Specifically, wildcard rules are installed to limit the number of flow entries used, while exact-match rule installment can offer application-specific fine-grained

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on November 06,2020 at 01:29:03 UTC from IEEE Xplore. Restrictions apply.

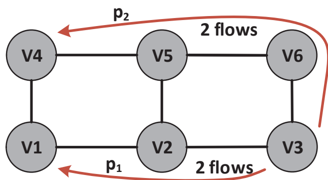


Fig. 1: A network scenario. There are 2 flows from v_3 to v_1 and the other 2 flows from v_3 to v_4. The flow intensity is set to 1 for simplicity. A load-balancing route configuration is illustrated.

schemes	v_1	v_2	v_3	v_4	v_5	v_6	max	total	fine-grained
ER	2	2	4	2	2	2	4	14	✓
DER	1	1	3	1	1	1	3	8	partial
HiFi	1	2	3	1	1	1	3	9	✓

TABLE I: Number of required entries on switches by three entry installment schemes. ER installs exact-match entries on all switches along the forwarding path of each flow. DER installs exact-match entries only for partial flows (not all flows). Our scheme installs exact-match entries on part of switches along a path (*i.e.*, v_2 for p_1 and v_3 for p_2), and the other switches along a path are installed wildcard entries. As a result, our scheme supports fine-grained management with a small number of entries and without additional device/software.

flow management. Together, they can provide a desirable route for each flow from its source to destination. It is worth noting that similar ideas have already been used in data center networks, called Presto, where wildcard rules and exact-match rules are installed on internal switches and edge switches, respectively [19]. However, their solution is only suitable for hierarchical networks (*e.g.*, Fat-Tree [20]), and cannot be efficiently applied to general networks (*e.g.*, HyperX [21]). Even worse, an edge switch with a limited flow-table may become a bottleneck [14]. Therefore, a more general prototype, that can be applied to various networks and can relieve flow-table size constraints, should be proposed.

To apply the idea of *HiFi* to a general network topology, we formulate the optimal wildcard and exact-match entry installment problems using integer linear programs (ILP) by modeling fine-grained management requirements, flow-table size constraints, and link capacity constraints, *etc.*. Unfortunately, neither problems has any optimal solutions in polynomial time. Hence, we design approximation algorithms to solve them, and analyze the approximation factors.

When solving the wildcard entry installment subproblem, we also take into consideration the case where it is impossible to provide fine-grained management for all flows in a network simply because there are too many flows. For example, assume that the network consists of 100 switches, and the flow-table size of each switch is 4,000. As a result, it contains a total of 400,000 entries in the network. When there are 600,000 flows, there's no way to provide fine-grained management for all these flows. For such a case, we design an approximation algorithm to maximize the number of flows that can be controlled individually given the flow- table size constraints.

Extensive simulation shows that *HiFi* helps to reduce the number of required flow entries by 45%-69% and reduce the control overhead by 28%-50% compared with the state-of-the-art solutions.

II. Motivation and HiFi Overview

A. A Motivating Example

This section gives an example to illustrate the advantages and disadvantages of both ER and DER. The usage of entries is summarized in Table I.

Definition 1 (Controllable Flow): If we achieve fine-grained management for a flow (*i.e.*, a flow can be matched by at least one exact-match entry), we call this as a *controllable flow* or say that this flow *can be controlled*.

As shown in Fig. 1, there are 4 flows in the network, 2 flows from v_3 to v_1 and the other 2 flows from v_3 to v_4. For simplicity, the intensity of each flow is set to 1. To achieve load balancing, the route configuration is illustrated in Fig. 1. Specifically, 2 flows follow the path $v_3 \to v_2 \to v_1$, and other 2 flows follow the path $v_3 \to v_6 \to v_5 \to v_4$. Accordingly, the maximum link load (*i.e.*, 2) is minimized.

To realize this routing, ER installs an exact-match entry on each switch along the forwarding path of each flow. Obviously, this scheme can support fine-grained management (*i.e.*, 4 flows are all controllable). However, this scheme will cost more entries, *e.g.*, the total number of consumed entries is 14 in the network (*i.e.*, 3.5 entries per flow on average), and the maximum number of consumed entries is 4 on switch v_3. Considering the limited flow entries on commodity SDN switches, it is impractical for large-scale networks [14].

On the other hand, DER leverages the default paths to reduce the entry cost. We assume that the default path from v_3 to v_1 is $v_3 \to v_2 \to v_1$, and the default path from v_3 to v_4 is $v_3 \to v_2 \to v_1 \to v_4$. When the flows arrive, all flows will be forwarded by default paths directly. In this case, the maximum link load is 4, and no flow is controllable. To reroute some flows and achieve better load balancing, we should deploy additional hardware (*e.g.*, monitor [17]) or software (*e.g.*, statistical modular [18]) to identify those flows and determine their traffic statistics. Then, 2 flows will be re-routed to path $v_3 \to v_6 \to v_5 \to v_4$ by installing exact-match entries. As a result, only two re-routed flows can achieve fine-grained management through exact-match entries and the total number of installed flow entries is 8 in the network. Specifically, we need to install one wildcard entry and two exact-match entries on switch v_3. Although DER can save flow entries, this scheme cannot fully guarantee fine-grained management for all flows (*i.e.*, only 2 flows are controllable). We should note that with more exact-match rules installed, more flows will be controllable.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on November 06,2020 at 01:29:03 UTC from IEEE Xplore. Restrictions apply.

B. Our Intuition

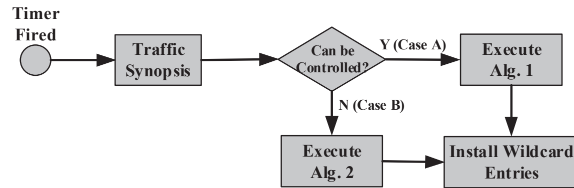
A question immediately following the above discussion is *can we do better by combining the merits of ER and DER?* Clearly, we should use as many wildcards as possible to save flow entries with the constraint that *no flow will be forwarded to the destination only with wildcard entries*. In the meantime, we should break the default path such that no flow will be forwarded to the destination only with wildcard entries. In other words, all flows should be controlled through at least one exact-match entry to achieve fine-grained flow control.

In Fig. 1, for flows from switch v_3 to switch v_1, we install one wildcard entry to match them on switch v_3 and switch v_1, respectively. When two flows, with egress switch v_1, arrive at v_3, they will be directly forwarded to v_2, which cannot find a matching entry for these two flows, and therefore, this flow will be reported to the controller, which can install two exact-match entries on switch v_2 for these two flows to achieve fine-grained management. Similarly, for flows from switch v_3 to switch v_4, we install one wildcard entry on switches v_4, v_5 and v_6, respectively. Besides, we install two exact-match entries on switch v_3. As a result, the total number of installed flow entries is 9, which is almost similar to that by DER. What's more important, *our scheme can achieve fine-grained management for all flows without additional device/software*. We call this scheme as hybrid rule placement for fine-grained management or *HiFi*.

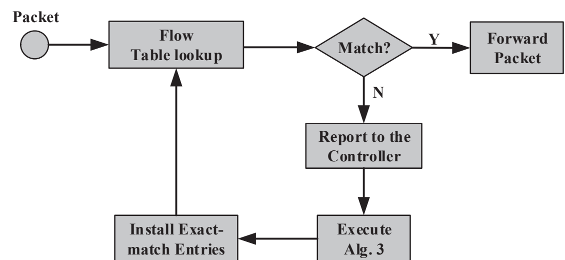
C. Architecture and Workflow of HiFi

HiFi achieves fine-grained flow management through two main control plane modules: wildcard entry installment and exact-match entry installment. The former periodically determines how to install/update wildcard rules (*i.e.*, Flow-Mod) on switches using collected traffic synopsis, while the latter installs application-specific exact-match entries (*i.e.*, Flow-Mod) by processing flow requests (*i.e.*, Packet-In) from the data plane.

Two constraints need to be met when installing the rules: 1) a flow should reach its destination; 2) when a flow is being forwarded to its destination, there will be at least one switch where the flow cannot find a wildcard rule to match. Thus, the switch will report the flow to the controller, and the controller will in turn install an exact-match rule on this switch for this flow. As a result, all following packets of this flow will be controlled by this exact-match entry. One may think that it is natural to study the joint optimization of wildcard and exact-match entries installment. However, due to traffic dynamics, it may not be feasible. If we update a wildcard entry, all flows matched with this entry will be affected, and their routes may be disrupted. Thus, it is inappropriate to update the wildcard entries frequently. Meanwhile, the exact-match entry installment is usually determined by application requirements, *e.g.*, load balancing or throughput maximization [8], and each new-arrival flow will trigger the exact-match installment event. Thus, it is necessary to frequently update the exact-match entries to pursue different application requirements.



(a) Wildcard entry installment triggered by timer.



(b) Exact-match entry installment triggered by new-arrival packets.

Fig. 2: Illustration of the HiFi's workflow.

In *HiFi*, *we will trigger 1) wildcard entry installment using timer, and 2) exact-match entry installment using packets*, which are described as follows:

Step 1 of *HiFi*: As shown in Fig. 2(a), when a timer (*e.g.*, 10min) expires, *HiFi* first estimates the traffic synopsis based on traffic matrix prediction, and decides if it is feasible for all flows to be controlled individually (which will be discussed detailly in Section III-C). If yes, which is referred to as *case A*, *HiFi* determines how to install wildcard entries to minimize the maximum flow entry utilization ratio among all switches (Section III). Otherwise, we cannot provide fine-grained management for all flows, called *case B*. *HiFi* would maximize the number of controllable flows subject to the flow-table size constraint (Section IV).

Step 2 of *HiFi*: As illustrated in Fig. 2(b), when a packet arrives at a switch, it will be handled by the flow table if a matching entry exists. Otherwise, *HiFi* would determine how to install exact-match entries for this flow (Section V).

III. Wildcard Entry Installment for Case A

We first assume that there is a feasible solution which allows all flows to be individually controlled (*i.e.*, we are dealing with case A), and address the wildcard entry installment problem for case A (WEI-A). Moreover, we design an efficient algorithm, and then analyze the approximation performance.

A. Network Model

An SDN network typically consists of three device sets: a cluster of controllers; an SDN switch set, $V = \{v_1, ..., v_n\}$, with $n = |V|$; and a terminal set, $U = \{u_1, ..., u_m\}$, with $m = |U|$. The controllers monitor the network status, and are responsible for route selection of all flows in the network. The switches perform packet forwarding and traffic

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on November 06,2020 at 01:29:03 UTC from IEEE Xplore. Restrictions apply.

measurement for flows based on the flow entries configured by the controller. These switches and terminals comprise the data/forwarding plane of an SDN. Thus, on the view of the data plane, the network topology can be modeled by a graph $G = (U \cup V, E)$, where E is the link set in the data plane.

Assume that there is a set of wildcard rules, denoted as $\mathbb{R} = \{r_1, r_2, ..., r_{m'}\}$, with $m' = |\mathbb{R}|$. For example, a natural way for setting wildcard rules is as follows: we adopt the destination-based wildcard rule (*e.g.*, destination-based OSPF method) for simplicity. Each wildcard rule r_i only specifies the destination u_i, and can match all the sources in the network. In this case, m' is equal to the number of destinations (*e.g.*, m) in a network. The setting of these wildcard rules has been widely used in different applications, *e.g.*, traffic engineering [14] and statistics collection [22].

B. Formulation for the WEI-A Problem

Due to the prior work of traffic matrix prediction on SDNs [23] [24], it is reasonable to assume that we can obtain a flow set, denoted as Π. Moreover, the set of flows passing through switch v to destination u is denoted as Π_u^v, and Π_u denotes the set of flows with destination u. The OSPF path and the destination of each flow $f \in \Pi$ are denoted as h_f and $d(f)$, respectively.

To achieve fine-grained management, each flow will match at least one exact-match entry along its forwarding path. In other words, *each flow should not be forwarded by matching wildcard entries on all switches along the path*. In order to determine how to install wildcard entries for each destination u, we first build a tree T_u rooted at u that branches according to the flow set Π_u. For simplicity, we use the variable q_u^v to denote whether the controller will install exact-match entries on switch v for flows Π_u^v or not. There are two cases for each switch v on tree T_u.

1) $q_u^v = 0$. We install a wildcard entry on switch v for destination u. From the view of saving flow entries, it is no need to install exact-match entries even for a selected fraction of flows Π_u^v.
2) $q_u^v = 1$. The controller will install exact-match entries on switch v for flows Π_u^v. Thus, it requires $|\Pi_u^v|$ exact-match entries on switch v, or we need to reserve $|\Pi_u^v|$ entries for a flow set Π_u^v. Note that how to deploy exact-match entries depends on the current traffic and practical requirements, and will be discussed in Section V.

One may think we could install a wildcard rule with a lower priority, while installing exact-match entries with higher priority for a selected fraction of flows in Π_u^v, which potentially leads to a lower cost of flow entries on switches. In fact, it is not practical on commodity switches. Assume that we have installed a wildcard entry on switch v for flows in Π_u^v. When a new flow arrives, flows will be forwarded directly through switch v by matching a wildcard entry. That means, we have no chance to install exact-match entries with higher priority for a selected fraction of flows in Π_u^v on switch v once flows arrive. Thus, we have to achieve fine-grained management on other switches for flows in Π_u^v and there is no need to install exact-match entries for a selected fraction of flows in Π_u^v on switch v.

As a result, it will cost a total number $b(v)$ of entries (including wildcard entries and reserved exact-match entries) on switch v. Each commodity switch is usually equipped with a limited number of flow entries (*e.g.*, 4,000 entries per switch [14]), and these entries will be shared by routing/measurement/security functions [18] [25]. A natural idea is to minimize the maximum number of required flow entries among all switches. However, it may not be fair for heterogeneous switches in the network. Thus, we expect to minimize the maximum flow entry utilization ratio among all switches in the network, so that the remaining flow entries on each switch can accommodate more flows with exact-match rules. Accordingly, we formulate this problem as follows:

$$\min \ \beta$$
$$S.t. \begin{cases} \sum_{v \in h_f} q_{d(f)}^v \geq 1, & \forall f \in \Pi \\ b(v) = \sum_{v \in T_u, u \in U} (q_u^v \cdot |\Pi_u^v| + 1 - q_u^v), & \forall v \in V \\ b(v) \leq \beta \cdot s(v), & \forall v \in V \\ q_u^v \in \{0, 1\}, & \forall v, u \end{cases} \tag{1}$$

The first set of inequalities denotes that each flow will match at least one exact-match entry, which means that each flow is controllable. The second set of equalities means that the total number of required entries on each switch v is $b(v)$. Note that, $q_u^v \cdot |\Pi_u^v|$ and $1 - q_u^v$ denote the number of reserved exact-match entries and the wildcard entry on switch v for destination u, respectively. The third set of inequalities denotes that the number of consumed flow entries on switch v should not exceed $\beta \cdot s(v)$, where β is the flow entry utilization ratio and $s(v)$ is the flow-table size on switch v. The objective is to minimize the maximum flow entry utilization ratio among all switches, that is, $\min \ \beta$.

C. Algorithm Design for WEI-A

This section develops a rounding-based wildcard entry installment algorithm to solve the WEI-A problem. The proposed algorithm consists of three main steps. The first step will relax the integer program, denoted as LP_1, by relaxing variable q_u^v. We can solve LP_1 in polynomial time with a linear program solver, and obtain the fractional solutions, denoted as $\widetilde{q}_u^v$, $\forall v \in V, u \in U$. In the second step, the fractional solution $\widetilde{q}_u^v$ will be rounded to the 0-1 solution $\widehat{q}_u^v$ to decide where to install wildcard entries for each destination u. The set of unvisited flows is denoted as Π', initialized as all flows in Π. Moreover, we initialize $\widehat{q}_u^v = 0$, $\forall v \in V, u \in U$. We arbitrarily choose an unvisited flow, denoted as f, from set Π'. The algorithm chooses a switch with maximum $\widetilde{q}_{d(f)}^v$ among all $v \in h_f$, and set $\widehat{q}_{d(f)}^v = 1$. That is, we will not install a wildcard entry on switch v for destination $d(f)$. Thus, all flows in set $\Pi_{d(f)}^v$ can be controlled on switch v. We update $\Pi' = \Pi' - \Pi_{d(f)}^v$. This step will terminate when all flows are visited. In the third step, we install wildcard entries based on rounding solutions. For each destination u and switch $v \in T_u$, we install one wildcard entry on switch

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on November 06,2020 at 01:29:03 UTC from IEEE Xplore. Restrictions apply.

v for destination u if $\widehat{q}_u^v = 0$. The algorithm is described in Alg. 1.

Algorithm 1 Wildcard Entry Installment for WEI-A

1: **Step 1: Solving the Relaxed WEI-A Problem**
2: Construct the relaxed problem LP_1
3: Obtain the fractional solutions $\widetilde{q}_u^v$, $\forall v \in V, u \in U$
4: **Step 2: Deriving the 0-1 Solution**
5: $\Pi' = \Pi$
6: $\widehat{q}_u^v = 0$, $\forall v \in V, u \in U$
7: **while** $\Pi' \neq \Phi$ **do**
8: Arbitrarily choose an unvisited flow f from Π'
9: Choose a switch with maximum $\widetilde{q}_{d(f)}^v$ among all $v \in h_f$, and set $\widehat{q}_{d(f)}^v = 1$, $\Pi' = \Pi' - \Pi_{d(f)}^v$
10: **Step 3: Installing Wildcard Entries**
11: **for** Each destination $u \in U$ **do**
12: **for** Each switch $v \in T_u$ **do**
13: **if** $\widehat{q}_u^v = 0$ **then**
14: Install a wildcard entry on switch v for u

Theorem 1: The proposed algorithm can achieve the θ-approximation for the WEI-A problem, where θ is the maximum number of switches visited by each flow.

Due to space limit, we omit the proof of theorem 1.

Note that, after the above algorithm completes, if we find that the total number of required flow entries on some switch exceeds its flow-table size, it means that we cannot provide fine-grained management for all flows due to flow-table size constraint. We will discuss that case in Section IV.

In some practical scenarios, only a specific set of flows (or applications) requires to be controlled with fine-grained management. For example, in a bank network system, there are 10 servers depositing key data, and only flows towards these key servers should be controlled. The other flows can be aggregated for network scalability and resource reusability. To deal with this case, we only need to change the flow set Π in Eq. (1) to the set of flows towards these key servers. Then we can minimize the maximum number of required flow entries to provide fine-grained management for these flows. Due to space limit, we omit the detailed description of this situation.

IV. Wildcard Entry Installment for Case B

In Section III-B, we assume that the flow-table size of each switch is enough to support all flows with fine-grained management. However, when there are too many flows in a large-scale network, we may not be able to provide fine-grained management for all flows due to flow-table size constraint. In this section, we solve the Wildcard Entry Installment problem for case B (WEI-B).

A. Definition of the WEI-B Problem

Under this situation, we just select partial flows for fine-grained management and others for coarse-grained management so as to serve all flows with flow-table size constraint. We formulate this problem as follows:

$$\max \sum\nolimits_{f\in\Pi} \xi_f$$

$$S.t. \begin{cases} \xi_f \leq \sum_{v\in h_f} q_{d(f)}^v, & \forall f \in \Pi \\ \sum_{v\in T_u, u\in U} (q_u^v \cdot |\Pi_u^v| + 1 - q_u^v) \leq s(v), & \forall v \in V \\ \xi_f, q_u^v \in \{0, 1\}, & \forall f, v, u \end{cases} \tag{2}$$

where ξ_f denotes whether flow f is controllable or not. The first set of inequalities denotes that flow f is controllable if this flow will match at least one exact-match entry along the path. The second set of inequalities describes that the required flow entries on each switch $v \in V$ should not exceed its flow-table size $s(v)$. The objective is to maximize the number of controllable flows, which is helpful for different applications, such as traffic engineering or attack detection [3] [4].

B. Algorithm Design for WEI-B

We give an approximation algorithm based on 0-1 knapsack to solve this problem. Before algorithm description, we consider a special case in which there is only one switch in the network. Assume that we have installed wildcard entries for all flows on switch v, and the number of occupied flow entries is $w(v)$. Then we need to replace some wildcard entries with exact-match entries to control some flows for fine-grained management. We can regard this special case as the 0-1 knapsack problem [26]. More specifically, the size of knapsack (or switch v) is the number of residual flow entries, *i.e.*, $s(v) - w(v)$. For each destination u, Π_u^v can be regarded as an individual object. If we install exact-match entries for flows with destination u, then we should install $|\Pi_u^v|$ exact-match entries (one for each flow) and delete the corresponding wildcard entry. Thus, it will increase $|\Pi_u^v| - 1$ flow entries. That is, the cost of Π_u^v is $c(\Pi_u^v) = |\Pi_u^v| - 1$. Besides, the profit of each set Π_u^v, denoted as $p(\Pi_u^v)$, is the number of uncontrolled flows in set Π_u^v. It can be solved by the previous knapsack algorithms, *e.g.*, [27].

This algorithm consists of $|V|$ (*i.e.*, the number of all switches) iterations and each iteration has two steps. In the first step, we adopt the fully polynomial time approximation scheme (FPTAS) algorithm [27] to solve the 0-1 knapsack problem for each residual switch. Then we choose a switch, denoted as v', with the maximum profit among all the residual switches. The FPTAS method for the 0-1 knapsack problem also determines the value of $q_u^{v'}$ for all $u \in U$ (*i.e.*, the individual objects that are put into the knapsack v'). In the second step, the algorithm updates the profit of each object Π_u^v. For simplicity, let $\overline{\Pi}$ be the set of controllable flows. The profit of an individual object $p(\Pi_{u_j}^{v_i})$ is updated as $p(\Pi_{u_j}^{v_i}) = |\Pi_{u_j}^{v_i} - \overline{\Pi}|$. The algorithm will terminate until all switches have been checked. The detailed algorithm is described in Alg. 2.

C. Performance Analysis

In the following, Q_G denotes the set of controllable flows by Alg. 2. In the lth iteration, the controllable flow set is G'_l, and the incremental profit is denoted as X'_l. That is, $X'_l = \omega(G'_l \setminus \bigcup_{i=1}^{l-1} G'_i)$, where $\omega(\cdot)$ denotes its cardinality.

Lemma 2: Alg. 2 achieves a $(2+\epsilon)$-approximation.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on November 06,2020 at 01:29:03 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 Maximizing Controllable Flows for WEI-B

1: $\overline{\Pi} = \Phi$
2: **while** $|V| > 0$ **do**
3: **Step 1: Choosing a switch with the maximum profit**
4: **for** each switch $v_i \in V$ **do**
5: Apply the FPTAS method of 0-1 knapsack to compute the maximum profit $p(v_i)$ for each switch v_i with knapsack size $s(v_i) - w(v_i)$
6: Select switch v' with the maximum profit
7: The installed wildcard rules on v' is denoted as $\mathbb{R}'$
8: **for** each wildcard rule $r_u \in \mathbb{R}'$ **do**
9: $\overline{\Pi} = \overline{\Pi} + \Pi_u^{v'}$
10: $V = V - \{v'\}$
11: **Step 2: Updating the profit of each flow set**
12: **for** each switch $v_i \in V$ **do**
13: **for** each wildcard rule $r_j \in \mathbb{R}$ **do**
14: $p(\Pi_{u_j}^{v_i}) = |\Pi_{u_j}^{v_i} - \overline{\Pi}|$

Proof: Let ψ be the approximation ratio of the FPTAS algorithm for 0-1 knapsack. Consider an instant that Alg. 2 has executed l-1 iterations. In the lth iteration, the algorithm chooses the switch $v_{l'}$. Assume that the optimal solution will select a flow set, denoted as O_l, from switch $v_{l'}$. If we choose O_l instead of G'_l in this iteration, the incremental profit becomes $\omega(O_l \backslash \bigcup_{i=1}^{l-1} G'_i)$, denoted as X''_l. Obviously, we have $\psi \cdot X'_l \geq X''_l = \omega(O_l \backslash \bigcup_{i=1}^{l-1} G'_i) \geq \omega(O_l \backslash Q_G)$. It follows

$$\begin{aligned}\psi \cdot \omega(Q_G) &= \sum_{l=1}^{n} \psi \cdot X'_l \geq \sum_{l=1}^{n} \omega(O_l \backslash Q_G) \\ &= \sum_{l=1}^{n} \omega(O_l \backslash Q_G) \geq \omega(\bigcup_{l=1}^{n} O_l \backslash Q_G) \\ &= \omega(OPT \backslash Q_G) \geq [\omega(OPT) - \omega(Q_G)] \end{aligned} \quad (3)$$

Thus, we have

$$(1+\psi) \cdot \omega(Q_G) \geq \omega(OPT) \quad (4)$$

The FPTAS method achieves the $(1+\epsilon)$-approximation for 0-1 knapsack problem [27], where ϵ is an arbitrarily small value. Thus, by Eq. (4), the proposed algorithm achieves $(2+\epsilon)$-approximation for our problem. ■

V. Exact-Match Entry Installment

In this section, we describe step 2 of *HiFi*: exact-match entry installment, which is triggered by new packet arrival events. When a packet arrives at a switch and there is no matched entries, the switch will report the packet header to the controller. The controller will install exact-match entries to achieve specific application requirements, *e.g.*, load balancing or throughput maximization.

A. Exact-Match Entry Installment for Load Balancing

This section studies the exact-match entry installment for load balancing (MT-LB) as a typical case. To obtain better network performance, we should dynamically update the flow routes so as to adapt to the traffic dynamics [28]. Thus, instead of the long-term traffic observation in Section III, we care for the current flow set Γ, and the traffic size (or intensity) of each flow $\gamma \in \Gamma$ is denoted by $f(\gamma)$. With the system running, the flow set Γ will be updated.

We first introduce how to construct a feasible path set $\mathbb{P}_{v_1}^{v_2}$ from switch v_1 to switch v_2, which is determined by the management policies and performance objectives. If there are too many feasible paths that satisfy the management policies, we may include only a certain number of the best ones under a certain performance criterion, such as having the shortest number of hops or having the large capacities. Then, we explore a feasible path set $\mathbb{P}_\gamma$ for each flow γ. Under the proposed *HiFi* framework, when a flow γ arrives at the network, since some wildcard entries may be installed on some switches, flow γ may be directly forwarded to a switch, denoted by v_γ, in which there is no matching entry for this flow. The forwarding path from the source to switch v_γ is denoted by p_γ^w. Then, we derive a feasible path set $\mathbb{P}_\gamma$ as follows: for each path $p \in \mathbb{P}_{v_\gamma}^{e(\gamma)}$, where $e(\gamma)$ denotes the egress switch of flow γ, we construct a path p' by combining p_γ^w, p, and the link between the egress switch $e(\gamma)$ and the destination $d(\gamma)$. If this path has no loop, we add it to $\mathbb{P}_\gamma$.

The MT-LB problem will select one feasible path from $\mathbb{P}_\gamma$ for each flow γ to achieve load balancing. Let $c(e)$ and $l(e)$ denote the capacity of link e and the traffic load on link e, which is available to the controller by OSPF-TE [13]. The load-balancing factor λ is defined as $\lambda = \max\{\frac{l(e)}{c(e)}, \forall e \in E\}$. We expect to minimize the load-balancing factor, *i.e.*, $\min\ \lambda$.

We give the formulation of the MT-LB problem. Let an indicator variable $y_\gamma^p \in \{0,1\}$ denote whether flow γ will be routed on a path $p \in \mathbb{P}_\gamma$ or not. Let $I(\gamma, p, v)$ be a binary value for exact-match entry installment: if switch v has already installed the wildcard entry for destination $d(\gamma)$, and the next hop of this wildcard entry point to overlaps with the next hop of switch v on path p, then there is no need to install an exact-match entry on switch v, *i.e.*, $I(\gamma, p, v) = 0$; otherwise $I(\gamma, p, v) = 1$. MT-LB solves the following problem:

$$\min\ \lambda$$
$$S.t. \begin{cases} \sum_{p \in \mathbb{P}_\gamma} y_\gamma^p = 1, & \forall \gamma \in \Gamma \\ \sum_{\gamma \in \Gamma} \sum_{p \in \mathbb{P}_\gamma : v \in p} y_\gamma^p \cdot I(\gamma, p, v) \leq \delta(v), & \forall v \in V \\ \sum_{\gamma \in \Gamma} \sum_{p \in \mathbb{P}_\gamma : e \in p} y_\gamma^p f(\gamma) \leq \lambda \cdot c(e), & \forall e \in E \\ y_\gamma^p \in \{0,1\}, & \forall p, \gamma \end{cases} \quad (5)$$

The first set of equations requires that each flow $\gamma \in \Gamma$ will be forwarded through a single path from $\mathbb{P}_\gamma$. The second set of inequalities describes the flow-table size constraint on each switch v, where $\delta(v)$ is the number of residual flow entries on switch v, with $\delta(v) \leq s(v)$. The third set of inequalities states that the traffic load on each link e should not exceed $\lambda \cdot c(e)$, where λ is the load-balancing factor (less than or equal to 1). The objective is to minimize the load-balancing factor λ.

Observing Eq. (5), there are standard rounding-based approximate methods for this problem. An example is relaxation and random rounding algorithm [29], denoted by Alg.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on November 06,2020 at 01:29:03 UTC from IEEE Xplore. Restrictions apply.

3. It relaxes Eq. (5) by replacing the fourth line of integer constraints with $0 \leq y^p_\gamma \leq 1$, and obtains a linear programming problem. We can solve it using the linear programming solver, and derive the optimal solution. Then, we round y^p_γ to zero or one probabilistically based on its fractional value. We use this method in the numerical evaluation of the proposed work and it produces very good results.

In the practical scenarios, many flows may burst in the network, and the controller is unable to provide exact-match entries for each individual flow due to flow-table size constraint. To deal with this case and make our solution more practical, we will choose some switch pairs with less traffic amount, and deploy default paths for these flows. Meanwhile, the controller removes the exact-match rules for these flows so as to set aside some entries for accommodating potential arrival flows.

VI. Performance Evaluation

A. Performance Metrics and Methodology

In this section, we evaluate *HiFi* through small-scale testbed implementation and large-scale simulations, and compare it with the following existing approaches. (1) RLJD [12] installs exact-match entries on all switches along each forwarding path. To enhance the competitiveness of this algorithm, we modify the per-flow routing strategy in the final step by heuristically aggregate per-flow entries based on destination while for each flow leaving at least one switch to keep the per-flow entry in order to satisfy the "controllable" constraint. After this modification, RLJD can achieve better performance than the original one. (2) Presto [19] is designed for hierarchical networks to achieve load balancing. Specifically, it installs exact-match entries on edge switches to control new arrival flows and installs wildcard entries on internal switches to relieve the load of core switches. For a fair comparison, we also extend this design to non-hierarchical network. Specifically, we install an exact-match entry only on the egress switch for each flow and install wildcard entries on other switches.

To compare the performance of three algorithms, we use the following performance metrics in our evaluation: (1) The number of Packet-in messages; (2) The number of controllable flows; (3) The number of required flow entries; (4) The control overhead (the communication traffic volume to/from the controller); (5) Flow setup delay; (6) Packet loss ratio; (7) The maximum throughput of the network; and (8) Load-balancing factor λ. When a flow arrives at a switch, and it does not match any existing entries on the flow table, the OpenFlow Agent of the switch will encapsulate the packet into a Packet-in message and send to the controller for requesting routing strategy. We measure the maximum number of encapsulated Packet-in messages on any switch during the simulation as the first metric. Once receiving the Packet-in message of this flow, the controller controls this flow, then we obtain the second metric. The controller computes the route and sends Flow-mod commands to corresponding switches for entry installment. We measure the maximum number of required flow entries on any switch at any time and the maximum communication traffic volume to/from the controller during the simulation as the third and fourth metrics. Once the flow entries are installed, the flow can be forwarded to destination according to matched flow entries. We measure the maximum flow setup delay and packet loss ratio of all flows as the fifth and sixth metrics. Meanwhile, we measure the maximum throughput that the network can support and the traffic link $f(e)$ of each link e. Then, we compute the load-balancing factor $\lambda = \max\{f(e)/c(e), e \in E\}$.

The simulations are performed under two scenarios. The first scenario has no flow-table size (FTS) constraint, assuming that the switches have sufficient entries to handle all flows. This hypothetical scenario tests how well three algorithms perform when the FTS is sufficient. The second scenario has an FTS constraint and tests the performance of these algorithms when the FTS is limited.

B. Testbed Evaluation

1) Implementation On the Platform: Our testbed is built on a real topology obtained from the Internet Topology Zoo [30], called Epoch [31].The SDN platform is mainly composed of three parts: a controller, 6 Open vSwitches (Version 2.5.3) [32] and 5 virtual machines (acting as terminals). Each Open vSwitch and its connected virtual machines are run on a server with a core i5-3470 processor and 8GB of RAM. The link capacity is set as 200Mbps for simplicity. We use the OpenDaylight Lithium-SR1 release [33] as the controller software running on a server with a core i7-8700k processor and 16GB of RAM.

We implement our tests with a set of synthetic and realistic workloads. Similar to previous works [18] [19] , our synthetic workloads include: (1) random flows, each terminal sends to several random terminals; (2) server flows, random terminals send the traffic to a number of designated terminals. These flows can simulate the traffic of mail servers and web servers; (3) associative flows, these flows simulate the communications between a terminal and a number of designated terminals, *e.g.*, traffic between the finance department and the database.

2) Testing Results: We run four sets of experiments on the SDN platform and execute each experiment 50 times and average the numerical results for accuracy. The first two sets of experiments are performed without flow-table size (FTS) constraint, the default number of flows is set as 300. The last two sets of experiments are performed with flow-table size (FTS) constraint and the default flow-table size is set as 100.

In the first experiment, we observe the number of required flow entries and the number of Packet-in messages on all switches. The testing results are shown in Figs. 3-4. Fig. 3 indicates that *HiFi*, Presto and RLJD need 69, 80 and 172 flow entries at most, respectively, which means that our proposed algorithm can reduce the maximum number of required flow entries by about 15% and 60% compared with Presto and RLJD, respectively. Fig. 4 shows that *HiFi*, Presto and RLJD generate 129, 149 and 231 Packet-in messages at most on any switch during the experiment,

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on November 06,2020 at 01:29:03 UTC from IEEE Xplore. Restrictions apply.

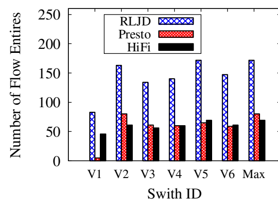


Fig. 3: No. of Flow Entries on Each Switch

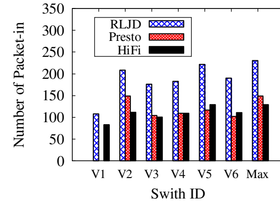


Fig. 4: No. of Packet-in on Each Switch

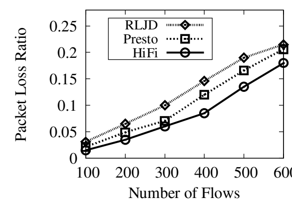


Fig. 5: Packet Loss Ratio vs. No. of Flows

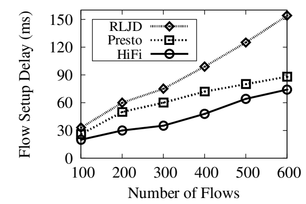


Fig. 6: Flow Setup Delay vs. No. of Flows

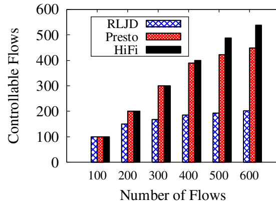


Fig. 7: No. of Controllable Flows vs. No. of Flows

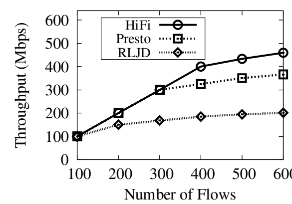


Fig. 8: Throughput vs. No. of Flows

respectively. That is because *HiFi* has pre-deployed some wildcard entries, which has reduced the interaction between data plane and control plane when flows arrive at switches. The work presented in [35] has illustrated that encapsulating Packet-in messages is time-consuming and cpu-consuming for low end CPU of switches and most commodity switches can only encapsulate Packet-in messages at the rate of 150 per second. The following experiments also indicate that too many Packet-in messages may cause high packet loss ratio and latency.

In the second experiment, we observe the packet loss ratio and flow setup delay by changing the number of flows in the network. As shown in Figs. 5-6, when we generate 500 new flows simultaneously using iperf3.3 in the network, *HiFi* can reduce the maximum packet loss ratio of any flows by about 29% and 19% compared with RLJD and Presto, respectively. Meanwhile, *HiFi* only needs 74ms at most to setup a new flow while RLJD needs 154ms and Presto needs 88ms to setup a new flow. Thus, the less interaction between data plane and control plane (*i.e.*, less Packet-in messages and less Flow-mod commands/flow entries) of *HiFi* will make the lower packet loss ratio and lower flow setup delay.

The third set of experiments compares the number of controllable flows by varying the number of flows in the network. As shown in Fig. 7, when there are 100 flows in the network, all algorithms can control all flows with fine-grained management. That is because the flow-table size (*i.e.*,100) is sufficient to handle all flows (*i.e.*,100 flows). However, when there are more flows in the network, *HiFi* can control more flows than the other algorithms under flow-table size constraint. That is because *HiFi* requires fewer entries per flow on average than RLJD, and distributes exact-match entries on all switches more evenly than Presto.

The last set of experiments observes the maximum throughput in the network by changing the number of flows in the network. As shown in Fig. 8, we set the flow-table size as 100. Due to our proposed system uses fewer flow entries, *HiFi* can achieve higher network throughput compared with other algorithms with the increasing of flows. For example, when there are 500 flows in the network, *HiFi* improves the network throughput by about 122% and 24% compared with RLJD and Presto, respectively.

C. Simulation Evaluation

1) Simulation Settings: We select two practical topologies. The first topology, denoted as (a), is a hierarchical Fat-Tree topology [20]. This topology has been widely used in many datacenter networks. It contains 16 core switches, 32 aggregation switches, 32 edge switches and 128 servers. The second one is a non-hierarchical campus network from [36], denoted as (b), containing 100 switches and 200 terminals. Every point in this section is averaged by 100 times. The link capacity is set as 1Gbps for simplicity. The flow size is drawn from random flows, server flows and associative flows discussed in Section VI-B1.

2) Performance Comparison Without FTS Constraint: The first set of simulations compares *HiFi*, RLJD and Presto in the scenario without FTS constraint. The results are shown in Figs. 9-11. Fig. 9 shows that *HiFi* uses much fewer entries than RLJD and Presto. For example, when there are 12×10^4 flows in the Fat-Tree network, *HiFi* reduces the maximum number of required entries by about 45% and 69% compared with Presto and RLJD, respectively.

We plot the load-balancing factor of these algorithms in Fig. 10. To observe the performance of different applications (*i.e.*, load-balancing or throughput maximization), we add another benchmark, denoted as OPT. We note that OPT may denote different solutions for various applications. For load balancing, OPT can be derived by solving relaxed MT-LB using a linear program solver. For network throughput, the OPT solution will gradually increase the traffic intensity until the link capacity is fully utilized.This figure shows that our solution only increases load-balancing factor by about 3% and 5% compared with RLJD and OPT, respectively. Besides, *HiFi* achieves better load-balancing factor than Presto, especially in a non-hierarchical campus network.

Fig. 11 shows that *HiFi* can achieve similar control communication overhead compared with Presto and achieve much smaller control communication overhead compared with RLJD. As the number of flows increases, RLJD installs

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on November 06,2020 at 01:29:03 UTC from IEEE Xplore. Restrictions apply.

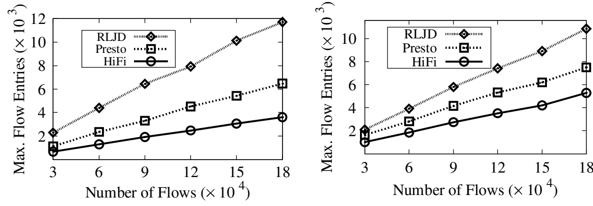


Fig. 9: Max. Flow Entries vs. No. of Flows Without FTS Constraint. *Left plot*: (a) Fat-Tree; *right plot*: (b) Campus.

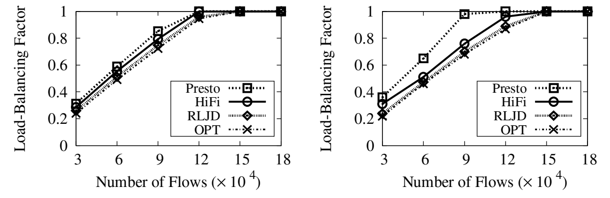


Fig. 10: No. of Flows vs. λ Without FTS Constraint. *Left plot*: (a) Fat-Tree; *right plot*: (b) Campus.

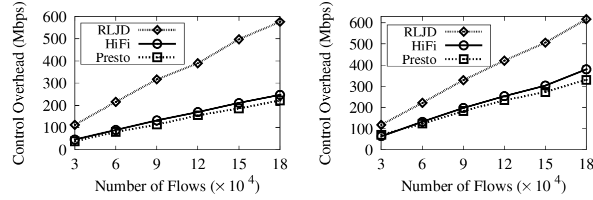


Fig. 11: Control Overhead vs. No. of Flows Without FTS Constraint. *Left plot*: (a) Fat-Tree; *right plot*: (b) Campus.

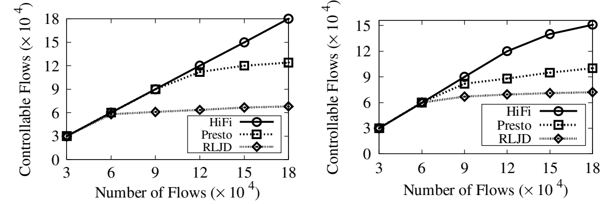


Fig. 12: Impact on No. of Controllable Flows With FTS Constraint. *Left plot*: (a) Fat-Tree; *right plot*: (b) Campus.

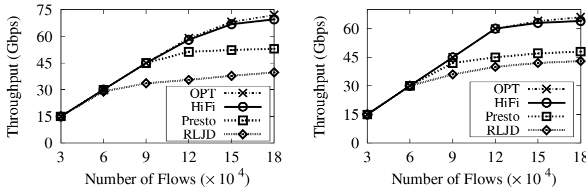


Fig. 13: Throughput vs. No. of Flows With FTS Constraint. *Left plot*: (a) Fat-Tree; *right plot*: (b) Campus.

more flow entries than *HiFi* and Presto, which results in higher control overhead than two other solutions. For example, when there are 12×10^4 flows in the campus network, the control overhead of RLJD, Presto and *HiFi* will reach 421Mbps, 234Mbps and 253Mbps, respectively.

3) Performance Comparison With FTS Constraint: The second set of simulations compares *HiFi*, RLJD and Presto in the scenario with FTS constraint, where the FTS constraint is set as 4,000 on each switch by default [14]. We first compare the number of controllable flows by changing the number of flows in the network. The results are shown in Fig. 12. We claim that our solution can control more flows than the other two algorithms. For example, as shown in the right plot of Fig. 12, when there are 15×10^4 flows, our proposed algorithm can control about 14×10^4 flows while the other two algorithms can only control 7.1×10^4 and 9.5×10^4 flows, respectively. It means that *HiFi* increases the number of controllable flows by about 48% and 97% compared with Presto and RLJD, respectively. Fig. 13 shows that when the number of flow entries is constant (*i.e.*, 4,000), the network performance (*e.g.*, throughput) of our algorithm is much better than that of Presto and RLJD. For example, when there are 15×10^4 flows in the Fat-Tree network, our proposed algorithm can improve throughput by about 28% and 77% compared with Presto and RLJD, respectively. Moreover, *HiFi* can achieve similar throughput compared with OPT, which means high effectiveness of our proposed approximation algorithms.

From these simulation results, we can draw some conclusions. First, from Figs. 9-11, when there is no FTS constraint, *HiFi* can reduce the number of maximum flow entries by about 45% and 69% compared with Presto and RLJD, respectively. Accordingly, *HiFi* decreases the control overhead by about 40% compared with RLJD. Besides, our proposed algorithm can achieve similar performance (*e.g.*, load-balancing factor, throughput) compared with RLJD and OPT. Moreover, *HiFi* can improve network performance by about 38% compare with Presto while using a similar (or less) number of flow entries in a non-hierarchical network. Second, from Figs. 12-13, when the FTS is limited, our algorithm can improve the number of controllable flows by about 48% and 97% and improve the throughput by about 28% and 77% compared with Presto and RLJD, respectively.

VII. Conclusion

In this paper, we have designed *HiFi*, which provides fine-grained flow management with a limited number of flow entries using a novel hybrid (wildcard and exact-match) rule placement scheme. Several algorithms with bounded approximation factors have been designed. We have implemented *HiFi* on our commodity SDN platform, and simulation results have shown the high efficiency of *HiFi*.

VIII. Acknowledgement

This research of Zhao, Xu and Huang is partially supported by the National Science Foundation of China (NSFC) under Grants 61822210, U1709217, and 61936015; by Anhui Initiative in Quantum Information Technologies under No. AHY150300. The research of Qiao is supported in part by National Science Foundation (NSF) Grant CNS-1626374.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on November 06,2020 at 01:29:03 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] D. Sajjadi, R. Ruby, M. Tanha, and J. Pan, "Fine-grained traffic engineering on sdn-aware wi-fi mesh networks," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 8, pp. 7593–7607, 2018.

[2] X. T. Phan and K. Fukuda, "Sdn-mon: Fine-grained traffic monitoring framework in software-defined networks," *Journal of Information Processing*, vol. 25, pp. 182–190, 2017.

[3] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang, "Is sampled data sufficient for anomaly detection?" in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, 2006, pp. 165–176.

[4] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *ACM SIGCOMM*, 2013, pp. 15–26.

[5] Y. Zhang, "An adaptive flow counting method for anomaly detection in sdn," in *Proceedings of the ninth ACM CoNEXT*, 2013, pp. 25–30.

[6] S. Agarwal, M. Kodialam, and T. Lakshman, "Traffic engineering in software defined networks," in *IEEE INFOCOM*, 2013, pp. 2211–2219.

[7] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM*, 2013, pp. 3–14.

[8] T. Pan, X. Guo, C. Zhang, J. Jiang, H. Wu, and B. Liuy, "Tracking millions of flows in high speed networks for application identification," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1647–1655.

[9] A. Craig, B. Nandy, I. Lambadaris, and P. Ashwood-Smith, "Load balancing for multicast traffic in sdn using real-time link cost modification," in *Communications (ICC), 2015 IEEE International Conference on*. IEEE, 2015, pp. 5789–5795.

[10] H. Xu, X.-Y. Li, L. Huang, H. Deng, H. Huang, and H. Wang, "Incremental deployment and throughput maximization routing for a hybrid sdn," *IEEE/ACM Transactions on Networking (TON)*, vol. 25, no. 3, pp. 1861–1875, 2017.

[11] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect sdn forwarding with rulescope," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 2016, pp. 1–9.

[12] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "On the effect of forwarding table size on sdn network utilization," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 1734–1742.

[13] D. Katz, K. Kompella, and D. Yeung, "Traffic engineering (te) extensions to ospf version 2," RFC 3630, September, Tech. Rep., 2003.

[14] 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.

[15] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Officer: A general optimization framework for openflow rule allocation and endpoint policy enforcement," in *Proc. IEEE INFOCOM*, 2015, pp. 478–486.

[16] H. Xu, H. Huang, S. Chen, G. Zhao, and L. Huang, "Achieving high scalability through hybrid switching in software-defined networking," *IEEE/ACM Transactions on Networking*, vol. 26, no. 1, pp. 618–632, 2018.

[17] B. Claise, "Cisco systems netflow services export version 9," 2004.

[18] H. Xu, H. Huang, S. Chen, and G. Zhao, "Scalable software-defined networking through hybrid switching," in *Proc. IEEE INFOCOM*, 2017.

[19] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," *ACM SIGCOMM Computer Communication Review*, pp. 465–478, 2015.

[20] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

[21] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "Hyperx: topology, routing, and packaging of efficient large-scale networks," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 41.

[22] H. Xu, Z. Yu, C. Qian, X.-Y. Li, and Z. Liu, "Minimizing flow statistics collection cost of sdn using wildcard requests," in *IEEE INFOCOM*, 2017, pp. 1–9.

[23] P. Cortez, M. Rio, M. Rocha, and P. Sousa, "Internet traffic forecasting using neural networks," in *The 2006 IEEE International Joint Conference on Neural Network Proceedings*. IEEE, 2006, pp. 2635–2642.

[24] A. Azzouni and G. Pujolle, "Neutm: A neural network-based framework for traffic matrix prediction in sdn," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018, pp. 1–5.

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

[26] G. P. Ingargiola and J. F. Korsh, "Reduction algorithm for zero-one single knapsack problems," *Management science*, vol. 20, no. 4-part-i, pp. 460–463, 1973.

[27] M. Bansal and V. Venkaiah, "Improved fully polynomial time approximation scheme for the 0-1 multiple-choice knapsack problem," *International Institute of Information Technology Tech Report*, 2004.

[28] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *Proc. ACM SIGCOMM*, 2014, pp. 539–550.

[29] T. Friedrich and T. Sauerwald, "Near-perfect load balancing by randomized rounding," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*. ACM, 2009, pp. 121–130.

[30] "The internet topology zoo," http://www.topology-zoo.org/.

[31] "The epoch topology," http://www.topology-zoo.org/maps/Epoch.jpg.

[32] "Open vswitch: open virtual switch," http://openvswitch.org/.

[33] "Linux foundation collaborative project," http://opendaylight.org/.

[34] "Iperf3.3," http://software.es.net/iperf/news.html#iperf-3-3-released.

[35] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen, "Scotch: Elastically scaling up sdn control-plane using vswitch based overlay," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 2014, pp. 403–414.

[36] "Simulating network topologies," http://www.ecse.monash.edu.au/twiki/bin/view/InFocus/LargePacket-switchingNetworkTopologies.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on November 06,2020 at 01:29:03 UTC from IEEE Xplore. Restrictions apply.