Optimal Sketch Configuration for Proportional Fairness in Software Defined Networks

IEEE ICNP 2018, Paper # 39

***Abstract*—Flow-level traffic statistics information plays a vital role for many applications, such as network management, attack detection and packet engineering. Compared with TCAM-based counting and packet sampling, sketches can provide flow-level traffic estimation with bounded error using compact data structures, thus have been widely used for traffic measurement. Under many practical applications, several different sketches will be deployed on each switch to support various requirements of traffic measurement. If each arrival packet is measured by all sketches on a switch (*i.e.*, without sketch configuration), it may lead to redundant measurement, and cost massive CPU resource, especially with increasing traffic amount. Due to limited computing capacity on most commodity switches, heavy traffic measurement overhead will seriously interfere with the basic rule operations, especially when some switches need to deal with many new-arrival flows or update routes of existing flows. To address this challenge, we propose an optimal sketch configuration for proportional fairness with per-switch computing capacity constraint (SCP) problem, so that each sketch can measure enough flows without unduly restricting the number of flows measured by other sketches in the network. Due to the NP-hardness of this problem, a greedy-based algorithm with approximation ratio $1/3$ is presented, and its time complexity is analyzed. We implement the proposed sketch configuration solution on the platform. The extensive simulation results and the experimental results show that the proposed algorithm can measure traffic of 44%-91% more flows compared with the existing solutions.**

***Index Terms*—Software Defined Networks, Traffic Measurement, Sketch Configuration, Approximation.**

I. Introduction

A lot of applications benefit from an accurate view of flow-level traffic statistics in a software defined network (SDN). For example, many data centers implement dynamic scheduling of flow routing based on traffic statistics [1]. An accurate global view of individual flows' traffic also helps to achieve better QoS [2], such as higher throughput, and lower latency. Moreover, traffic statistics information is important for network management, such as accounting management, and performance management [3] [4] [5].

There are three different ways for traffic measurement, TCAM-based counting [6] [7], packet sampling [8] [9], and sketches [10] [11], respectively. However, due to limited size of TCAM and SRAM on a commodity SDN switch [12], both TCAM-based counting and packet sampling can only achieve coarse-grained traffic measurement [13]. Though measurement accuracy can be improved by increasing the T-CAM size or sampling rate, the memory/computing resource usage will dramatically increase and pose scalability issues, especially in high-speed networks [14]. On the contrary, sketches can provide fine-grained traffic measurement for individual flows. Unlike the other two measurement solutions, sketches can summarize traffic statistics of all packets using compact data structures with fixed-size memory, while incurring only bounded estimation errors [14]. Many sketch-based solutions have been proposed in the literatures to address different measurement requirements [10] [11]. The applications of some typical sketches are listed in Table I.

TABLE I: Some Typical Sketches and Their Applications

Measurement Tasks	Sketch
Superspreaders/DDoS	Count-Min [15]
Traffic changes detection	k-ary [16]
Flow size distribution	MRAC [14]
Count traffic	Bloom Filter [17]
Heavy hitters	Count-Min [15] Cold Filter [18]
Top-k flow identification	CountSketch [19]

Unfortunately, it is challenging to deploy sketches in practice for two reasons. 1) Sketches (*e.g.*, Count-Min [15]) are only primitives and usually can not provide the flow IDs, which restricts their usages for network management. Instead, they should be supplemented with additional operations to fully support a measurement task. In particular, to collect meaningful traffic statistics, we must add extensions to sketches to make them reversible. That is, additional operations should be added to make the mapping between identification of flow and "Counter" field. For example, Deltoid [20] encodes flow headers with extra counters in each bucket and updates these counters on every packet. As another example, some applications, such as top-k measurement, require additional heap operations to fulfill these tasks. However, such extensions and heap operations incur heavy computing overhead [14]. 2) Sketches are often task specified, and a sketch only measures one simple object, such as heavy hitters and traffic changes detection. To implement various measurement requirements, the SDN controller usually requires different types of statistics knowledge from switches. Therefore, multiple sketches need to be deployed on each switch [13] [14]. For example, OpenSketch [13] deploys Count-Min, k-ray, and Bloom filter sketches on each switch to implement various applications, *e.g.*, heavy hitters, traffic changes detection, and traffic counting.

Consequently, if a packet is measured/processed by all sketches at a switch upon its arrival, the total measurement overhead per packet may be massive, not to mention that today's networks require very high throughput (*e.g.*, beyond 10 Gbps). On the other hand, OpenFlow capable switches usually have limited processing power [21], which needs to be shared between rule operations and statistics collection besides measurement. The testing results in [12] have shown that switch can complete only 275 flow setups per second even without any traffic load. If traffic measurement on sketch costs 50% CPU utilization, Switch can only complete 137 flow setups per second, let alone packet forwarding.

Therefore, *it is important to perform efficient traffic measurement with limited per-switch computing overhead*, so that basic rule operations on each switch are not interfered, especially for high-speed networks. Though some works, *e.g.*, [22], use the hash value of a packet's 5-tuple to distribute traffic measurements among switches, it requires all packets to carry their ingress-egress pairs, which are not available in practical networks [23], and also requires massive memory cost for auxiliary information.

Inspired by the fact that a flow will pass through several switches from its source to destination, we propose to solve the problem by dynamically controlling which sketches (not all sketches) will be turned on or be enabled at each switch such that a flow can be measured by all required sketches in a distributed manner without violating per-switch computing capacity constraint. Due to limited CPU capacity on a switch, we may not be able to measure the traffic of all flows [22]. Thus, our objective is to derive the statistics information of more flows processed by each kind of sketch so as to draw a more accurate view of traffic statistics. Specifically, we consider the optimization of proportional fairness, so that each kind of sketch can measure many flows without unduly restricting the number of flows measured by other sketches. The main contributions of this paper are:

1) We formulate the optimal sketch configuration with per-switch computing capacity constraint for proportional fairness (SCP) problem. The complexity of this problem is analyzed.
2) We then present an efficient algorithm with approximation ratio $1/3$ based on the greedy 0-1 knapsack method, and analyze the time complexity of the proposed algorithm. To make our study more practical, we also discuss how to extend our solution to multi-path.
3) We implement the proposed algorithm on our SDN platform. The testing experiment and extensive simulation results show that our proposed algorithm can measure traffic of 44%-91% more flows compared with the existing solution.

II. Preliminaries

In this section, we first introduce some typical sketches and motivate the problem by presenting microbenchmark on software implementations of these sketches. Then we show our solution with an intuitive example. Last, we formally formulate the optimal sketch configuration problem.

A. *Typical Sketches and Overhead*

The processing pipeline of each sketch usually consists of update operations (*e.g.*, hashing and addition) on the data structure and other operations (*e.g.*, searching in the heap). Both operations incur CPU computing overhead on switches. In the following, we introduce several typical sketches and their operational overhead per packet.

- **Count-Min Sketch [15]**: A Count-Min sketch consists of a two-dimensional array with d rows and w columns, and every element of this data structure is called "*slot*". For an arrival packet, the switch will hash this packet to a certain slot in each row, and increase the counter of this slot by the packet size. Then, the total computing overhead is d hash operations and d addition operations.
- **k-ary Sketch [16]**: A k-ary sketch consists of d rows and w columns. Its update procedure is similar to that of Count-Min. For an arrival packet, it will be hashed to a certain slot in each row, and the slot's counter will be updated. As a result, the total computing overhead per packet is d hash operations and d addition operations. The main difference from Count-Min is the query function. Assume that the two-dimensional array is denoted as $T[i, j]$, with $1 \leq i \leq d$ and $1 \leq j \leq w$. The sum of a row will be firstly computed, denoted as Δ, *i.e.*, $\Delta = \sum_{j=1}^{w} T[1, j]$. Then, its estimation for row i is computed as $\delta_i = \frac{T[i,h_i(f)]-\Delta/w}{1-1/w}$, where h_i is the hash function for row i, and f is flow ID. The flow size estimation is the medium of these d estimated values. Since the estimation function is triggered per flow, and an elephant flow contains a large number of packets, the overhead for the estimation function can be ignored compared with that for updating each packet.
- **CountSketch [19]**: Similar to Count-Min, CountSketch also keeps an array of d rows and w columns, and two sets of hash functions. The first set of hash functions maps flows to slots, and the other set will hash flows to $\{-1, 1\}$. During the update process, CountSketch uses the hash function in the first set to determine a slot. Then the packet size is multiplied by the value of the hash function in the second set, and increased to the slot's counter. To query a flow, CountSketch also queries each row, but selects the median value as the estimated flow size. To track the top-k flows, CountSketch maintains a heap to record the flow keys and their sizes. When a packet arrives, CountSketch updates the two-dimensional array, and checks whether this flow exists in the heap or not. If yes, its heap entry is simply increased by the packet's size. Otherwise, CountSketch will estimate its flow size by querying the two-dimensional array. If the estimated flow size is larger than the minimum one in the heap, this flow will take the place of the flow with the minimum traffic size

in the heap. As a result, it needs $O(1)$ time to check the existence of a flow key, and $O(\log k)$ time to modify the entry in the heap.

TABLE II: Number of CPU Cycles for per Packet's Measurement

Bloom Filter	Count-Min	k-ary
77	78	81
Cold Filter	CountSketch	MRAC
150	350	404

B. Testing Results for Sketches' Overhead

This section will test the sketch's computing overhead per packet on the open virtual switch (OVS) [24], and the average number of CPU cycles (or CPU cycles for abbreviation) is adopted as the metric. In our experiment, our OVS runs on a VMware with 1GB of RAM and 3.7GHz of CPU frequency, and the sketches are integrated with the OVS. The detailed setting of SDN platform is described in Section V-C. Since packet forwarding costs CPU resource through the OVS, we should avoid this impact on the sketch measurement overhead. To this end, packet forwarding is disabled during the testing. We mainly test the measurement overhead per packet for six sketches, Bloom Filter, Count-Min, k-ary, Cold Filter, CountSketch, and MRAC, respectively. Each of these sketches has 5 rows and 2000 columns, except of Bloom Filter with 1 row and 2000 columns. For each sketch, we take a test during a minute, and count the number, denoted as g, of measured packets on this sketch. Let G be the CPU frequency. Then, we obtain its average CPU cycles for sketch measurement is $\frac{60 \cdot G}{g}$. The number of CPU cycles for six kinds of sketches is listed in Table II. Obviously, the measurement overhead of CountSketch (350) is much more than that of Count-Min (78), for CountSketch needs additional heap operations besides updating.

C. A Motivation Example for Sketch Configuration

In this section, we give an example to motivate our solution. As shown in Fig. 1, there are three flows $\{\gamma_1, \gamma_2, \gamma_3\}$ in the network. We assume that each flow contains 10 packets. The controller requires to deploy two sketches $\{s_1, s_2\}$ for all three flows. For simplicity, the measurement overhead of each sketch per packet is denoted as 1 unit. In the left plot of Fig. 1, two sketches are enabled on each switch. As a result, the measurement overhead on switches v_1, v_3, v_4, and v_5 is 40. Through proper sketch configuration, each switch just enables one sketch at most, as illustrated in the right plot of Fig. 1, so that (1) each flow can be measured by sketches s_1 and s_2; and (2) the maximum measurement overhead among all switches is reduced. Specifically, the measurement overhead on switches v_1, v_3, v_4, and v_5 is reduced to 20, as described in Table III.

Motivated by this example, we should enable a subset of sketches on each switch for efficient traffic measurement. To this end, there are two different ways. One is called

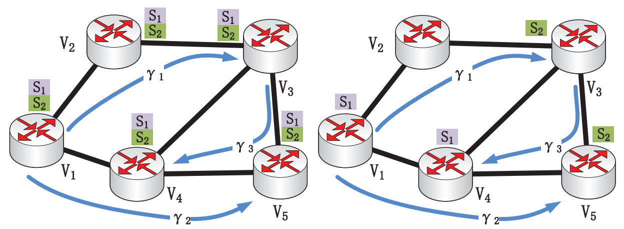

Fig. 1: An Example of Sketch Configuration. *Left plot*: Without sketch configuration; *right plot*: With proper sketch configuration.

TABLE III: Illustration of Measurement Overhead on Switches

	v_1	v_2	v_3	v_4	v_5
Without configuration	40	20	40	40	40
With configuration	20	0	20	20	20

sketch placement [25]. That is, the controller only places those proper (or enabled) sketches on switches. The other is sketch configuration. That is, all sketches are pre-deployed on a switch, and the state of each sketch can be dynamically configured (on or off) by the controller. The first way is memory efficient since each switch only needs to keep a subset of all sketches. However, network traffic is dynamic and ever-changing [23], and the sketch placement needs to be adjusted based on the observed traffic statistics in an online manner to meet the capacity constraints. The former, unfortunately, hinders such processes as the sketch software needs to be compiled to work. On the contrary, the latter supports runtime sketch configuration, which will be validated by our SDN platform described in Section IV.

D. Optimal Sketch Configuration for Proportional Fairness

An SDN typically consists of a logically-centralized controller and a set of switches, $V = \{v_1, ..., v_n\}$, with $n = |V|$. These switches comprise the data plane of an SDN. Thus, the network topology of the data plane can be modeled by $G = (V, E)$, where E is a set of links connecting these switches. Note that the logical controller may be a cluster of distributed controllers [26], which help to balance the control overhead among these controllers. Since we focus on the per-switch measurement overhead, the number of controllers will not significantly impact this metric. For simplicity, we assume that there is only one controller.

We define the optimal sketch configuration for proportional fairness (SCP) problem. Under the general SDN framework, we denote the flow based on originator-destination (OD) pairs as $\Gamma = \{\gamma_1, ..., \gamma_m\}$, with $m = |\Gamma|$. For simplicity, we assume that each flow will be forwarded through only one path, and discuss the extension to the multi-path case in Section III-D. With the advantage of centralized control in an SDN, the controller knows all the installed rules on switches, thus mastering the route path of each OD-pair/flow. Let P_{γ_k} denote the path of flow γ_k. The set of flows (or pairs) passing through switch v_i is denoted as Γ_i. In an SDN network, each switch can count the number of forwarded packets through port statistics, and we can derive the total number of forwarded packets on a switch by adding them together. We denote the set of packets passing through switch v_i as $\mathcal{P}_i$.

To fulfill different application requirements, we assume that each switch has deployed a set of q sketches, denoted as $\mathcal{S} = \{s_1, ..., s_q\}$. These sketches may measure traffic for different objects. For a sketch s_j, its measurement/computing overhead per packet is denoted as $c(s_j)$, which can be measured by the number of CPU cycles. We will configure the status (on or off) of each sketch on a switch. If the status of a sketch is on, we call this sketch "enabled". To avoid the additional control on switches, we assume that each arrival packet will be measured by all enabled sketches on this switch. As a result, the total measurement overhead on each switch v_i is denoted as $c(v_i)$. Due to limited computing power on each switch, we expect a given fraction (*e.g.*, 30%) of computing capacity for traffic measurement. Thus, the computing overhead for traffic measurement on switch v_i should not exceed the threshold C_i.

In many applications, such as flow spread and traffic changes detection, it is expected that more flows can be measured. In the following, if all packets of a flow are measured by sketch s_j, we call that this flow is *covered* by sketch s_j. After sketch configuration, the total number of covered flows by sketch s_j is denoted as β_j. Due to CPU resource constraint on commodity SDN switches, we may not cover all flows. One natural object is to maximize the minimum number of covered flows among all sketches, *i.e.*, max-min fairness. This fairness manner is simple, but may reduce the total number of covered flows measured by all these sketches. Thus, our goal is to propose an optimal sketch configuration in a proportional fairness manner [27] [28]; the configuration allows each sketch measuring enough flows without unduly restricting the number of flows measured by other sketches, *i.e.*, $\max \sum_{j=1}^{k} \log \beta_j$. As another alternative metric, we can maximize the amount of measured traffic in a proportional fairness, which will be discussed in Section III-E.

Accordingly we formalize the optimal sketch configuration problem as follows:

$$\max \quad \sum\nolimits_{j=1}^{q} \log \ \beta_j$$

$$S.t. \begin{cases} \sum_{v_i \in P_{\gamma_k}} x_i^j \geq z_k^j, & \forall \gamma_k, s_j \\ \sum_{\gamma_k \in \Gamma} z_k^j \geq \beta_j, & \forall s_j \\ c(v_i) = \sum_{s_j \in \mathcal{S}} x_i^j \cdot c(s_j) \cdot |\mathcal{P}_i| \leq C_i, & \forall v_i \\ x_i^j, z_k^j \in \{0,1\}, & \forall s_j, v_i, \gamma_k, \end{cases} \quad (1)$$

where x_i^j denotes the result of sketch configuration by the controller. That is, if $x_i^j = 1$, the sketch s_j is enabled on switch v_i. Otherwise, its status is off. The first set of inequalities denotes whether flow γ_k is covered ($z_k^j = 1$) by sketch s_j or not. The second set of inequalities ensures that at least β_j flows will be covered by sketch s_j. The third set of inequalities means the cost on each switch v_i should not exceed C_i, in which C_i is the reserved computing capacity for traffic measurement on switch v_i. The objective is to optimize the proportional fairness among all sketches.

Theorem 1: The SCP problem is NP-hard.

We can prove the NP-hardness by showing that the 0-1 knapsack problem [29] is a special case of SCP by assuming that there is only one switch in the network. Due to limited space, we omit the detailed proof here.

E. Differences to Existing Problems

One may think that the SCP problem seems similar to some existing problems, such as the budgeted maximum coverage (BMC) problem [30] or the maximum coverage problem with group budget constraints (MCG) problem [31]. Before discussing the differences from these two problems, we first give the definition of the BMC problem.

Definition 1 (Budgeted Maximum Coverage (BMC) Problem [30]): Given a ground set X, a collection of sets $M = \{M_1, M_2, ..., M_h\}$, with each set M_j associated with a cost $c(M_j)$ defined over a domain of weighted elements, and a cost threshold $\mathcal{C}$, find a subset of $M' \subseteq M$, such that the total cost of sets in M' does not exceed $\mathcal{C}$, and the total weight of elements covered by M' is maximized.

Differences with the BMC Problem: The BMC problem regards that all the element sets M are put in a group. However, for the SCP problem, each switch corresponds to a group. Thus, there are n groups, in which each group for each switch v_i is associated with a cost constraint C_i.

Definition 2 (Maximum Coverage with Group Budget Constraints (MCG) Problem [31]): Given a ground set M and subsets $\{M_1, M_2, ..., M_h\}$, each set M_j is associated with a cost/budget $c(M_j)$. Given sets $G_1, G_2, ..., G_l$, each G_i, called a group, is a subset of $\{M_1, M_2, ..., M_h\}$. Further, there are given an overall cost $\mathcal{C}$ and a cost $\mathcal{C}_i$ for each group G_i, $1 \leq i \leq l$. A solution is a subset $H \subseteq \{M_1, M_2, ..., M_h\}$ such that the total cost of the sets in H is at most $\mathcal{C}$. For any group G_i, the total cost of the sets in $H \cap G_i$ will not exceed $\mathcal{C}_i$. The objective is to find such a subset H to maximize the size of the union of sets in H.

Differences with the MCG Problem: There are three main differences between SCP and MCG. First, MCG only includes one ground set, while SCP has q kinds of ground sets, one for each sketch. Second, there is no total cost constraint for SCP. Third, the objectives are different between these two problems. The SCP problem aims to optimize the proportional fairness among sketches, while the MCG problem aims to find such a subset H to maximize the size of the union of sets in H. Thus, the solutions for MCG can not be directly used to solve our SCP problem.

III. Algorithm Design of Proportional Fairness

Due to the NP-hardness, it is difficult to solve the SCP problem in polynomial time. In this section, we propose an efficient algorithm, and analyze its approximation factor. We also give some discussion to enhance the algorithm.

A. Algorithm Design

In this section, we present a sketch configuration (SCK) algorithm based on 0-1 knapsack [29] for proportional fairness. The detailed description of the SCK algorithm is given in Alg. 1. Initially, the profit of each sketch on switch v_i is $\log|\Gamma_i|$ (Line 4), where Γ_i denotes the set of flows passing through switch v_i. The cost of each sketch s_j on switch v_i is denoted as $c(s_j^i) = c(s_j) \cdot |\mathcal{P}_i|$, where $c(s_j)$ is the computing overhead per packet and $\mathcal{P}_i$ denotes the set of packets through switch v_i. The algorithm mainly consists of iterations, each of which is divided into two steps. In the first step, for each switch, we compute the maximum profit using the greedy 0-1 knapsack method [29] under a computing cost constraint (Line 8). Then, we choose a switch, denoted as v_i, with the maximum profit, determine the set of enabled sketches on switch v_i, and update the set of covered flows by these sketches. In the second step, we update the profit for each sketch s_j on switch v_i as $p(s_j^i) = \log|\Gamma_i \cup \overline{\Pi}_j| - \log|\overline{\Pi}_j|$ (Line 17), where $\overline{\Pi}_j$ denotes the current set of flows covered by sketch s_j.

Algorithm 1 SCK: Sketch Configuration using 0-1 Knapsack

```
1:  $V$ =a set of all switches.
2:  **for** each switch $v_i \in V$ **do**
3:      **for** each sketch $s_j \in \mathcal{S}$ **do**
4:          $p(s_j^i) = \log|\Gamma_i|$
5:          $c(s_j^i) = c(s_j) \cdot |\mathcal{P}_i|$
6:  **while** $|V| > 0$ **do**
7:      **Step 1: Choosing a switch with maximum profit**
8:      Regard every switch $v_i$ as a package and compute the
        profit $p(v_i)$ using greedy 0-1 knapsack [29]
9:      Select switch $v_i$ with the maximum profit
10:     The enabled sketch set on switch $v_i$ is denoted as $\mathcal{S}'$
11:     Set the status of each sketch in $\mathcal{S}'$ on, and others off
12:     The flow set covered by $s_j \in \mathcal{S}'$ is denoted as $\overline{\Pi}_j$
13:     $V = V - \{v_i\}$
14:     **Step 2: Updating the profit of each sketch**
15:     **for** each switch $v_i \in V$ **do**
16:         **for** each sketch $s_j \in \mathcal{S}$ **do**
17:             $p(s_j^i) = \log|\Gamma_i \cup \overline{\Pi}_j| - \log|\overline{\Pi}_j|$
```

B. Greedy Method for the 0-1 knapsack problem [29]

As described above, the 0-1 knapsack algorithm is a core module for the SCK algorithm. For each switch v_i, we regard each sketch s_j as an item, whose profit and cost are denoted as $p(s_j^i)$ and $c(s_j^i)$, respectively. Our objective is to maximize the total profit of the selected items with a total cost constraint C_i. The knapsack algorithm first computes the profit-cost ratio for each sketch s_j as: $\delta(s_j^i) = \frac{p(s_j^i)}{c(s_j^i)}$. Then, we sort all the sketches by the decreasing order of their profit-cost ratios. Finally, we check each sketch to determine whether this sketch will be selected or not with cost constraint. The formal algorithm is described in Alg. 2.

Algorithm 2 Greedy Method for 0-1 knapsack on switch v_i

```
1: **for** each sketch $s_j \in \mathcal{S}$ **do**
2:     Compute the profit-cost ratio as $\delta(s_j^i) = \frac{p(s_j^i)}{c(s_j^i)}$
3: Sort the sketches by the decreasing order of their profit-
   cost ratios
4: **for** each sketch $s_j \in \mathcal{S}$ **do**
5:     **if** $c(s_j^i) \le C_i$ **then**
6:         $p(v_i) = p(v_i) + p(s_j^i)$
7:         $C_i = C_i - c(s_j^i)$
```

The performance analysis for the greedy algorithm is described next.

Lemma 2: The greedy knapsack algorithm can achieve the approximation factor of $\frac{1}{2}$ for the 0-1 knapsack [29].

Lemma 3: The time complexity of the greedy knapsack algorithm is $O(q \cdot \log q)$.

Due to space limit, we omit the detailed analysis here.

C. Performance Analysis

We first prove a simple conclusion, which will serve performance analysis of the SCK algorithm. Assume that Y and Z are arbitrary non-empty sets.

Lemma 4: $\log|Y_1 \cup Z| + \log|Y_2 \cup Z| \ge \log|Y \cup Z| + \log|Z|$, where $Y_1 \cup Y_2 = Y$, and $Y_1 \cap Y_2 = \phi$.

Proof: Assume that $|Y| = y$ and $|Z| = z$. Moreover, $|Y_1 \cup Z| = z + y_1$, and $|Y_2 \cup Z| = z + y_2$. Then, we have $|Y \cup Z| = z + y_1 + y_2$. Obviously, $(z + y_1) \cdot (z + y_2) \ge z \cdot (z + y_1 + y_2)$. As a result, $\log|Y_1 \cup Z| + \log|Y_2 \cup Z| \ge \log|Y \cup Z| + \log|Z|$. ■

Now, we give the following lemma. Given a set Y, we consider a division $\{Y_1, ..., Y_n\}$ of set Y. That is, $Y_1 \cup ... \cup Y_n = Y$, and $Y_i \cap Y_j = \phi$, $\forall i, j$.

Lemma 5: $\sum_i (\log|Y_i \cup Z| - \log|Z|) \ge \log|Y \cup Z| - \log|Z|$.

Proof: We prove this lemma by induction on variable n. When $n = 1$ or 2, the lemma is proved. We assume that the lemma is proved for any $n \le k$. Now, we consider the case $n = k + 1$.

$$\begin{aligned}
&\sum\nolimits_{i=1}^{k+1} (\log|Y_i \cup Z| - \log|Z|) \\
=&\sum\nolimits_{i=1}^{k-1} (\log|Y_i \cup Z| - \log|Z|) \\
&+ (\log|Y_k \cup Z| - \log|Z|) + (\log|Y_{k+1} \cup Z| - \log|Z|) \\
\ge&\sum\nolimits_{i}^{k-1} (\log|Y_i \cup Z| - \log|Z|) \\
&+ (\log|(Y_k \cup Y_{k+1}) \cup Z| - \log|Z|) \\
\ge&\log|(Y_1 \cup ... \cup Y_{k+1}) \cup Z| - \log|Z| \\
=&\log|Y \cup Z| - \log|Z| \qquad (2)
\end{aligned}$$

Note that the third and fourth inequalities follow by the induction as $n = 2$ and $n = k$, respectively. ■

Let Q_i and Q'_i be two vectors of flow sets as follows: $Q_i = [Q_{i,1}, ..., Q_{i,q}]$ and $Q'_i = [Q'_{i,1}, ..., Q'_{i,q}]$, where $Q_{i,j}$ and $Q'_{i,j}$ are both sets of flows. We define the profit of Q_i

as $\omega(Q_i) = \sum_{j=1}^{q} \log |Q_{i,j}|$. For simplicity, we also define a vector operation as follows:

$$\omega(Q_i \uplus Q'_i) = \sum\nolimits_{j=1}^{q} \log |Q_{i,j} \cup Q'_{i,j}| \tag{3}$$

The incremental profit from vectors Q'_i to Q_i is:

$$\begin{aligned}\omega(Q_i \backslash Q'_i) &= \omega(Q_i \uplus Q'_i) - \omega(Q'_i)\\ &= \sum\nolimits_{j=1}^{q} (\log |Q_{i,j} \cup Q'_{i,j}| - \log |Q'_{i,j}|)\end{aligned} \tag{4}$$

In the following, Q_G denotes a vector of flow sets covered by all sketches after the SCK algorithm. In the l^{th} iteration of SCK, G'_l denotes the vector of flow sets covered by all sketches, and the incremental profit is denoted as X'_l. Obviously $X'_l = \omega(G'_l \backslash \biguplus_{i=1}^{l-1} G'_i)$. For simplicity, the optimal solution for SCP is denoted as OPT.

Lemma 6: The SCK algorithm can achieve the approximation ratio $1/3$ for the SCP problem.

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

$$\begin{aligned}\omega(Q_G) &= \sum\nolimits_{l=1}^{n} X'_l \geq \sum\nolimits_{l=1}^{n} \alpha \cdot \omega(O_l \backslash Q_G)\\ &= \alpha \cdot \sum\nolimits_{l=1}^{n} \omega(O_l \backslash Q_G)\\ &\geq \alpha \cdot \omega(\biguplus\nolimits_{l=1}^{m} O_l \backslash Q_G)\\ &= \alpha \cdot \sum\nolimits_{l=1}^{n} [\omega(O_l \uplus Q_G) - \omega(Q_G)]\\ &\geq \alpha[\omega(\biguplus\nolimits_{l=1}^{n} O_l \uplus Q_G) - \omega(Q_G)]\\ &= \alpha \cdot [\omega(OPT \uplus Q_G) - \omega(Q_G)]\\ &\geq \alpha \cdot [\omega(OPT) - \omega(Q_G)]\end{aligned} \tag{5}$$

Thus, we have

$$\omega(Q_G) \geq \frac{\alpha}{1+\alpha} \cdot \omega(OPT) \tag{6}$$

Since the greedy method achieves the approximation ratio $1/2$ for 0-1 knapsack [29], by Eq. (6), the SCK algorithm can achieve the approximation ratio $1/3$ for SCP. ■

Lemma 7: The time complexity of SCK is $O(n^2 \cdot q + n \cdot m \cdot q)$.

Proof: Initially, the algorithm computes the profits and costs for all sketches on each switch, whose time complexity is $O(n \cdot q)$. Then, there are at most n iterations in the SCK algorithm, and each iteration consists of two main steps. In the first step, the time complexity of the greedy algorithm is $O(q \cdot \log q)$ [29]. Then it takes $O(m \cdot q)$ time to update the sets of covered flows/pairs, where m is the number of pairs in the network. In the second step, we update the profit of each sketch, which takes $O(n \cdot q)$ time. As a result, the total time complexity of the SCK algorithm is $O[n \cdot q + n \cdot (n + q \cdot \log q + m \cdot q + n \cdot q)] = O(n^2 \cdot q + n \cdot m \cdot q)$. ■

D. Extending to Multi-Path

Since multi-path is an efficient way for load balancing and congestion avoidance, it has been widely used in many applications [32]. Adapting the above solution to multi-path requires some modification, but is feasible. For simplicity, we still assume that packets are routed deterministically (*e.g.*, by pre-fixed rules), but may have multiple routes. To implement multi-path routing in SDNs, group entries are required. Each group entry consists of several buckets, each of which consists of an action operation and the weight. As a result, we know the fraction (*i.e.*, the weight in the bucket) of traffic through each path (*i.e.*, specified by the action operation).

Even in the deterministic setting, there are three potential problems. First, if the packets with a heavy feature (*e.g.*, the destination address is heavy) are divided into over many routes, it can increase the difficulty of accurately finding heavy hitters, removing false positives and preventing false negatives. Second, due to multi-path forwarding, one path may only burden a fraction traffic of a flow. Thus, we should revise how to update the profit of a sketch for each switch. Third, since each switch only measures a fraction traffic of a pair, it is another challenge to merge these measured values into an accurate result.

The first issue occurs when multi-path routing reduces the maximum traffic load on each path. We use CountSketch as an example to illustrate this case. For a flow γ_j, its weight is denoted as w_i through switch v_i. When a packet of flow γ_j arrives at switch v_i, the switch will update the bitmap. Then the switch estimates its traffic size, denoted as λ_i. If we directly use λ_i for checking in the heap, it may reduce the counting correctness. We modify the traffic estimation as λ_i/w_i, and use this to execute the heap operations.

To cope with the second issue, we use the path's weight for computing the profit. Specifically, for each switch v_i, assume that all paths through this switch is denoted as P_{v_i}. Then, the initial profit of each sketch s_j on switch v_i is denoted as $\log \sum_{p \in P_{v_i}} w(p)$. We also use the path weight for updating the sketch's profit.

Third, after collecting the sketch statistics from switches, how to merge the measured results into a final result depends on the sketches. For a flow, we have obtained t value pairs, denoted as $< \lambda'_j, w_j >$, with $1 \leq j \leq t$. We take the following sketches as examples.

1) *Count-Min*: For this sketch, we can derive the traffic estimation as $\sum_{j=1}^{t} \lambda'_j / \sum_{j=1}^{t} w_j$.
2) *CountSketch*: After updating the data structure, the switch v_i estimates its flow size as λ'_i/w_i, and then updates the heap. If a flow occurs in one or several heaps of switches, similar to Count-Min, its estimation is derived as $\sum_{j=1}^{t} \lambda'_j / \sum_{j=1}^{t} w_j$.

E. Discussion

In this section, we discuss another version of our SCP problem, whose objective is to measure more traffic with proportional fairness. We formalize this problem as follows:

$$\max \quad \sum\nolimits_{j=1}^{q} \log \ T_j$$

$$S.t. \begin{cases} \sum\nolimits_{v_i \in P_{\gamma_k}} x_i^j \geq z_k^j, & \forall \gamma_k, s_j \\ \sum\nolimits_{\gamma_k \in \Gamma} z_k^j \cdot f(\gamma_k) \geq T_j, & \forall s_j \\ c(v_i) = \sum\nolimits_{s_j \in \mathcal{S}} x_i^j \cdot c(s_j) \cdot |\mathcal{P}_i| \leq C_i, & \forall v_i \\ x_i^j, z_k^j \in \{0, 1\}, & \forall s_j, v_i, \gamma_k \end{cases} \quad (7)$$

The definition of x_i^j is same as that in Eq. 1. The first set of inequalities denotes whether the flow γ_k is covered ($z_k^j = 1$) by sketch s_j or not. The second set of inequalities ensures that the amount of measured traffic by sketch s_j is not less than T_j, where $f(\gamma_k)$ is the traffic size of flow γ_k, can be obtained through sketch measurement. The third set of inequalities means the cost on each switch v_i should not exceed its computation threshold C_i. The objective is to optimize the proportional fairness among sketches.

We note that our proposed SCK algorithm can be applied to solve this problem with some modification, and the approximation factor is $1/3$ too. Due to space limit, we omit the algorithm description and detailed analysis here.

~~IV. Details for System Implementation~~

In this section, we describe some details for system implementation. Our system mainly consists of three main modules. Within a fixed period (*e.g.*, 5 min), we collect (1) port traffic statistics information using the OpenFlow interfaces (*Section IV-A*), and (2) sketch statistics information (*Section IV-B*). Then, the controller runs the SCK algorithm based on port/sketch statistics collection from switches, and dynamically configures the sketches (*Section IV-C*).

A. Port Traffic Statistics Collection

During system running, the controller should master the traffic load on each link to acquire the number $|\mathcal{P}_i|$ of forwarded packets on each switch v_i. The openflow standard specifies the OFPT_PORT_STATUS interface for port traffic statistics collection. Since each link connects with two switches, we can collect port statistics only from a subset of switches to reduce the controller overhead. To this end, we use the minimum set cover algorithm [33] for port traffic statistics collection. Specifically, the controller determines from which switches the port statistics information will be collected and sends a set of the OFPT_PORT_STATUS requests to specified switches. After receiving these requests, the switch will report the statistics information of all ports to the controller. As a result, the controller knows (1) the accurate traffic loads of all ports (links), and (2) the number of forwarded packets on each switch. We should note that the link traffic statistics also help to better deal with traffic dynamics through flow rerouting, which is not the focus of this paper.

B. Sketch Statistics Collection

Different from the port statistics collection, the OpenFlow standard does not provide the interface for s-ketch statistics collection. Thus, we should implement this function in our SDN platform. OVS processes packets through the ingress function OVS_VPORT_RECEIVE. After each packet arrives at a switch and comes into the OVS_VPORT_RECEIVE function, it will be processed by all enabled sketches on this switch. To collect the sketch collection, each switch is connected with the controller using the long-term TCP connection. Then all switches keep listening to controller for the request. Once receiving the request, the sketch statistics are encapsulated into the TCP packets and sent to the controller. At last, the controller receives the sketch statistics from switches using the RECVMSG function.

C. Sketch Reconfiguration

Based on the collected port statistics information (*e.g.*, its traffic load or the number of packets) and the sketch statistics information (the number of flows or the traffic amount of each flow), the controller will reconfigure the sketches on each switch. All sketches can be implemented using C language and integrated as part of OVS datapath kernel module. To configure each sketch, the controller will send commands to control the status (on or off) of each sketch on a switch. Specifically, each switch is connected with the controller using the long-term TCP connection. All switches keep listening to the controller for the command consisting of several 0-1 variables, each of which stands for the state of a sketch through the RECVMSG function. "1" denotes enabled while "0" denotes disabled. Each arrival packet will be processed by these enabled sketches on this switch. Since the number of sketches in the network is limited, the overhead of sending command requests to each switch is very small.

V. Performance Evaluation

In this section, we first introduce the metrics and benchmarks for performance comparison (*Section V-A*). We then evaluate our proposed algorithm by comparing with the random method through simulations (*Section V-B*). ~~Finally, we implement our proposed algorithm on the SDN platform, and provide the testing results (*Section V-C*).~~

A. Performance Metrics and Benchmarks

In this paper, we expect to measure more flows with proportional fairness, which benefits different applications, *e.g.*, traffic engineering, with the constraint of CPU processing capacity through efficient sketch configuration. Thus, we adopt the following metrics in our numerical evaluations.

1) *Flow cover ratio (FCR)*. The controller computes the number of covered flows by each sketch. The flow cover ratio is defined as the number of covered flows dividing the number of all flows in the network.

We denote the flow cover ratio of each sketch as $\theta_j = \beta_j/m$, where β_j is the number of flows covered by sketch s_j and m is the number of flows in the network. In this paper, we measure three metrics of FCR, the average, the maximum and the minimum, respectively. For example, the average flow cover ratio is $\overline{\theta} = \sum_{j=1}^{q}\theta_j/q$.

2) *Variance*. Based on the flow cover ratio of all sketches, we compute its variance to check the property of proportional fairness by different algorithms. We denote the variance as $\sum_{j=1}^{q}(\theta_j - \overline{\theta})^2/q$.

Since there is no existing solution for the SCP problem, we compare the proposed SCK algorithm with the random solution, denoted as *RND*, through both simulations and prototype experiments. Specifically, the controller will randomly choose sketches configured on each switch with the CPU processing capacity constraint.

B. Simulation Evaluation

1) Simulation Setting: In the simulations, as running examples, we select two practical and typical topologies: one for campus networks and the other for data center networks. The first topology, denoted as (a), contains 100 switches, 200 servers and 397 links from [34]. The second one is the fat-tree topology [35], denoted as (b), which has been widely used in many data center networks. The fat-tree topology has in total 80 switches (including 16 core switches, 32 aggregation switches, and 32 edge switches) and 192 servers. To observe the impact of different traffic traces on the measurement performance, we adopt two types of traffic traces. One is the 2-8 distribution. Specifically, the authors of [12] have shown that less than 20% of the top-ranked flows may be responsible for more than 80% of the total traffic. The other is that the traffic size of each flow follows the Gaussian distribution. For simplicity, we assume the uniform packet size (*e.g.*, 1KB), and a flow contains 2.5K packets on average. We adopt six kinds of sketches, Count-Min, CountSketch, Bloom Filter, Cold Filter, MRAC and k-ary, respectively, in our simulation. The sketch computing overhead per packet is listed in Table II. By observing the configuration of some commodity SDN switches, *e.g.*, H3C and Pica8, the switch's CPU capacity is set as 3GHz, and we assume that at most 50% CPU capacity will be allocated for traffic measurement. We execute each simulation 100 times, and take the average of the numerical results.

2) Simulation Results: We run three groups of simulations to check the effectiveness of the SCK algorithm.

The first group of four simulations observes the FCR and its variance by changing the CPU capacity constraints from 0.5GHz to 1.5GHz. We generate 30K flows by default in the network. Fig. 2 shows that the average flow cover ratio is increasing with enhanced CPU capacity constraints for both algorithms. Our SCK algorithm can significantly improve the flow cover ratio compared with the random algorithm. Specifically, given the CPU capacity constraint of 1GHz,

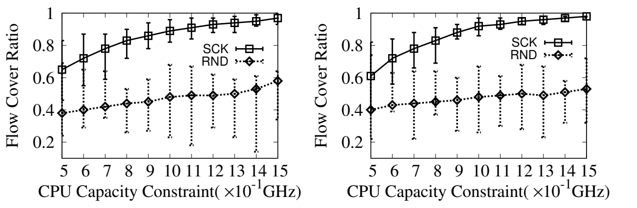

Fig. 2: Flow Cover Ratio vs. CPU Capacity Constraint under 2-8 Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

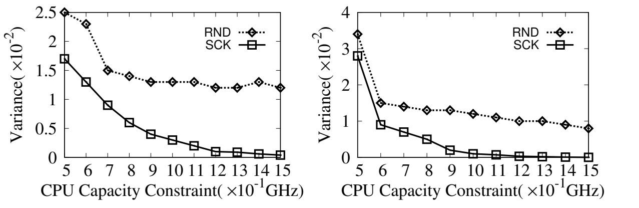

Fig. 3: Variance vs. CPU Capacity Constraint under 2-8 Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

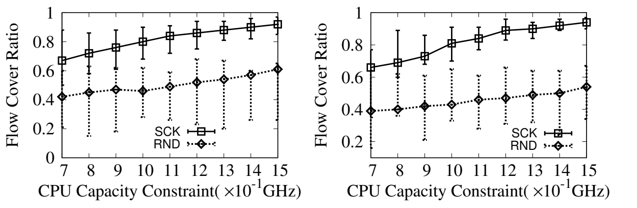

Fig. 4: Flow Cover Ratio vs. CPU Capacity Constraint under Gaussian Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

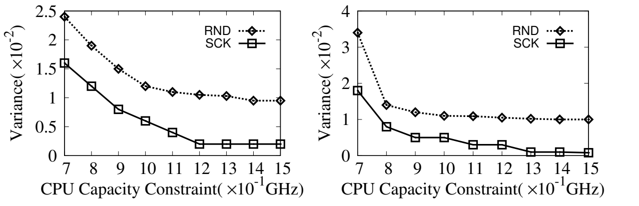

Fig. 5: Variance vs. CPU Capacity Constraint under Gaussian Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

the average, maximum, and minimum FCRs by our SCK algorithm are 0.89, 0.96 and 0.82, respectively, while the random algorithm achieves the average, maximum, and minimum FCRs only about 0.48, 0.68, and 0.23, respectively, by the left plot of Fig. 2. That is, our proposed algorithm can improve the average flow cover ratio by 85% compared with RND. Fig. 3 shows that the variance of the flow cover ratio decreases with enhanced CPU capacity constraint from 0.5GHz to 1.5GHz for both algorithms. The variance of our proposed algorithm is much less than that of the random algorithm. For example, the variance of our SCK algorithm is only about $1/12$-$1/4$ as that of the RND algorithm when the CPU capacity constraint is 1GHz by Fig. 3.

Figs. 4 and 5 plot the FCR and its variance under the Gaussian traffic distribution. The trend of these curves is similar as that of curves in Figs. 2 and 3. When the CPU capacity constraint is 1GHz, we observe that our SCK algorithm can improve the average flow cover ratio 51%-

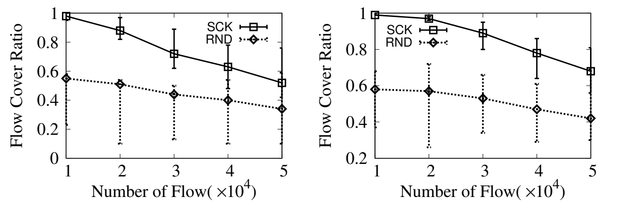
Fig. 6: Flow Cover Ratio vs. Number of Flows Under 2-8 Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

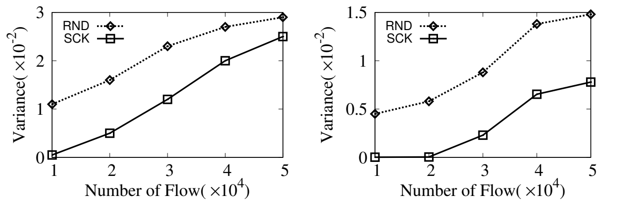
Fig. 7: Variance vs. Number of Flows Under 2-8 Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

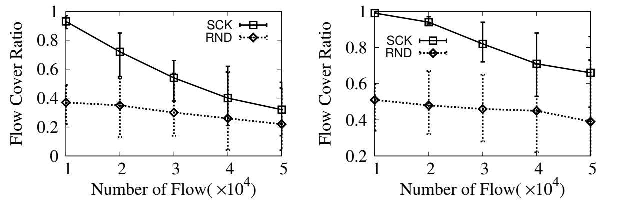
Fig. 8: Flow Cover Ratio vs. Number of Flows Under Gaussian Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

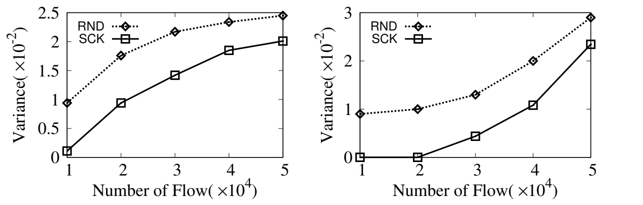
Fig. 9: Variance vs. Number of Flows Under Gaussian Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

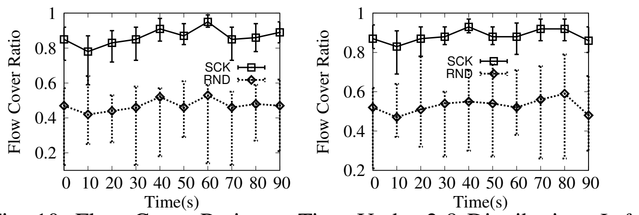
Fig. 10: Flow Cover Ratio vs. Time Under 2-8 Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

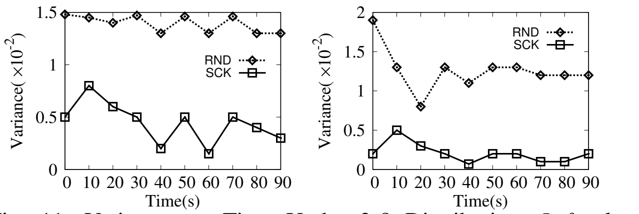
Fig. 11: Variance vs. Time Under 2-8 Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

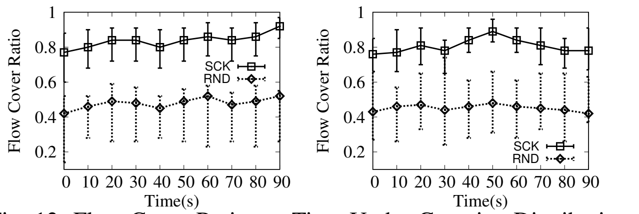
Fig. 12: Flow Cover Ratio vs. Time Under Gaussian Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

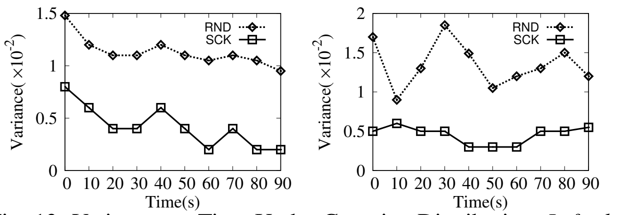
Fig. 13: Variance vs. Time Under Gaussian Distribution. *Left plot*: Topology (a); *right plot*: Topology (b).

89% compared with the RND algorithm by Fig. 4. The variance of the SCK algorithm is only $1/2$ as that of RND.

In the second simulation set, we observe the FCR and its variance by changing the number of flows from 10K to 50K. By default, we set the CPU capacity constraint as 0.8GHz. Figs. 6 and 8 show that the average flow cover ratio is decreasing with more flows in the network. Specifically, when there are 30K flows in the network, the average, maximum and minimum FCRs of our SCK algorithm are 0.72, 0.89, and 0.62 by the left plot and 0.85, 0.95, and 0.80 by the right plot in Fig. 6. Moreover, our SCK algorithm can improve the average FCR 52%-78% compared with the RND algorithm by Fig. 6. Figs. 7 and 9 show the variance of the flow cover ratio in the network. The trend of the curves is increasing with more flows in the network. Specifically, Fig. 7 shows that the variance of the RND algorithm is almost 2 times as that of our SCK algorithm under the 2-8 distribution. While under the Gaussian distribution, the variance of the RND algorithm are almost 1.5 times as that of SCK by the left plot of Fig. 9.

The third set of simulations observes the FCR and its variance of both algorithms over time. In this simulation set, we generate different numbers of flows in each period (*e.g.*, 10s in the simulation). Due to traffic dynamics, the performance of each algorithm fluctuates over time by Figs. 10-13. Our SCK algorithm increases the flow cover ratio by about 65%-85% compared with the RND algorithm by Fig. 10. In Figs. 11 and 13, the variance also varies over time, and the variance of SCK is more less than that of RND. For example in Fig. 11, the variance of the SCK algorithm is almost 1/16-1/6 times as that of RND.

From these simulation results in Figs. 2-13, we can make the following conclusions. First, our SCK algorithm can achieve better flow cover ratio than the RND algorithm. For example, our SCK algorithm can improve the average flow cover ratio by 67%-91% compared with the RND algorithm by the left plot in Fig. 2. Second, our proposed algorithm has the excellent property of proportional fairness about the flow cover ratio. That is, the variance of SCK is smaller than that of the random algorithm. For example, the variance of the RND algorithm is at least 1.5 times as that of SCK by Fig. 3. Thus, our SCK algorithm performs better than the random algorithm on proportional fairness.

C. Test-bed Evaluation

1) Implementation On the Platform: Since a commodity physical SDN switch is usually a closed system, it is difficult to update its software (*e.g.*, firmware). As an alternative way, we then implement the proposed algorithm on the software switches, *e.g.*, Open Virtual Switches (OVS). Moreover, different sketches can be integrated with the OVS.

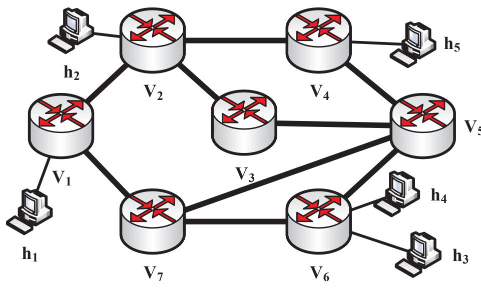

Fig. 14: Topology of the SDN Platform. The data plane of our platform is mainly composed of two parts: seven OpenFlow enabled virtual switches $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$, and five terminals $\{h_1, h_2, h_3, h_4, h_5\}$.

In our SDN platform, there are two main categories of devices. One is the SDN controller. We use Ryu (version 4.13 [36]) as the controller software, running on a server with a core i7-6700 processor and 8GB of RAM. The other is the virtual switch, which is implemented using the OVS 2.7.2 [24], running on a VMware with 1GB of RAM and 3.7GHz of CPU frequency. The topology of our SDN platform is illustrated in Fig. 14. The data plane of an SDN comprises of 7 virtual switches, which support the OpenFlow v1.3 standard. Additionally, five terminals are implemented on the virtual machines. Each flow is identified by three elements, including source IP, source port and destination IP, so that each terminal is able to generate different numbers of flows to others.

We consider a campus network scenario. Since network security is the basic function in the network, Count-Min sketch and Bloom Filter sketch are chosen to detect the DDoS attack and the sniffing attack, respectively. Moreover, to achieve better QoS, four different sketches, k-ary, MRAC, CountSketch and Cold Filter are also adopted to obtain flow statistics information for load balancing and network management. In the experiment, all sketches described in Table I are be integrated with OVS and our SCK algorithm can be tested based on our experiments. Besides, the cost of CPU cycles for each sketch is listed in Table II.

2) Testing results: Since packet forwarding occupies the CPU resource on the OVS, our testing results show that the required number of CPU cycles for forwarding each

TABLE IV: An Instance of Sketch Configuration on Switches

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	FCR
Count-Min	1	0	1	0	1	1	0	1.0
CountSketch	0	1	0	1	0	0	1	0.7
Cold Filter	1	0	1	0	1	0	0	0.9
Bloom Filter	1	1	1	0	1	1	1	1.0
MRAC	1	1	1	0	1	1	1	0.7
k-ary	1	0	1	0	1	0	1	1.0

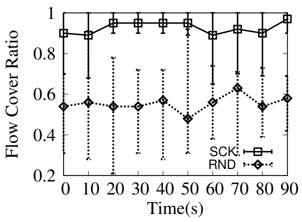

Fig. 15: Flow Cover Ratio vs. Time

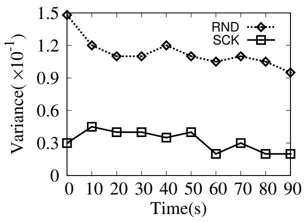

Fig. 16: Variance vs. Time

packet is about 80 with the help of data plane development kit (DPDK) [37]. To ensure the packet forwarding and rule operations correctly, we assume that the CPU resource for traffic measurement is set as 1GHz (or about 30%). In the following testing, we generate 2,000 flows which are under 2-8 distribution by default. Specifically, each of top-20% flows include 6,000-10,000 packets, and each of other flows includes 200-600 packets. We note that each flow contains a large number of packets so as to increase the total sketch measurement overhead during the testing.

After running our SCK algorithm, an instance of sketch configuration on all switches is listed in Table IV, where "1" denotes enabled and "0" denotes disabled. For example, CountSketch, Bloom Filter, and MRAC are enabled on switch v_2. Besides, the flow cover ratios of all sketches are also listed in Table IV. By this testing, we can obtain the network throughput of 3.84M packets per second. Then, we check the effectiveness of our proposed algorithm by observing the performance (the flow cover ratio and its variance) over time. We randomly generate 2,000 flows in each period (*e.g.*, 10s) according to the 2-8 distribution. As depicted in Fig. 15, the SCK algorithm can improve the average flow cover ratio at least 44% compared with the RND algorithm. Meanwhile, we observe that the FCR's variance of the SCK algorithm is only 1/2 times as that of the RND algorithm by Fig. 16.

VI. Conclusion

In this paper, we studied how to perform optimal sketch configuration on switches for proportional fairness. We proposed the SCP problem, and designed an efficient algorithm with approximation ratio $1/3$ for this problem. We implemented the proposed algorithm on our SDN platform, and the simulation results showed high efficiency of our proposed algorithm. In the future, we will study the trade-off between sketch's memory cost, measurement accuracy and computing cost for more practical designs.

References

[1] 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.

[2] D. Li, Y. Shang, and C. Chen, "Software defined green data center network with exclusive routing," in *IEEE INFOCOM*, 2014, pp. 1743–1751.

[3] B. Wang, Y. Zheng, W. Lou, and Y. T. Hou, "Ddos attack protection in the era of cloud computing and software-defined networking," *Computer Networks*, vol. 81, pp. 308–319, 2015.

[4] Q. Yan, F. R. Yu, Q. Gong, and J. Li, "Software-defined networking (sdn) and distributed denial of service (ddos) attacks in cloud computing environments: A survey, some research issues, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 602–622, 2016.

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

[6] Z. Su, T. Wang, Y. Xia, and M. Hamdi, "Flowcover: Low-cost flow monitoring scheme in software defined networks," in *Global Communications Conference (GLOBECOM)*. IEEE, 2014, pp. 1956–1961.

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

[8] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 245–256, 2004.

[9] P. Phaal and M. Lavine, "sflow version 5," *URL: http://www. sflow. org/sflow_version_5. txt, J uli*, 2004.

[10] A. Kumar, J. Xu, and J. Wang, "Space-code bloom filter for efficient per-flow traffic measurement," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, pp. 2327–2339, 2006.

[11] T. Li, S. Chen, and Y. Ling, "Per-flow traffic measurement through randomized counter sharing," *Networking, IEEE/ACM Transactions on*, vol. 20, no. 5, pp. 1622–1634, 2012.

[12] 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.

[13] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *the 10th USENIX Symposium on Networked Systems Design and Implementation*, 2013, pp. 29–42.

[14] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 113–126.

[15] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[16] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: methods, evaluation, and applications," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. ACM, 2003, pp. 234–247.

[17] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[18] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," 2018.

[19] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoretical Computer Science*, vol. 312, no. 1, pp. 3–15, 2004.

[20] G. Cormode and S. Muthukrishnan, "What's new: Finding significant differences in network data streams," *IEEE/ACM Transactions on Networking*, vol. 13, no. 6, pp. 1219–1232, 2005.

[21] 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.

[22] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "csamp: A system for network-wide flow monitoring." in *NSDI*, vol. 8, 2008, pp. 233–246.

[23] Y. Yu, C. Qian, and X. Li, "Distributed and collaborative traffic monitoring in software defined networks," in *Proceedings of the third workshop on HotSDN*. ACM, 2014, pp. 85–90.

[24] "Open vswitch," http://openvswitch.org/.

[25] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 101–114.

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

[27] L. Li, M. Pal, and Y. R. Yang, "Proportional fairness in multi-rate wireless lans," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008, pp. 1004–1012.

[28] W. Li, S. Wang, Y. Cui, X. Cheng, R. Xin, M. A. Al-Rodhaan, and A. Al-Dhelaan, "Ap association for proportional fairness in multirate wlans," *IEEE/ACM Transactions on Networking (TON)*, vol. 22, no. 1, pp. 191–202, 2014.

[29] A. Gupta, "Approximations algorithms," 2005.

[30] S. Khuller, A. Moss, and J. S. Naor, "The budgeted maximum coverage problem," *Information Processing Letters*, vol. 70, no. 1, pp. 39–45, 1999.

[31] C. Chekuri and A. Kumar, "Maximum coverage problem with group budget constraints and applications," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2004, pp. 72–83.

[32] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, and M. Handley, "Data center networking with multipath tcp," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 10.

[33] U. Feige, "A threshold of ln n for approximating set cover," *Journal of the ACM (JACM)*, vol. 45, no. 4, pp. 634–652, 1998.

[34] "The network topology from the monash university," http://www.ecse.monash.edu.au/twiki/bin/view/InFocus/LargePacket-switchingNetworkTopologies.

[35] 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.

[36] S. Ryu, "Framework community: Ryu sdn controller," 2016.

[37] "Dpdk," http://dpdk.org/.