An incrementally-deployable P4-enabled architecture for network-wide heavy-hitter detection

Damu Ding, *Student Member, IEEE*, Marco Savi, Gianni Antichi, and Domenico Siracusa

***Abstract*—The advent of Software-Defined Networking with OpenFlow first, and subsequently the emergence of programmable data planes, has boosted lots of research around many networking aspects: monitoring, security, traffic engineering. In the context of monitoring, most of the proposed solutions show the benefits of data plane programmability by simplifying the network complexity with a one big-switch abstraction. Only few papers look at network-wide solutions, but consider the network only composed by programmable devices. In this paper, we argue that the primary challenge for a successful adoption of those solutions is the deployment problem: how to compose and monitor a network consisting of both legacy and programmable switches? We propose an approach for incrementally deploy programmable devices in an ISP network with the goal of monitoring as many distinct network flows as possible. While assessing the benefits of our solution, we realized that proposed network-wide monitoring algorithms might not be optimized for a partial deployment scenario. We then also developed and implemented in P4 a novel strategy capable of detecting network-wide heavy flows: results show that it can achieve better accuracy than state-of-the-art solutions while relying on less information from the data plane and leading to only marginal additional packet processing time.**

***Index Terms*—Network monitoring, Programmable data plane, Incremental deployment, Heavy-hitter detection**

I. Introduction

Network monitoring is of primary importance: it is the main enabler of various network management tasks, ranging from accounting [2][3], traffic engineering [4][5], anomaly detection [6], Distributed Denial-of-Service (DDoS) detection [7], Super-spreader detection [8], and scans detection [9]. With the advent of Software-Defined Networking (SDN), the significance of network monitoring has been certainly boosted. This is because, SDN, with the idea of a (logically) centralized control, allows an easy coupling of network management operations with the observed network status. As a result, SDN has been seen as the answer to many of the limitations of legacy network infrastructures [10][11][12]. However, such a noble intent has been limited by its current predominant realisation, the OpenFlow (OF) protocol. Indeed, current OpenFlow APIs are ill-suited and cannot provide accurate data-plane measurements: the main mechanism exposes the per-port and per-flow counters available in the switches [13]. An application running on top of the controller can periodically poll each counter using the standard OF APIs and then react accordingly, instantiating the appropriate rule changes. As a consequence, OF suffers from two important limitations: *(i)* the controller needs to know in advance which flows have to be monitored in the data plane and *(ii)* as the data plane exposes just simple counters, the controller needs to do all the processing to determine the network state. This also implies that OF-enabled devices are only able to collect raw flow statistics to be sent to a monitoring collector, causing significant communication overhead for monitoring purposes. This limitation is well-know for legacy devices as well (e.g. SNMP- and NetFlow-supported equipment) [14].

Lately, the advent of the so-called *programmable switches* (e.g. P4-enabled switches [15]) has introduced the possibility to program data plane with advanced functionality and enabled the possibility to implement more refined monitoring solutions directly in the switch hardware. Such an innovative technology has attracted a growing number of researchers and practitioners that in turn have proposed many different solutions to enhance SDN capabilities in the context of network monitoring [16][17][18][19][20]. As a result, the prospect of realizing fine-grained network-wide monitoring, by analyzing the exposed information from all the switches in a network, has attracted a lot of interest [18][21][22]. For instance, memory-efficient data structures, such as *sketches* [23][24], have been proven to be implementable in programmable switches to reduce redundant monitoring information. However, in practice, a one-shot replacement of all the existing legacy devices with programmable switches is not a feasible solution due to operational and budget burdens. Clearly, this limits the benefits in terms of network flow monitoring performance, since a partial deployment leads to a reduced flow visibility.

This paper proposes a novel approach for an *incremental deployment* of programmable switches in Internet Service Provider (ISP) networks. To optimize network-wide monitoring practices, it is important to have visibility over the largest number of distinct flows. To this end, we exploit the HyperLogLog algorithm [25] that is generally used for the count-distinct problem, approximating the number of distinct elements in a multi-set. While assessing the benefits of our solution, we shortly realized that state-of-the-art network-wide monitoring algorithms might not perform optimally in a partial deployment scenario. We therefore propose a new algorithm that is capable of detecting *network-wide* heavy flows (i.e., *heavy hitters*) using as input only partial information from the data plane. We evaluate our incremental deployment strategy alongside the proposed heavy-hitter detection algorithm in simulation. We also implemented our heavy-hitter detection

Damu Ding is with Fondazione Bruno Kessler, Trento, Italy and University of Bologna, Bologna, Italy. Marco Savi and Domenico Siracusa are with Fondazione Bruno Kessler, Trento, Italy. Gianni Antichi is with Queen Mary University of London, London, United Kingdom.

A preliminary version of this paper appeared in [1], presented at the IEEE Conference on Network Softwarization (NetSoft), Paris, France, in 2019

©2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

strategy in P4 [15] and tested it in an emulated environment. By comparing our incremental deployment solution with state-of-the-art proposals, results show that we can achieve a better monitoring accuracy using less switches. Moreover, our heavy-hitter detection strategy outperforms existing ones in terms of F1 score while relying on less information from the data plane, while leading to only slightly higher packet processing times in executing the programmable switch pipeline.

The main contributions of the paper are as follows:

- We tackle the problem of partial deployment of programmable networks in the context of network monitoring. We propose a new strategy that allows to incrementally deploy new programmable switches and simultaneously maximize the network monitoring operation.
- We propose a new strategy for network-wide heavy-hitter detection which is robust to partial deployments. We implemented a prototype of such a strategy in P4 language [15] and tested it.

The remainder of the paper is organized as follows. In Section II we explain the best practices for an effective partial deployment of programmable switches, while Section III presents the algorithmic background. Section IV describes our incremental deployment algorithm, and Section V presents our network-wide heavy-hitter detection strategy and its implementation in P4 language. Section VI and Section VII report our evaluation results and comparison with the state of the art. Finally, Section VIII recalls the related work and Section IX concludes this paper.

II. Hints for improved monitoring performance with limited flow visibility

When only a limited number of programmable switches can be deployed in an ISP network, the network operator must ensure that they are deployed in such a way it is made the best use of them in terms of monitoring performance, measured as we will explain later by F1 score. Fig. 1 shows the results of a simple test: we simulated an ISP topology of 100 nodes [26] with real traffic, and we evaluated the F1 score of an existing threshold-based network-wide heavy-hitter detection strategy, proposed by Harrison et al. [22] (see Section VI for more details on simulation settings and evaluated metrics) when only one legacy switch/router[1] is replaced with a programmable switch. The graph shows all the 100 possible deployments. What we can see is that the F1 score *(i)* is in all cases low but *(ii)* substantially varies depending on the placement position of the programmable switch. Consideration *(i)* comes from the fact that by replacing only one switch the flow visibility is very low, since only heavy hitters crossing such switch can be detected, while *(ii)* proves that how we deploy programmable switches in the network is a fundamental aspect to ensure good monitoring performance with limited flow visibility.

Our intuition is that, when deploying a single programmable switch, *an effective strategy is to replace the one crossed by the highest number of distinct flows*. This is because the highest

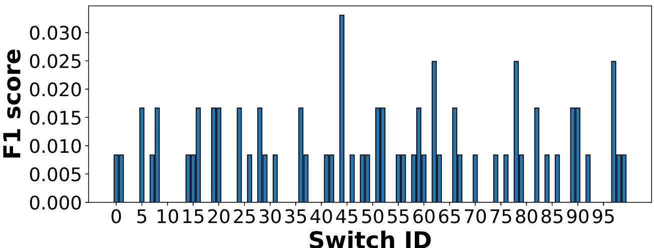


Fig. 1. F1 score of Harrison's heavy-hitter detection strategy with single programmable-switch deployment

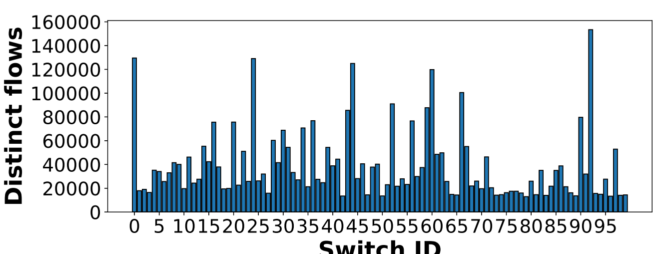


Fig. 2. Number of distinct flows crossing each switch

the number of monitored flows is, the highest (on average) the chance of monitoring some heavy hitters is. This consideration holds also for any other flow-based network-wide monitoring algorithm (e.g. heavy changes detection [17], network traffic entropy estimation [16], etc.), whose analysis is however out of the scope of this paper. Fig. 2 shows the number of distinct flows crossing each one of the switches in the given time interval. Two observations can be made: *(i)* if a switch crossed by a few number of distinct flows is replaced, it is highly probable that it cannot detect any heavy hitters (i.e., F1 score is often zero); *(ii)* a (weak) correlation between F1 score and number of monitored distinct flows indeed exists. For example, replacing the switch with ID 44 leads to the highest F1 score, and the same switch is one of the switches crossed by the highest number of distinct flows. However, the network-wide heavy-hitter detection strategy proposed in [22] has not been explicitly designed to best exploit the available information on switches' monitored distinct flows, unlike our proposed strategy (see Section V), so correlation between F1 score and number of distinct flows is small.

The same considerations can be made when more than one programmable switches have to be deployed in the network. In this case, it must be ensured that *the subset of programmable switches to be deployed monitors the highest number of distinct flows overall* (i.e., neglecting duplications): this guarantees satisfactory performance in the execution of monitoring tasks such as heavy-hitter detection. The considerations and intuitions discussed in this section have thus guided us in the design of our algorithm for incremental deployment of programmable switches, shown in Section IV.

III. Background

In this section, we present the background for our incremental deployment algorithm and network-wide heavy-

[1] In the remainder of this paper, we will use the generic term *legacy device* to generically refer to legacy switches or routers. In fact, programmable switches can support both Layer-2 and Layer-3 functionalities.

hitter detection strategy. Our algorithms rely on different data streaming concepts and methods.

A. Estimated count of distinct flows

An efficient and effective method to count a number of distinct items from a set is HyperLogLog [25]. In our specific case, given a packet stream $S = \{a_1, a_2, ..., a_m\}$, where each packet is characterized by a specific `(srcIP,dstIP)` pair (generically called *flow key*), it returns an estimation of cardinality of flows, i.e., how many `(srcIP,dstIP)` distinct pairs exist in the stream[2]. In this paper, we use $\hat{n} \leftarrow Hll(S)$ as notation to indicate input and output of the HyperLogLog algorithm: Hll indicates the algorithm, S the input packet stream and $\hat{n}$ the cardinality of flows (i.e., number of distinct flows). The relative error of HyperLogLog is only $\frac{1.04}{\sqrt{m}}$, where m is the size of HyperLogLog register. Apart from its high accuracy, HyperLogLog is also very fast since the query time complexity is O(1). Moreover, calculating the *union* (or merge) of two or more HyperLogLog data structures (also called *sketches*) is also very efficient, and can be used to count the number of distinct flows in the union of two (or more) data streams, e.g. S_a and S_b. In our notation, this can be written as $\hat{n}_{union} \leftarrow Hll(S_a \cup S_b)$, where $\hat{n}_{union}$ is the number of distinct flows of the packet streams union $S_a \cup S_b$.

B. Estimation of flow packet count for heavy-hitter detection

Estimating the number of packets for a specific flow is fundamental for a proper detection of heavy hitters. Different algorithms exist to perform such an estimation: we choose to use Count-Min Sketch [24], which relies on a probabilistic data structure (i.e., sketch) based on pairwise-independent hash functions. Count-Min Sketch envisions two types of operations: *Update* and *Query*. Update operation is responsible for continuously updating the sketch with information on incoming packets in the switch, whereas Query operation is used to retrieve the estimated number of incoming packets for a specific flow. To formalize the problem, we consider a stream of packets $S = \{a_1, a_2, ..., a_m\}$. The Count-Min Sketch algorithm returns an estimator of packet count $\hat{f}_x$ of flow $x \sim (\texttt{srcIP}_x, \texttt{dstIP}_x)$ satisfying the following condition: $Pr[|\hat{f}_x - f_x| > \varepsilon|S|] \leq \delta$, where ε $(0 \leq \varepsilon \leq 1)$ is the relative biased value and δ $(0 \leq \delta \leq 1)$ is the error probability. In Count-Min Sketch, the space complexity is $O(\varepsilon^{-1} log_2(\delta^{-1}))$, and per-update time is $O(log_2(\delta^{-1}))$ [27]. Additionally, the estimation of packet count satisfies $f_x \leq \hat{f}_x \leq f_x + \varepsilon|S|$, where f_x is the real packet count value. The accuracy of Count-Min Sketch depends on ε and δ, which can be tuned by respectively defining *(i)* the output size N_s of each hash function and *(ii)* the number N_h of hash functions of the data structure.

In previous definitions, a heavy hitter is a flow whose packet count overcomes a threshold $\vartheta|S|$ $(0 < \vartheta < 1)$. If a Count-Min Sketch is adopted, the probability to erroneously detect a heavy hitter due to packet miscount is defined in the following way: $Pr[\exists x | \hat{f}_x \geq (\vartheta + \varepsilon)|S|] \leq \delta$. If N_h is large enough, error probability δ is negligible.

[2]Note that in this paper, without any loss of generality, we consider source/destination pairs as flow identifiers. However, other definitions could also be adopted (e.g. 5-tuple).

IV. An algorithm for incremental deployment of programmable switches

In this section we propose a novel algorithm for the incremental upgrade of a legacy infrastructure with programmable switches, which aims at ensuring good network monitoring performance, as discussed in Section II.

A. Problem definition

Our problem of incremental deployment of programmable switches can be formalized in the following way.
Given:

- An ISP network topology of a legacy network infrastructure $\mathcal{G} = (\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ is the set of legacy devices and $\mathcal{L}$ the set of interconnection links;
- A long-term estimation of the transmitted packets in the network between different sources and destinations (i.e., traffic matrix T), including their routing paths and possible re-routing paths in case of failures. From this information it is possible to retrieve the estimated packet stream T_i for each switch $i \in \mathcal{N}$. Note that, unlike in intra data-center scenarios, traffic distribution in ISP networks is much less dynamic, being routes prevalence and persistence increasing over time [28]. For this reason, it is possible to estimate a reasonable traffic matrix T using historical data;
- A number $P \leq |\mathcal{N}|$ of legacy devices to be replaced with programmable switches;

Replace a subset of $\mathcal{P}$ (such that $P = |\mathcal{P}|$) of legacy devices with programmable switches with the goal of monitoring the highest number of distinct flows in the network and in an *incremental way*. This means that it must be assured that any subset of programmable switches $\mathcal{Z}$ (with $|\mathcal{Z}| \leq P$) that have been already deployed in the network as intermediate step, monitors the highest number of distinct flows as well.

B. Incremental deployment algorithm

As we mentioned above, HyperLogLog has good performance on estimating the distinct flows from an union of packet streams, so we use it to estimate the number of distinct flows passing through a set of legacy devices [29]. The pseudo code of our proposed algorithm is shown in Algorithm 1. To place the first programmable switch, we compute the estimated number of distinct flows $\hat{n}$ carried by each of the $i \in \mathcal{N}$ legacy devices using the HyperLogLog algorithm. The input of HyperLogLog, for each device $i \in \mathcal{N}$, is T_i. For the replacement with a programmable switch, the algorithm selects the legacy device crossed by the highest number (max) of distinct flows (Lines 4-7). Such legacy device is added to $\mathcal{P}$. Once the first legacy device has been replaced, the principle to replace any other legacy device consists in progressively finding the one that, if replaced, allows to overall monitor the highest number of distinct flows in the network. The choice must be carefully taken, since different devices may be crossed by the same flows, and thus some flow information can be duplicated. To do so, we exploit the *union* property of HyperLogLog (Line 14), recalled in Section III. As shown in

Algorithm 1: Incremental deployment algorithm

Input: Long-term traffic statistics T, Network topology $\mathcal{G}$, Number of legacy devices P to be replaced

Output: Set of legacy devices $\mathcal{P}$ to be replaced

1 $max \leftarrow 0$, $\mathcal{P} \leftarrow \{\}$, $\hat{n} \leftarrow 0$;
2 $\hat{n}^{pre} \leftarrow 0$, $T^{pre} \leftarrow \{\}$, $key \leftarrow$ empty;
3 **for** *Each legacy device* $i \in \mathcal{N}$ *carrying traffic* T_i **do**
4 $\quad \hat{n} \leftarrow Hll\ (T_i)$;
5 $\quad$ **if** $\hat{n} > max$ **then**
6 $\quad\quad max \leftarrow \hat{n}$
7 $\quad\quad key \leftarrow i$
8 $\mathcal{P}.add(key)$
9 **if** $P > 1$ **then**
10 $\quad \hat{n}^{pre} \leftarrow max$
11 $\quad T^{pre} \leftarrow T_{key}$
12 $\quad$ **while** $\mathcal{P}.size() \leq P$ **do**
13 $\quad\quad$ **for** *Each switch* $i \in \mathcal{N} \setminus \mathcal{P}$ *carrying traffic* T_i **do**
14 $\quad\quad\quad \hat{n} \leftarrow Hll(T^{pre} \cup T_i)$
15 $\quad\quad\quad$ **if** $\hat{n} > max$ **then**
16 $\quad\quad\quad\quad max \leftarrow \hat{n}$
17 $\quad\quad\quad\quad key \leftarrow i$
18 $\quad\quad \mathcal{P}.add(key)$
19 $\quad\quad \hat{n}^{pre} \leftarrow max$
20 $\quad\quad T^{pre} \leftarrow T^{pre} \cup T_{key}$
21 **return** $\mathcal{P}$

Lines 10-21, the algorithm estimates the number of monitored distinct flows $\hat{n}^{pre}$ coming from the union of packet streams *(i)* of all the previously-upgraded programmable switches (T^{pre}) and *(ii)* of any legacy device $i \in \mathcal{N} \setminus \mathcal{P}$ still in the network (T_i). Then, the algorithm selects for replacement (and thus addition to set $\mathcal{P}$) the legacy device i leading to the largest number of monitored distinct flows overall (max). This operation is iterated until a number $P = |\mathcal{P}|$ of legacy switches has been replaced. Once the set $\mathcal{P}$ has been defined, the network operator can proceed with the physical replacement of the legacy devices with programmable switches, while ensuring interoperability in a hybrid environment [11]. Note that our incremental deployment algorithm focuses on the replacement of switches instead of new additions to the network. In fact, adding new switches may imply changes to routing and flow statistics alteration, making our algorithm ineffective.

V. A NETWORK-WIDE HEAVY-HITTER DETECTION STRATEGY ROBUST TO PARTIAL DEPLOYMENT

In this section we propose a novel network-wide heavy-hitter detection strategy. While taking inspiration from an existing approach [22], it differs from the state of the art by introducing the concept of *local* and *global sample lists*, which make it robust to partial deployments aiming at maximizing the network flow visibility.

Figure 3 shows the interaction between switches and a centralized controller for network-wide heavy-hitter detection,

Fig. 3. Interaction between controller and programmable switches for network-wide heavy-hitter detection in a partial deployment scenario

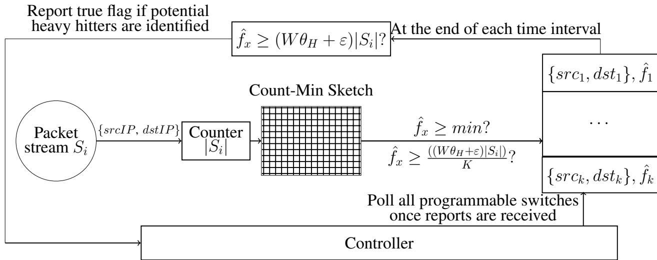


Fig. 4. Scheme of the proposed network-wide heavy-hitter detection strategy

in the case of partial deployment of programmable switches (i.e., when only some switches can perform monitoring operations in the data plane). Time is divided in intervals, and in every time interval each programmable switch dynamically stores in a *sample list* only those flows whose packet count is larger than a dynamic *sampling threshold*. At the end of each time interval, if any programmable switch stores in its sample list one or more flows with packet count larger than a dynamic *local threshold*, it reports a *true flag* to the centralized controller. The flows whose packet count overcomes the local threshold are called *potential heavy hitters*. The controller, if at least one true-flag report is received, then polls all the programmable switches in the network to gather the {flow key, packet count} pairs of all the flows stored in their sample lists. This information is used to estimate the whole *network volume*, i.e., the number of all the unique packets transmitted in the network in the given time interval. Finally, the controller computes the so-called *global threshold* leveraging the estimated network volume and gets all the network-wide heavy hitters, i.e., the flows from sample lists whose packet count is larger than the global threshold. Finally, all the flow and packet statistics are reset and a new time interval is started.

In the following subsections, we formalize the problem of network-wide heavy-hitter detection and describe in detail the algorithms running both in the programmable switches and in the controller to implement the proposed high-level strategy.

A. Problem definition

We formulate the network-wide heavy-hitter detection problem as follows.

Given:

Algorithm 2: Network-wide heavy-hitter detection algorithm - Programmable switch $i \in \mathcal{P}$ (Data plane)

Input: Flow stream S, Local minimum min, Heavy-hitters identification fraction θ_H, Local ratio W, Sampling rate K, Count-Min Sketch size $N_h \times N_s$, Time interval T_{int}
Output: $flag$ (`true` if potential heavy hitters are identified in T_{int}, `false` otherwise)

1 $\varepsilon \leftarrow 1/N_s$
2 **Function** *StoreFlowsInSampleList*:
3 $|S_i| \leftarrow 0$
4 $flag \leftarrow$ `false`
5 **while** *currentTime* $\in T_{int}$ **do**
6 **for** *Each packet belonging to flow* $x \sim (srcIP_x, dstIP_x)$ *received* **do**
7 $|S_i| \leftarrow |S_i| + 1$
8 **if** $\hat{f}_x \geq min$ *and* $\hat{f}_x \geq \frac{(W\theta_H+\varepsilon)|S_i|}{K}$ **then**
9 $SampleList_i(x) \leftarrow \hat{f}_x$
10 **Function** *PotentialHHsDetection*:
11 **if** *currentTime* $=$ *End time of* T_{int} **then**
12 **for** *Each flow* x *in* $SampleList_i$ **do**
13 **if** $\hat{f}_x < \frac{(W\theta_H+\varepsilon)|S_i|}{K}$ **then**
14 $SampleList_i.remove(x)$
15 **if** $SampleList(x) \geq (W\theta_H + \varepsilon)|S_i|$ **then**
16 $flag \leftarrow$ `true`
17 **return** $flag$

- A heavy-hitter identification fraction θ_H $(0 < \theta_H < 1)$;
- A time interval T_{int};
- The set of unique packets S transmitted in the network;

Identify the set of flows which are network-wide heavy hitters (HH), i.e., carry in the time interval T_{int} a number of packets larger than the global threshold $\theta_H|S|_{tot}$, where $|S|_{tot}$ is the *network volume* (i.e., number of transmitted packets).

B. Algorithm in programmable switches (Data plane)

As shown in Fig. 4, when a packet comes into a programmable switch i, a packet counter named $|S_i|$ is updated to count all the incoming packets. A Count-Min Sketch data structure, which is used to store the estimated packet counts for all the flows, is updated to include the information from the current packet, and then it is queried to retrieve the estimated packet count $\hat{f}_x$ for the flow $x \sim$ (srcIP$_x$, dstIP$_x$) such packet belongs to. This information is used to understand whether the packet belongs to a flow that must be inserted in the sample list.

The flow x is inserted in the sample list if $\hat{f}_x \geq \frac{(W\theta_H+\varepsilon)|S_i|}{K}$, where $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$ is the sampling threshold. The parameters W $(W \geq 1)$ and K $(K \geq 1)$ affect the size of the sample list: the larger W and the smaller K are, the smaller the sample list size is, thereby consuming less memory in the switch. However, as we will report in the next subsection, this would reduce the accuracy on the estimation of the overall network volume and on the identification of heavy hitters. Therefore, K (called *sampling rate*) should be carefully set in each programmable switch to store only flows carrying a significant number of packets. ε is instead the biased value caused by Count-Min Sketch (see Section III): we sum $\varepsilon = 1/N_s$ to $W\theta_H$ in order to compensate such bias. The sample list is thus used to dynamically store the packet counts for the most frequent flows crossing the switch.

Since at the beginning of each time interval T_{int} the sampling threshold is low, being $|S_i|$ a small value (that can even be < 1), flows with small packet counts would be stored in the sample list.

We thus introduce a parameter min operating in conjunction with the sampling threshold: only if packet count of the considered flow is larger than both min and sampling threshold $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$, the flow is inserted in the sample list. This is described in Lines 6-11 of Algorithm 2.

At the end of the time interval T_{int}, $|S_i|$ counts all the incoming packets in the considered time frame. Thus, as shown in Lines 13-14, the algorithm removes from the sample list all those flows with packet count lower than $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$, where $|S_i|$ is the final stored value. This means that the algorithm keeps in the sample list only those flows which have packet counts larger than the final sampling threshold, while discarding the flows with packet counts greater than the temporary threshold dynamically computed and updated within the time interval. Note that the sample list stores at most $\frac{K}{(W\theta_H+\varepsilon)}$ flows, and thus its memory occupation increases as K increase or W decrease, as already discussed.

Finally, the algorithm evaluates whether potential heavy hitters cross the switch. They are the flows whose packet counts are greater than the switch local threshold, set as $(W\theta_H + \varepsilon)|S_i|$. A true flag is sent to the controller if at least one potential heavy hitter is detected, otherwise no information is sent (Lines 15-17). Note that local threshold and sampling threshold are similar: with respect to sampling threshold, local threshold just misses in its definition K, which has been introduced to set the size of the sample list. The primary role of W is instead to set the proportion (or *ratio*) between the local threshold $(W\theta_H + \varepsilon)|S_i|$ and the global threshold $\theta_H|S|_{tot}$, being S_i a local (and smaller) value than $|S|_{tot}$.

C. Algorithm in centralized controller (Control plane)

At the end of the time interval T_{int}, once the controller receives from programmable switches any report including a true flag, it polls all of them to obtain their sample lists. Note that different sample lists can include the estimated packet count for the same flows: this happens if a flow crosses multiple programmable switches. To avoid an overestimation of $|S|_{tot}$, Algorithm 3 makes sure that *(i)* only the minimum-estimated packet count is kept, i.e., the one less overestimated by Count-Min Sketch, and *(ii)* it is stored in a list (i.e., $GlobalSampleList$) including all distinct flows from sample lists (Lines 2-8). The algorithm then sums up the packet counts for all the identified distinct flows and estimates the whole network volume $|S|_{tot}$ (Lines 9-12). If packet counts

Algorithm 3: Network-wide heavy-hitter detection algorithm - Centralized controller (Control plane)

Input: Heavy-hitter identification fraction θ_H, Time interval T_{int}, Sample lists $SampleList_i$ from all programmable switches $i \in \mathcal{P}$
Output: Set of network-wide heavy hitters HH in T_{int}

```
1  Function RetrieveDistinctFlowsPacketCounts:
2      for Each switch i ∈ P do
3          for Each flow x in SampleList_i do
4              if flow x is in GlobalSampleList then
5                  if SampleList_i(x) < GlobalSampleList(x) then
6                      GlobalSampleList(x) ← SampleList_i(x)
7              else
8                  GlobalSampleList(x) ← SampleList_i(x)
9  Function EstimateVolume:
10     |S|_tot ← 0
11     for Each flow x in GlobalSampleList do
12         |S|_tot ← |S|_tot + GlobalSampleList(x)
13 Function GetNetworkWideHH:
14     for Each flow x in GlobalSampleList do
15         if GlobalSampleList(x) ≥ θ_H |S|_tot then
16             HH.add(x)
17     return HH
```

of flows belonging to *GlobalSampleList* (which surely includes the potential heavy hitters) are larger than the global threshold $\theta_H|S|_{tot}$, we consider them as network-wide heavy hitters HH (Lines 13-17). At last, the controller triggers the reset of counters in all programmable switches.

D. Implementation in P4 language

1) Programmable switches (Data plane): We have implemented our prototype of algorithm in P4, leveraging the P4 behavioral model [30] to describe the behavior of P4 switches (e.g., parsers, tables, actions, ingress and egress in the P4 pipeline). The source code is available at [31]. In P4-enabled switches, registers are stateful memories whose values can be read and written [32]. We first allocate a one-sized register S_i to count the overall number of packets in the given time interval T_{int}. When each packet arrives at the switch, the value in this register is incremented by one.

Moreover, we allocated several one-dimensional registers for Count-Min Sketch implementation. We chose *xxhash* [33] to implement the needed pairwise-independent hash functions by varying the *seed* of each hash function, which is associated to a different register. In our case, the seed is set to be the same value as the index of each register. In the Count-Min Sketch *Update* operation, each register relies on its corresponding hash function, which takes as input $x \sim$`(srcIP,dstIP)`, to obtain as output the index of the register cell whose value must be incremented by one.

To perform the *Query* operation on the Count-Min Sketch, we set a *count-min* variable to the queried value obtained from the first register, which is associated to the first hash function. Consequently, the queried value for the remaining registers is compared with *count-min*. If the queried value for a register is smaller than current *count-min* value, then *count-min* is updated accordingly. Its final value is thus the packet count estimation *count-min* $= \hat{f}_x$ for the queried flow x.

One of the drawbacks of P4 language is that it does not allow variable-sized lists, as *sample list* is. We thus used three same-sized registers, named *samplelist_src*, *samplelist_dst* and *samplelist_count* to implement the sample list and to store *srcIP*, *dstIP* and *packet count* of flows, respectively. Information on flow keys `(srcIP,dstIP)` associated to significant packet counts is stored in these registers. Since P4 language does not support *for* loops to find the flows with smallest packet count in the sample list to be replaced, the hash function CRC32 is used to decide whether to replace list's entries.

When the stored *count-min* value is larger than both the preset minimum value *min* and sampling threshold $\frac{(W\theta_H+\epsilon)|S_i|}{K}$ (where the value of S_i is read from register S_i), the algorithm hashes the flow key `(srcIP,dstIP)` using CRC32 hash function and checks whether the sample list register cells indexed by the output value of CRC32 are empty or not. If they are empty, *srcIP*, *dstIP* and *packet count* are added to sample list registers in the same indexed position. Otherwise, the switch compares the current estimated packet count $\hat{f}_x$ with stored packet count $\hat{f}_y$ (from the generic stored flow y) in *samplelist_count*. If the current packet count $\hat{f}_x$ is larger than existing packet count $\hat{f}_y$, the algorithm replaces *srcIP*, *dstIP* and *packet count* to new values from flow x. Additionally, if packet count $\hat{f}_x$ is larger than current largest value *max* stored in the register *count_max*, *max* is replaced by $\hat{f}_x$.

Using the P4 behavioral model, *ingress_global_timestamp* allows to record the timestamp when the switch starts processing the incoming packet. Hence, when *ingress_global_timestamp* is larger than current time interval T_{int} end time, and *max* in the register *count_max* is larger than current local threshold $(W\theta_H+\epsilon)|S_i|$, the switch clones the current packet and embeds a customized one-field header including the field *Flag* with binary value 1. This cloned packet is sent to the controller to report that a potential (and local) heavy hitter exists, while the original packet is forwarded to the expected destination.

2) Centralized controller (Control plane): We implemented a preliminary version of the centralized controller in Python. The controller can read the registers in the switches by using the *simple_switch_CLI* offered by the P4 behavioral model. If the controller receives packets from P4 switches at the end of time interval T_{int}, it gathers the registers *samplelist_src*, *samplelist_dst* and *samplelist_count* from P4 switches and merges them into a global sample list. Finally, according to the heavy-hitter global threshold $\theta_H|S|_{tot}$, the controller is able to detect network-wide heavy hitters. Finally, the controller resets all registers in P4 switches and starts a new round of heavy-hitter detection.

TABLE I
SIMULATION PARAMETERS

HyperLogLog Size m	2^{12}
Time interval T_{int}	5s [16]
Sampling rate K	10
Local ratio W	3
Heavy-hitter identification fraction θ_H	0.05% [16]
Local minimum min	1
Count-Min Sketch size ($N_h \times N_s$)	40×10000

VI. SIMULATION RESULTS

Based on open source implementations of HyperLogLog [29] and Count-Min Sketch [34], we implemented our incremental deployment algorithm and we simulated both our and Harrison's [22] network-wide heavy-hitter detection strategies in Python. In the following the simulation settings are reported.

A. Simulation settings and evaluation metrics

1) Traces and network topology: We divided 50 seconds 2018-passive CAIDA flow trace [35] into 10 time intervals. The programmable switches send reports to the controller when they detect potential heavy hitters at the end of any of those time intervals: in each time interval are transmitted around 2.3 million packets. As testing topology, we adopted a 100-nodes ISP backbone network [26]: adjacency matrix and plotted topology are available in [31]. A 32-bit cyclic redundancy check (CRC) [36] function was used to randomly assign each packet (characterized by a specific `(srcIP,dstIP)` pair) to a source/destination node couple in the network, and each packet is routed on the shortest path.

2) Tuning parameters: Unless otherwise specified, we set the simulation parameters as reported in Table I. For HyperLogLog, we considered $m = 2^{12}$ because it leads to small-enough relative errors for our purposes (see Section III). We then chose T_{int} and θ_H as per [16] and a Count-Min Sketch with size $N_h = 40 \times N_s = 10000$. Considering that each counter in Count-Min Sketch occupies 4 Bytes, the memory to be allocated for the sketch in each switch can be quantified as $10000 \cdot 40 \cdot 4B = 1.6MB$. Since the total memory of a real programmable switch chip (i.e., Barefoot Tofino [37]) is few tens of MB [38], it may be a reasonable size for a real large-scale ISP network scenario. However, in the following we will give a sensitivity analysis of monitoring performance with respect to N_s and N_h, analyzing what happens if more stringent memory requirements arise in the switch. Moreover, we chose $W = 3$ and $K = 10$ after a rigorous sensitivity analysis since, as will be shown later, these values lead, for the considered network topology, to the best trade-off between communication overhead, occupied memory and F1 score (i.e., they allow to overcome state-of-the-art performance for all the considered metrics).

3) Metrics: We set *recall* R and *precision* Pr as key metrics to evaluate our network-wide heavy-hitter detection strategy. They are defined in the following way:

$$R = \frac{Count^{detected/true}_{HeavyHitters}}{Count^{detected/true}_{HeavyHitters} + Count^{undetected/true}_{HeavyHitters}} \quad (1)$$

$$Pr = \frac{Count^{detected/true}_{HeavyHitters}}{Count^{detected/true}_{HeavyHitters} + Count^{detected/false}_{HeavyHitters}} \quad (2)$$

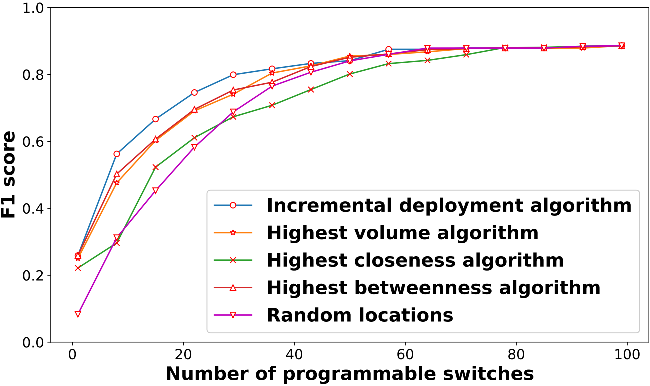


Fig. 5. Performance evaluation of our incremental deployment against some existing algorithms in the detection of network-wide heavy hitters

In our evaluations, we consider *F1 score* ($F1$) as compact metric taking into consideration both precision and recall, and measuring the accuracy of our strategy. It is defined as:

$$F1 = \frac{2 \cdot Pr \cdot R}{Pr + R} \quad (3)$$

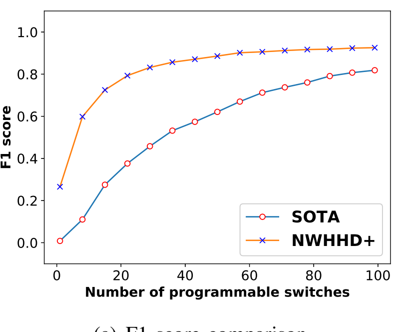
Additionally, we consider each {flow key, packet count} pair as *unit* to evaluate both consumed *communication overhead* (when sent) and overall *occupied memory* (when stored in programmable switches). We consider as *unit* also flags sent by programmable switches to controller for local heavy-hitter notification[3]. All the reported results are the average value obtained in the considered 10 time intervals. We did not include the Count-Min Sketch size in the evaluated occupied memory, since it is constant: Table II gives an insight on Count-Min Sketch memory occupation for some sketch sizes, including $N_h = 40 \times N_s = 10000$.

B. Evaluation of the incremental deployment algorithm

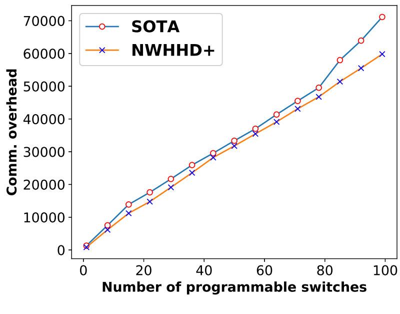
We compare our *Incremental deployment algorithm*, where programmable switches are used for the detection of network-wide heavy hitters, with four existing algorithms: *Highest volume*, *Highest closeness*, *Highest betweenness* and *Random locations*. The Highest volume algorithm exploits long-term flow statistics to replace first the switches overall crossed by the largest number of packets. In the Highest closeness algorithm, the switches are ordered according to decreasing closeness, and in a partial deployment of P programmable switches only the top P switches in the list are replaced [39]. The Highest betweenness algorithm behaves in the same way, but betweenness of nodes [40] is evaluated instead of closeness. Both of these algorithms only depend on the network topology, and their underlying assumption is that nodes with highest centrality should be replaced first. Finally, the Random locations algorithm replaces P randomly-selected nodes: we average results over five randomized instances.

As shown in Fig. 5, which reports F1 score as a function of the number of deployed programmable switches, our Incremental deployment algorithm allows network operators to

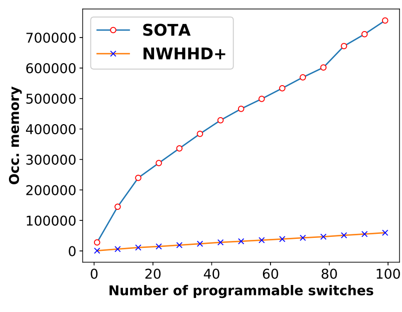
[3] In real scenarios, each flag can be encoded by one bit, while {flow key, packet count} pairs require few bytes to be encoded. However, we consider both of them as a single *unit* for the purpose of easier quantification.



(a) F1 score comparison



(b) Communication overhead comparison



(c) Memory occupation comparison

Fig. 6. Performance comparison of NWHHD+ with a state-of-the-art strategy [22]

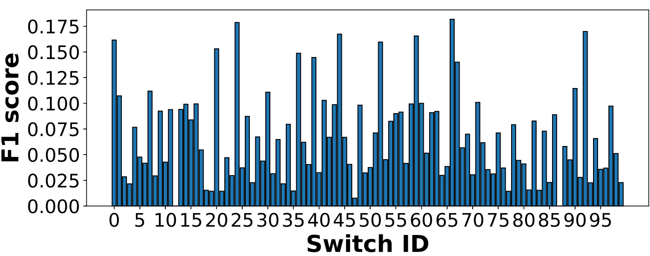


Fig. 7. F1 score of NWHHD+ with single-programmable-switch deployment

deploy a less number of programmable switches while ensuring the same F1 score of the other algorithms. It especially works well when a small number of programmable switches is deployed (i.e., for less than 50 switches), while it has comparable performance as the other algorithms when more than half programmable switches are deployed. This means that our strategy of first replacing switches that monitor the highest number of distinct flows effectively improves flow visibility when it is inherently limited, also with respect to a strategy defined on the same long-term flow statistics (i.e., Highest volume algorithm). In fact, this latter strategy misses to consider that switches could carry many packets belonging to a multitude of small flows, and network-wide heavy hitters may remain undetected.

C. Evaluation of the network-wide heavy-hitter detection strategy in an incremental deployment scenario

We compare the performance of our proposed network-wide heavy-hitter detection strategy, named *NWHHD+* for the sake of brevity, with the state-of-the-art strategy (called *SOTA* in the remainder of the section) proposed by Harrison et al. [22]. In order to fairly compare these two strategies, we set the global threshold for SOTA to $T_g = \overline{d}\theta_H|S|_{tot}$, where $\overline{d}$ is the average path length for the flows in all the time intervals, θ_H is fraction for heavy-hitter identification, and $|S|_{tot}$ is the whole network volume. Smoothing parameter is $\alpha = 0.8$ as per [22].

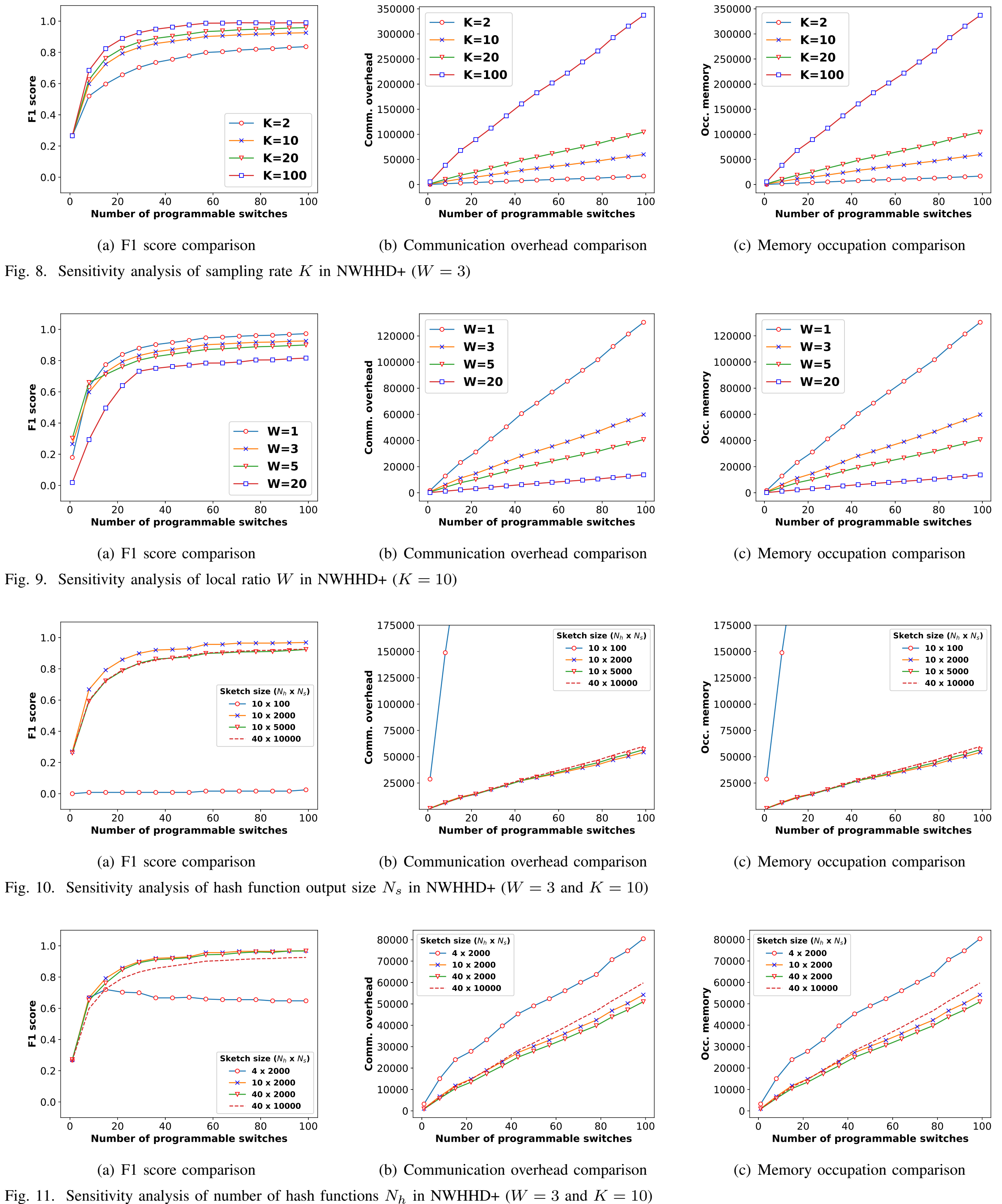
Figure 6(a) shows that NWHHD+, when deploying programmable switches using our incremental deployment algorithm, always leads to higher F1 score than SOTA, especially when the number of programmable switches in the network is small. This means that NWHHD+ better exploits partial flow information provided by the programmable switches to detect the network-wide heavy hitters.

Figure 6(b) shows instead a comparison on the average-generated communication overhead. It clearly shows that NWHHD+ has a comparable communication overhead as SOTA, and NWHHD+ leads to less communication overhead than SOTA as the number of programmable switches increases. In NWHHD+, if at least one local heavy hitter is identified in a given time interval (as always happens in our simulations), at the end of it the controller polls all the programmable switches to estimate the global network volume. Such an approach may cause high communication overhead, but by properly tuning W (as shown later) the local threshold can be increased, significantly reducing communication exchanges between switches and controller. Conversely, the SOTA strategy coarsely estimates the overall network volume at the controller and polls the programmable switches only if the estimated value is above the defined global threshold.

Figure 6(c) shows the average occupied memory by the two strategies. NWHHD+ outperforms SOTA, always occupying much less memory. This happens because NWHHD+ only stores *(i)* the sample list (and not all the {flow key, packet count} pairs, as SOTA does) and *(ii)* one local threshold for all the flows in each programmable switch (while SOTA stores per-flow local thresholds).

Note that unlike SOTA, as can be noticed from all the graphs of this section, NWHHD+ leads to very similar communication overhead and memory occupation results. In fact, communication overhead units in NWHHD+ are equal to {flow key, packet count} pair units stored in the switches sample lists, since such lists from all programmable switches are sent to the controller at the end of any time interval in which at least a local heavy hitter has been detected, plus the number of flags reported by programmable switches to controller to notify the existence of local heavy hitters. This means that the unit difference between communication overhead and memory occupation is at most the number of deployed programmable switches, and this small value cannot be noticed on the graphs.

Additionally, Fig. 7 recalls the simple test described in Section II. What we report in the figure is F1 score for all the possible 100 deployments for the programmable switch when NWHHD+ is adopted. Compared to Fig. 1, we can see that the F1 score is generally higher in NWHHD+ (as



(a) F1 score comparison (b) Communication overhead comparison (c) Memory occupation comparison

Fig. 8. Sensitivity analysis of sampling rate K in NWHHD+ ($W = 3$)

(a) F1 score comparison (b) Communication overhead comparison (c) Memory occupation comparison

Fig. 9. Sensitivity analysis of local ratio W in NWHHD+ ($K = 10$)

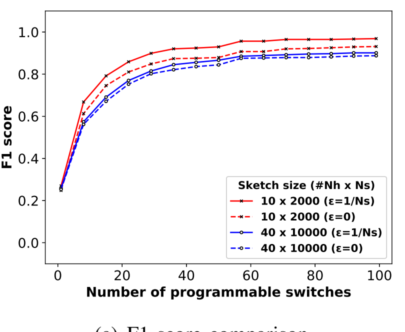
(a) F1 score comparison (b) Communication overhead comparison (c) Memory occupation comparison

Fig. 10. Sensitivity analysis of hash function output size N_s in NWHHD+ ($W = 3$ and $K = 10$)

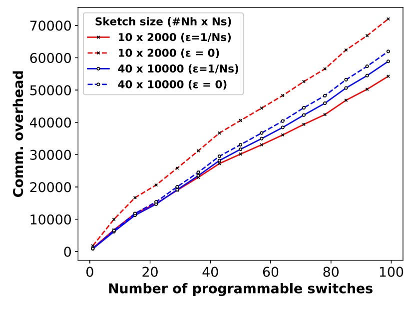
(a) F1 score comparison (b) Communication overhead comparison (c) Memory occupation comparison

Fig. 11. Sensitivity analysis of number of hash functions N_h in NWHHD+ ($W = 3$ and $K = 10$)

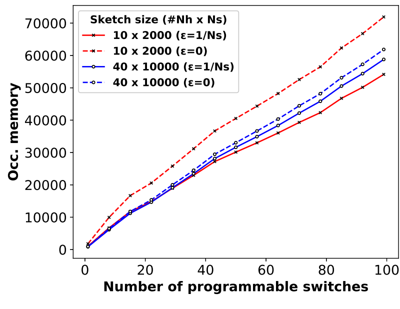
already discussed), and that in most of the cases the peaks in F1 score correspond to IDs of switches that are crossed by a high number of distinct flows (see Fig. 2). This means that NWHHD+ better exploits the distinct flows information than SOTA. This can be even further proven by computing the normalized cross-correlation in $\tau = 0$ [41] between the number of distinct flows and F1 score, which is 13.11 for NWHHD+ and 6.57 for SOTA.



(a) F1 score comparison



(b) Communication overhead comparison



(c) Memory occupation comparison

Fig. 12. Impact of ϵ in NWHHD+ ($W = 3$ and $K = 10$)

TABLE II
COUNT-MIN SKETCH SIZE AND ITS MEMORY OCCUPATION

$N_h \times N_s$	Memory occ.	$N_h \times N_s$	Memory occ.
40×10000	1.6MB	10×2000	80KB
10×100	4KB	10×5000	200KB
4×2000	32KB	40×2000	320KB

In order to show how the performance of NWHHD+ is sensitive to different tuning parameters, we ran simulations by varying the following parameters one at a time, while fixing the others to the values specified in Table I.

1) Sensitivity to sampling rate K: Figure 8 shows the sensitivity analysis of NWHHD+ performance on sampling rate K. As shown in Figure 8(a), when K increases, F1 score increases as well, especially from $K = 10$ to $K = 100$. Figure 8(b) shows that also communication overhead significantly increases as K increases, as well as occupied memory (Figure 8(c)). The reason is that an increase of K leads to a smaller sampling threshold and thus to a bigger sample list to be stored and sent to the controller. In summary, choosing a larger K leads to a performance improvement on heavy-hitter detection, but implies an extra consumption of memory and communication resources. In our case, $K = 10$ is a good trade-off choice since it leads to comparable F1 score than choosing larger K while using much less memory and generating much less communication overhead.

2) Sensitivity to local ratio W: Figure 9 shows the sensitivity analysis of NWHHD+ performance on local ratio W. As shown in Figure 9(a), F1 score decreases as W increases, and the difference of F1 score is substantial between $W = 5$ and $W = 20$. Figure 9(b) and Figure 9(c) show that both communication overhead and occupied memory also significantly decrease when W increases. This happens because, opposite to K, an increase of W leads to a decrease in the sample list size, which means that less {flow key, packet count} pairs are reported to the controller when the switches are polled (i.e., less communication overhead) and less memory is also occupied. As side effect, the detection accuracy is also affected (i.e., lower F1 score). Thus, a smaller W enhances the performance on heavy-hitter detection, but more memory and communication overhead are required. According to the shown results, $W = 3$ leads to a good trade-off among F1 score, communication overhead and memory occupation.

3) Sensitivity to Count-Min Sketch hash function output size N_s: Figure 10 shows the sensitivity analysis of NWHHD+ performance on hash function output size N_s of Count-Min Sketch. For better understanding, we report the memory occupation of various Count-Min Sketches in Table II. The blue cells recall the sketch size adopted in this paper, unless otherwise specified. As shown in Figure 10(a), if the output size is too small (i.e., $N_s = 100$), Count-Min Sketch highly overestimates flows packet count and this causes very poor F1 score. By increasing the value of N_s, F1 score significantly increases until around $N_s = 2000$. However, after this value (i.e., for $N_s = 10000$) F1 score slightly decreases. The reason of this behavior is that $N_s = 2000$ leads to a slight overestimation of packet count for flows stored in sample lists that does not badly affect heavy-hitter detection. Conversely, such slight overestimation compensates missing packet counts of small flows that are not stored in those lists, leading to an estimated whole network volume S_{tot} very close to the real value. This effect makes the controller able to identify network-wide heavy hitters with a more accurate global threshold, thus leading to high F1 score.

Figures 10(b) and 10(c) show that the overestimated packet count due to too small output size also causes very large-sized and badly-estimated sample lists and consequently leads to high and unnecessary occupied memory and communication overhead. While increasing N_s, communication overhead and memory occupation decrease until a minimum is reached at around $N_s = 2000$ then, for larger N_s, they marginally increase. This slight increase is due to the value $\varepsilon = 1/N_s$ used in Algorithm 2 for bias compensation in sampling and local thresholds (see Section V-B). The effectiveness of introducing ε in our strategy is shown in Figure 12: considering ε in the computation of sampling and local thresholds not only increases F1 score, but also decreases communication overhead and memory occupation. On the other hand, considering $\varepsilon = 1/N_s$ also means that N_s directly affects the value of sampling threshold $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$ and local threshold $(W\theta_H + \varepsilon)|S_i|$. If the output size N_s is larger, ε is smaller: this leads to smaller sampling and local thresholds. A smaller sampling threshold generates a larger sample list to be stored and sent to the controller (i.e., higher communication overhead and occupied memory). Conversely, having a smaller N_s leads to less communication overhead and occupied memory, and

TABLE III
SENSITIVITY TO W IN THE CASE OF FULL DEPLOYMENT ($K = 10$)

Evaluated metrics	SOTA	NWHHD+			
		W=1	W=3	W=5	W=20
F1 score	0.821	0.948	0.926	0.881	0.823
Communication overhead	71877	131707	60354	41076	13898
Occupied memory	760042	131608	60255	40977	13799

TABLE IV
SENSITIVITY TO K IN THE CASE OF FULL DEPLOYMENT ($W = 3$)

Evaluated metrics	SOTA	NWHHD+			
		K=2	K=10	K=20	K=100
F1 score	0.821	0.838	0.926	0.959	0.989
Communication overhead	71877	16943	60354	105397	340143
Occupied memory	760042	16844	60255	105298	340044

this explains why lower N_s (but not too small, where overestimation dominates) generates lower communication overhead and requires less memory than sketches with larger output size.

4) Sensitivity to number of Count-Min Sketch hash functions N_h: Figure 11 shows the sensitivity analysis of NWHHD+ performance on number of hash functions N_h of Count-Min Sketch. As shown in Figure 11(a), F1 score increases as the number of hash functions N_h increases. Nevertheless, when N_h is large enough to correctly estimating flow packet counts, further increasing it does not improve heavy-hitter detection performance anymore. Figures 11(b) and 11(c) show that communication overhead and memory occupation decrease as the number of hash function N_h increases. Especially, if N_h is too small, wrongly-estimated flow packet counts lead to high communication overhead and memory consumption. Finally, note that sketch sizes of 10×2000 and 40×2000 both lead to better results than sketch size of 40×10000. This happens because, as shown previously, a too large N_s has bad impact on NWHHD+ performance: results thus show that NWHDD+ is more sensitive to variations to N_s than to N_h.

D. Insights on network-wide heavy-hitter detection in a full deployment scenario

We report an evaluation of NWHHD+ strategy against SOTA also in a *full deployment scenario*, i.e., when all legacy devices have been replaced with programmable switches. As done previously for a partial deployment scenario, Tables III and IV show the sensitivity of NWHHD+ to W and K, and a performance comparison with SOTA. Table III shows that with $W = 20$ NWHHD+ and SOTA have comparable F1 score, but NWHHD+ leads to a significant reduction of both memory occupation and communication overhead. Similar results can be obtained by properly tuning K (Table IV). Also in this case, with $K = 2$, NWHHD+ and SOTA have similar F1 score, but NWHHD+ considerably reduces communication overhead and memory occupation. To summarize, these results show that NWHHD+ is a good strategy also for network-wide heavy-hitter detection in a full deployment scenario.

VII. EVALUATION IN EMULATED P4 ENVIRONMENT

Based on an open source P4 implementation of Count-Min Sketch [42], we implemented both our network-wide heavy-hitter detection (i.e., NWHHD+) and Harrison's (i.e., SOTA) strategies in P4 language and tested them. In the following, we report some details on the emulated network environment.

A. Environment settings and evaluation metrics

1) Emulated network environment: We chose Mininet [43] as our emulated network environment for the deployment of P4 switches implementing the network-wide heavy-hitter strategies. The P4 code is compiled by p4c compiler [44] into a JSON file that describes the behavior of P4 switch (i.e., parser, tables and actions in the P4 pipeline). Then, the JSON file is loaded by any P4 switch created according to the behavioral model [30]. Finally, a topology in Mininet is created connecting such behavior-defined P4 switches. The adopted topology is composed by three interconnected switches, and there is a host connected to each switch. All the packets are forwarded from source host to destination host on the shortest path. We did not consider a larger topology for scalability reasons, since all the P4 switches are emulated on a virtual machine deployed by OpenStack on our testbed with dedicated access to $4 \times$ 2.7GHz CPU cores and to 4GB of RAM. However, given the nature of the performed tests, as we will show later, this does not represent a limitation. The controller is implemented in Python as we explained in Section V-D2.

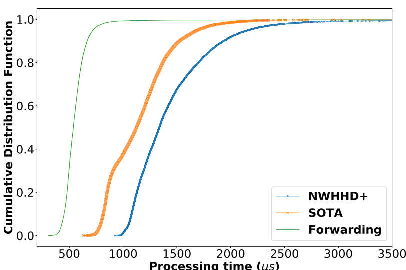
2) Tuning parameters: We used the same settings as our simulations in Python shown in Table I, unless otherwise specified. Additionally, we set the size of register $\mathsf{S_i}$ to 1 and all the remaining registers but Count-Min Sketch (e.g., sample list) to 100. Count-Min Sketch registers size is set to N_s.

3) Metrics: We evaluate NWHHD+ performance in terms of *packet processing time*. We believe that packet processing time is an important metric to evaluate, since it discloses whether the algorithm implemented in the P4 pipeline performs well or not and whether it can be used for line-rate transmissions. We used Wireshark to capture the time when each packet arrives at an input interface of a P4 switch and the time when it is forwarded by an output interface of the same switch. The packet processing time is defined as the difference between such times and estimates the time spent by the packet in the P4 switch pipeline. We also implemented a simple *forwarding* strategy, where no heavy-hitter detection (neither SOTA nor NWHHD+) is performed and the packet is just forwarded to the right output interface.

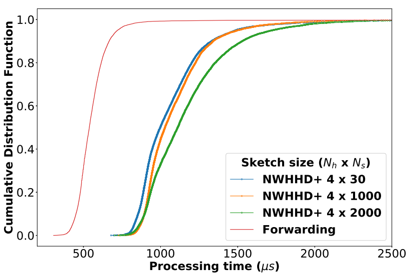
As additional metric, we also evaluate the *controller response time*. This metric represents the time overhead generated in the interaction between data plane and control plane for the identification of heavy hitters.To measure such response time, we captured the following timestamps at the controller: *(i)* the timestamp related to the first true-flag packet that arrives at the controller, meaning that at least one potential heavy-hitter exists in the network and *(ii)* the timestamp when any network-wide heavy hitter has been detected (if it exists). The response time is then defined as the difference between the latter and the former timestamps.

B. Evaluation of packet processing time

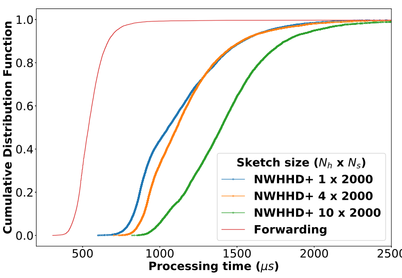
Figure 13 reports the cumulative distribution function of packet processing time measured for 10000 generated packets. As shown in Figure 13(a), both NWHHD+ and SOTA



(a) Comparison with SOTA and simple forwarding



(b) Sensitivity to output size of hash functions N_s



(c) Sensitivity to number of hash functions N_h

Fig. 13. Cumulative distribution function of packet processing time for NWHDD+ (10000 packets)

strategies lead to more packet processing time than simple forwarding, since more operations need to be performed on the packet. However, 50% of the packets can be processed within 1500 μs in the switch when the Count-Min Sketch size is set to $N_h \times N_s = 10 \times 2000$. Since our strategy has more read and write actions in the additional registers (e.g., sample list) than SOTA, SOTA leads to slightly lower processing times. Increasing the output size N_s (up to a certain threshold) and the number of hash functions N_h can improve F1 score for heavy-hitter detection as shown in Figures 10 and 11, but this also has some impact on packet processing time in P4 switches. Figure 13(b) shows how N_s affects packet processing time: it slightly increases as N_s increases significantly. This happens because a higher N_s requires a hash function performing more lookups to obtain the output value. Figure 13(c) shows instead the impact of N_h: results clearly show that increasing the number of hash functions N_h has a more impacting effect on packet processing time than increasing the output size of hash functions N_s.

These evaluations confirm that the size of Count-Min Sketches implemented in the data plane must be carefully defined. In fact, an increase in N_h improves monitoring performance (see Figure 11) but requires larger packet processing time. Moreover, by also referring to Figure 10, correctly dimensioning N_s is of paramount importance to avoid both large packet processing time and F1 score reduction. Finally, note also that packet processing times shown in this section (i.e., in the order of few ms) include the time needed to cross several virtualized layers in the single-node emulated environment. In real carrier-grade hardware (e.g. Barefoot Tofino, with throughput in the order of 6.5 Tb/s or more [37]), packet processing time is expected to be several orders of magnitude lower (i.e., in the order of ns or few μs).

C. Evaluation of the controller response time

For each different time interval size, we measured the response time in 10 intervals and computed the average response time. Table V shows the average response time. It is around 1.5 s and slightly increases while increasing T_{int}, since more data needs to be processed for longer time intervals. These results mean that network-wide heavy hitters are detected in average 1.5s after that the controller receives the first flag from any switch identifying a potential heavy hitters. Note that this time depends on the computational capacity of the controller, so we expect it to be even smaller while adopting carrier-grade hardware for the controller in real deployments. Moreover, we do not include in the evaluated response time the retrieval time of the flag messages from the programmable switches, which strongly depends on where the controller is placed and is at most in the order of few tens of ms [45] (i.e., negligible with respect to the controller response time).

TABLE V
AVERAGE CONTROLLER RESPONSE TIME

Time interval T_{int}	5s	10s	15s	20s
Response time	1.4488s	1.4542s	1.4656s	1.5036s

VIII. RELATED WORK

A. Partial deployment of SDN solutions in ISP networks

The appearance of SDN simplifies the network management and enhances the flexibility of the network. However, currently it is not feasible to upgrade all legacy switches to SDN switches due to the limitation of budgets and operational burdens, so the current trend for network operators is to deploy a limited number of SDN switches and make the network best work in a hybrid environment. A good strategy for partial SDN deployment is thus needed to cost-effectively bring benefits to ISPs. Unfortunately, obtaining the best partial deployment of SDN switches is a NP-hard problem [46]. In literature, most of the works focus on the problem of partial deployment of OpenFlow switches [13] in legacy infrastructures, and either Integer Linear Programming (ILP) [47] or incremental deployment heuristic algorithms [46][48][49][50] have been adopted to solve such problem, focusing on interoperability and routing issues in a hybrid environment while achieving the best load balancing or maximizing the throughput. Incremental deployment heuristic strategies are a good approach to solve the problem of partial deployment, since they aim at iteratively replacing legacy equipment by ensuring local optimal performance. However, the previous work neither takes into account the problem of incremental deployment of programmable switches in a legacy infrastructure to improve network monitoring performance nor proposes a solution for topological placement of programmable switches: with our paper, we try to fill this gap. Moreover, our solution is interoperable with other techniques: the raw data gathered from legacy devices could be used with filtered data collected from programmable switches for an improved network monitoring.

B. Network-wide heavy-hitter detection in programmable data planes

In the last years, many strategies have been proposed to monitor heavy hitters directly in the data plane by exploiting the flexibility of programmable switches. Some among them are OpenSketch [51], UnivMon [16], Elastic Skecth [18], FlowRadar [17] , SketchVisor [19], NitroSketch [52], Sketch-Learn [53] and HashPipe [20]. However, they only focused on heavy-hitter detection at a single SDN switch, but this is not enough for heavy-hitter detection in large networks, since some heavy hitters may be undetected or wrongly detected by relying on limited information at a single location.

Thus, the concept of *network-wide heavy hitter* has been introduced in literature [54][55][56][57]. A network-wide heavy hitter uses distributed information, which can be made available by programmable switches, to accurately and effectively monitor heavy hitters from a global perspective. Harrison et al. [22] and Basat et al. [21] have proposed two different strategies to monitor network-wide heavy hitters. In Harrison's strategy [22], at the end of each time interval, if any heavy hitter has been detected through a local threshold-based mechanism in P4-enabled switches, the controller polls the programmable switches and uses a different (global) threshold-based mechanism to decide whether local heavy hitters are network-wide heavy hitters or not. However, in their strategy, packets belonging to the same flow are counted multiple times by different switches, and this duplicated information is not discarded by the controller while estimating network-wide heavy hitters: for this reason, it is very difficult to correctly set the global threshold. Basat's work [21] provides a solid method for network-wide heavy-hitter detection by using a data streaming model, but the introduced communication overhead and occupied memory are significant. Another key limitation is that the hash functions needed by their strategy do not exist in practice. Our network-wide heavy-hitter detection strategy is similar to the one proposed by Harrison et al., but we define new and more intuitive local and global thresholds and we exploit information on distinct flows to prevent duplicate counting of packets, thus reducing the communication overhead and occupied memory in programmable switches and improving monitoring performance.

IX. Conclusion and future work

In this paper, we presented a new greedy algorithm for an effective incremental deployment of SDN programmable switches in legacy infrastructures that aims at monitoring as many distinct network flows as possible. This algorithm best supports monitoring tasks such as heavy-hitter detection when only a limited number of legacy devices can be replaced with programmable switches. We also proposed a novel network-wide heavy-hitter detection strategy which works well in conjunction with our incremental deployment approach. This strategy has been implemented in P4 language and tested in an emulated environment. Both the incremental deployment algorithm and the network-wide heavy-hitter detection strategy were proven to outperform existing approaches. By adopting our incremental deployment algorithm, network operators can ensure very good monitoring performance by replacing less than half of the legacy devices in the network. Moreover, our network-wide heavy-hitter detection strategy outperforms an existing approach both when only a limited number of programmable switches is deployed and when the network is entirely upgraded, since it allows network operators to strike a balance between heavy-hitter detection accuracy, communication overhead and occupied memory. As marginal side effect, our strategy has been shown to lead to slightly more packet processing time in the execution of the P4 pipeline than the considered state-of-the-art approach.

As future work, we intend to validate our network-wide heavy-hitter detection strategy in a testbed composed of three programmable P4 switches, and thus test it in a real environment. Furthermore, we also plan to extend our strategy to execute a wider range of network-wide monitoring tasks, such as heavy change detection or entropy estimation.

Acknowledgement

The research leading to these results has received funding from the European Commission within the H2020 R&I program, Grant Agreement No. 856726 (GN4-3 project).

References

[1] D. Ding, M. Savi, G. Antichi, and D. Siracusa, "Incremental deployment of programmable switches for network-wide heavy-hitter detection," in *IEEE Conference on Network Softwarization (NetSoft 2019)*, 2019.

[2] N. Duffield, C. Lund, and M. Thorup, "Charging from Sampled Network Usage," in *ACM Internet Measurement Workshop (IMW)*, 2001.

[3] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2002.

[4] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving Traffic Demands for Operational IP Networks: Methodology and Experience," in *IEEE/ACM Transactions on Networking*, vol. 9, issue 3, 2001.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine Grained Traffic Engineering for Data Centers," in *ACM COnference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2011.

[6] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, 2005.

[7] C. Wang, T. T. Miu, X. Luo, and J. Wang, "Skyshield: a sketch-based defense system against application layer DDoS attacks," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, pp. 559–573, 2018.

[8] Y. Liu, W. Chen, and Y. Guan, "Identifying high-cardinality hosts from network-wide traffic measurements," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 5, pp. 547–558, 2016.

[9] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum, "New Streaming Algorithms for Fast Detection of Superspreaders," in *Network and Distributed System Security Symposium (NDSS)*, 2005.

[10] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined Wan," in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[11] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford, "Central control over distributed routing," in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[12] A. Gupta, R. MacDavid, R. Birkner, M. Canini, N. Feamster, J. Rexford, and L. Vanbever, "An industrial-scale software defined internet exchange point," in *USENIX Networked Systems Design and Implementation (NSDI)*, 2016.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM Computer Communication Review*, vol. 38, no. 2, 2008.

[14] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, “Opennetmon: Network monitoring in openflow software-defined networks,” in *IEEE Network Operations and Management Symposium (NOMS)*, 2014.

[15] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, “P4: Programming protocol-independent packet processors,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[16] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, “One sketch to rule them all: Rethinking network flow monitoring with UnivMon,” in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[17] Y. Li, R. Miao, C. Kim, and M. Yu, “FlowRadar: A better NetFlow for data centers.” in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[18] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, “Elastic sketch: Adaptive and fast network-wide measurements,” in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[19] 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 *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[20] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, “Heavy-hitter detection entirely in the data plane,” in *ACM Symposium on SDN Research (SOSR)*, 2017.

[21] R. B. Basat, G. Einziger, S. L. Feibish, J. Moraney, and D. Raz, “Network-wide routing-oblivious heavy hitters,” in *IEEE/ACM Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2018.

[22] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, “Network-wide heavy hitter detection with commodity switches,” in *ACM Symposium on SDN Research (SOSR)*, 2018.

[23] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, “Heavyguardian: Separate and guard hot items in data streams,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2018, pp. 2584–2593.

[24] G. Cormode, “Count-min sketch,” in *Springer Encyclopedia of Database Systems*, pp. 511-516, 2009.

[25] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, “Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm,” in *Discrete Mathematics and Theoretical Computer Science*, pp. 137-156, 2007.

[26] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois, “A declarative and expressive approach to control forwarding paths in carrier-grade networks,” *ACM SIGCOMM computer communication review*, vol. 45, no. 4, pp. 15–28, 2015.

[27] 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.

[28] M. Chowdhury, R. Agarwal, V. Sekar, and I. Stoica, “A longitudinal and cross-dataset study of internet latency and path stability,” *Dept. EECS, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2014-172*, 2014.

[29] “Python implementation of HyperLogLog,” https://github.com/ekzhu/datasketch.

[30] “P4 behavioral model,” https://github.com/p4lang/behavioral-model.

[31] “P4 implementation of NWHHD+,” https://github.com/DINGDAMU/Network-wide-heavy-hitter-detection.

[32] “P4 register definition,” https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[33] Y. Collet, “xxhash–extremely fast hash algorithm,” https://github.com/Cyan4973/xxHash.

[34] “Python implementation of Count-Min sketch,” https://github.com/rafacarrascosa/countminsketch.

[35] “CAIDA UCSD Anonymized Internet Traces Dataset,” http://www.caida.org/data/passive/passive_dataset.xml.

[36] G. Castagnoli, S. Brauer, and M. Herrmann, “Optimization of cyclic redundancy-check codes with 24 and 32 parity bits,” *IEEE Transactions on Communications*, vol. 41, no. 6, pp. 883–892, 1993.

[37] “Barefoot Tofino,” https://www.barefootnetworks.com/products/brief-tofino/.

[38] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, “In-network computation is a dumb idea whose time has come,” in *ACM Workshop on Hot Topics in Networks (HotNets)*, 2017.

[39] K. Okamoto, W. Chen, and X.-Y. Li, “Ranking of closeness centrality for large-scale social networks,” in *Springer International Workshop on Frontiers in Algorithmics*, 2008.

[40] M. E. Newman, “Scientific collaboration networks. II. shortest paths, weighted networks, and centrality,” *APS Physical review E*, vol. 64, no. 1, 2001.

[41] D. Sundararajan, “Convolution and correlation,” in *Wiley Discrete-wavelet Transform*, pp. 21-36, 2015.

[42] “P4 implementation of Count-Min sketch,” https://github.com/open-nfpsw/M-Sketch/tree/master/count_min_Vanilla_P4.

[43] “Mininet,” http://mininet.org/.

[44] “P4c,” https://github.com/p4lang/p4c.

[45] A. K. Singh and S. Srivastava, “A survey and classification of controller placement problem in sdn,” *International Journal of Network Management*, vol. 28, no. 3, p. e2018, 2018.

[46] D. K. Hong, Y. Ma, S. Banerjee, and Z. M. Mao, “Incremental deployment of SDN in hybrid enterprise and ISP networks,” in *ACM Symposium on SDN Research (SOSR)*, 2016.

[47] M. Markovitch and S. Schmid, “Shear: A highly available and flexible network architecture marrying distributed and logically centralized control planes,” in *IEEE International Conference on Network Protocols (ICNP)*, 2015.

[48] 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*, vol. 25, no. 3, pp. 1861–1875, 2017.

[49] M. Huang and W. Liang, “Incremental SDN-enabled switch deployment for hybrid software-defined networks,” in *IEEE Computer Communication and Networks (ICCCN)*, 2017.

[50] D. Levin, M. Canini, S. Schmid, F. Schaffert, and A. Feldmann, “Panopticon: Reaping the benefits of incremental SDN deployment in enterprise networks,” in *USENIX Annual Technical Conference*, 2014.

[51] M. Yu, L. Jose, and R. Miao, “Software defined traffic measurement with OpenSketch,” in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[52] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, “Nitrosketch: Robust and general sketch-based monitoring in software switches,” in *Proceedings of the ACM Special Interest Group on Data Communication*. ACM, 2019, pp. 334–350.

[53] Q. Huang, P. P. Lee, and Y. Bao, “Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 576–590.

[54] G. Cormode and M. Garofalakis, “Sketching streams through the net: Distributed approximate query tracking,” in *International conference on Very large data bases*, 2005, pp. 13–24.

[55] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston, “Finding (recently) frequent items in distributed data streams,” in *IEEE International Conference on Data Engineering (ICDE)*, 2005, pp. 767–778.

[56] K. Yi and Q. Zhang, “Optimal tracking of distributed heavy hitters and quantiles,” *Algorithmica*, vol. 65, no. 1, pp. 206–223, 2013.

[57] Q. Huang and P. P. Lee, “A hybrid local and distributed sketching design for accurate and scalable heavy key detection in network data streams,” *Computer Networks*, vol. 91, pp. 298–315, 2015.