Lightweight Acquisition and Ranging of Flows in the Data Plane

ANDREA MONTERUBBIANO, University of Rome - Sapienza, Italy
JONATAN LANGLET, Queen Mary University of London, UK
STEFAN WALZER, Cologne University, Germany
GIANNI ANTICHI, Politecnico di Milano and Queen Mary University of London, Italy, UK
PEDRO REVIRIEGO, Universidad Politécnica de Madrid, Spain
SALVATORE PONTARELLI, University of Rome - Sapienza, Italy

As networks get more complex, the ability to track *almost all the flows* is becoming of paramount importance. This is because we can then detect transient events impacting only a subset of the traffic. Solutions for flow monitoring exist, but it is getting very difficult to produce accurate estimations for every *<flowID,counter>* tuple given the memory constraints of commodity programmable switches. Indeed, as networks grow in size, more flows have to be tracked, increasing the number of tuples to be recorded. At the same time, end-host virtualization requires more specific flowIDs, enlarging the memory cost for every single entry. Finally, the available memory resources have to be shared with other important functions as well (e.g., load balancing, forwarding, ACL).

To address those issues, we present FlowLiDAR (Flow Lightweight Detection and Ranging), a new solution that is capable of tracking almost all the flows in the network while requiring only a modest amount of data plane memory which is not dependent on the size of flowIDs. We implemented the scheme in P4, tested it using real traffic from ISPs and compared it against four state-of-the-art solutions: FlowRadar, NZE, PR-sketch, and Elastic Sketch. While those can only reconstruct up to 60% of the tuples, FlowLiDAR can track 98.7% of them with the same amount of memory.

CCS Concepts: • **Networks** → **Network measurement**.

Additional Key Words and Phrases: Programmable Data Plane, High-Speed Networking, Flow Measurement

ACM Reference Format:
Andrea Monterubbiano, Jonatan Langlet, Stefan Walzer, Gianni Antichi, Pedro Reviriego, and Salvatore Pontarelli. 2023. Lightweight Acquisition and Ranging of Flows in the Data Plane. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 3, Article 44 (December 2023), 24 pages. https://doi.org/10.1145/3626775

1 INTRODUCTION

Nowadays, telemetry is the foundation for many network management tasks such as traffic engineering, performance diagnosis, and attack detection [6, 24, 25, 29, 52, 56, 60–62]. In particular, flow-level monitoring is a first-class citizen, as it allows the debugging (e.g., the process of finding causes and victims) of many performance-critical data plane events, such as packet drops [35, 62], congestion [36] or path change [6, 29]. Here, it is of paramount importance capturing information related to *almost all flows*, so to guarantee fine-grained traffic analysis [25] that make it possible to

Authors' addresses: Andrea Monterubbiano, University of Rome - Sapienza, Rome, Italy; Jonatan Langlet, Queen Mary University of London, London, UK; Stefan Walzer, Cologne University, Cologne, Germany; Gianni Antichi, Politecnico di Milano and Queen Mary University of London, Milan,London, Italy, UK; Pedro Reviriego, Universidad Politécnica de Madrid, Madrid, Spain; Salvatore Pontarelli, University of Rome - Sapienza, Rome, Italy.

2476-1249/2023/12-ART44
https://doi.org/10.1145/3626775

detect events impacting only a subset of the traffic (e.g., transient loops [32], blackholes [61], and switch faults [62]). Unfortunately, although state-of-the-art programmable switch ASICs allow for a configurable per-packet logic, they are also well-known to be resource constrained [30, 41, 47], limiting their ability to record *all flows in the network*.

Solutions for flow-level monitoring exist [23, 34, 50, 59], but three trends are making this practice a steadily more challenging task: (1) networking devices are becoming commonly adopted in support of many use-cases, impacting the amount of resources to be solely dedicated to flow-monitoring; (2) the increase in switches' per-port capacity has consistently outpaced the growth of their internal ASIC memory, making the latter a very scarce resource; (3) the rise of end-host virtualization and cloud-native services imposes flow IDs that are larger than the common 5-tuple [3, 17] (i.e., IPs, protocol and layer 4 ports) further adding pressure to the memory requirements of flow-level telemetry.

In an effort to reduce the memory requirements at switches, the research community has been proposing various solutions [23, 34, 54]. Some rely on probabilistic data structures with bounded errors to store counters and keep track only of the flowIDs for heavy flows (e.g., ElasticSketch [54]) so to enable the reconstruction of *<flowID, counter>* tuples. The problem is that they can suffer of potentially unacceptable inaccuracies when required to track *short flows* [23]. Others encode flowIDs and associated counters directly in the ASIC (e.g., FlowRadar [34]) or adopt signal-processing techniques to limit the amount of resources to be used (e.g., NZE [23]). Although they can potentially track all flows in the network, they experience a loss in accuracy when fine-grained flow-level telemetry (i.e., flow IDs more specific than the standard 5-tuple) is needed (§ 2). An alternative approach in this scenario is to send the flowIDs to the control plane and keep only the counters in the dataplane as done in the PR-sketch [49]. This makes the dataplane memory independent of the flowID size. However, the dataplane memory needed for the filter used to detect flows and the counters is still large, limiting the number of flows that can be monitored.

We present FlowLiDAR (§ 3), a new solution that can track *almost all flows present in the network*. As in the PR-sketch we decouple flowIDs from their associated counters. The former are continuously reported from the ASIC and stored in the switch OS, while the latter are stored directly in the ASIC using a combination of probabilistic data structures. This allows to have a system where the required dataplane memory does not depend anymore on how big flowIDs are. At the end of a configurable time window, both parts of the information can be combined together to retrieve the exact *<flowID, counter>* values for almost all the flows.

FlowLIDAR introduces key innovations over the PR-sketch design that allow us to significantly reduce the amount of data plane memory needed to achieve close to 100% accuracy in the estimation of the flows. For example, we use an exact equation solving in the dataplane to extract the size of the flows from the counters at the end of a measurement epoch. We also propose a new mechanism, named *lazy updates* that eliminates the need to use counters in the ASIC for most flows that have only one or a few packets. This not only reduces the dataplane memory needed for the counters, it also reduces the complexity of the equation solving and improves its accuracy. We implemented FlowLiDAR in P4 (§ 4) and tested against real traffic traces taken from a large ISP (§ 5). We found that using the same amount of memory, it improves the accuracy of flow counting in terms of average relative error (ARE) and average absolute error (AAE) when compared against state-of-the-art solutions such as NZE, the PR-Sketch and Elastic Sketch by up to 10x, 100x, and 100x, respectively. Moreover, while FlowLiDAR is able to successfully track 98.7% of existing flows, other techniques can only reconstruct up to 60% of them. The main novelties and features of our proposed flow monitoring scheme, FlowLiDAR are:

(1) Most flows with only one or a few packets do not use counters in the dataplane. This reduces the number of counters needed very significantly as most flows tend to have few packets.
(2) The extraction of the values from the shared counters is done using a mathematical formulation that increases the accuracy of the flow packet count estimates and reduces the memory needed.
(3) It can be efficiently implemented in programmable dataplanes using P4.
(4) It outperforms state of the art algorithms in terms of memory *vs* accuracy trade-off.

2 MOTIVATION

Tracking *all flows* present in a network is of paramount importance [23, 34, 50, 53]. This is because it allows to capture events that otherwise would be easily missed: transient loops, blackholes, and switch faults (i.e., data centers operators have reported table corruptions that lead to packet losses or incorrect forwarding for a small portion of the traffic [62]). Those may affect just a few packets during a very short time period introducing losses into the network. The problem is that even just a few losses can cause significant tail-latency increases and throughput drops for both TCP and RDMA traffic, potentially leading to violations of service level agreements (SLAs) and even a decrease of revenue [10]. Unfortunately, tracking all flows with high precision is also becoming increasingly difficult, mainly because of three main trends:

Trend #1: networking devices are progressively used in support of many use-cases. The rise of programmable data-planes has allowed the research community to explore their help in support of a widespread number of applications: congestion control [20, 36], load balancing [2, 28], caching [26] or machine learning accelerators [46], to name a few.
Consequence: the more functions are added to the data-plane, the more pressure is put on its memory and logic resources, further reducing its (already limited) capabilities when it comes to just flow-level telemetry.

Trend #2: increasing mismatch between link speeds and memory capacity. In the last decade, switching ASICs have increased their aggregate capacity from less than a Tb/s to more than 50 Tb/s.[1] This has allowed state-of-the-art switches to support more and higher-bandwidth physical ports. The direct consequence is the need to internally support the recording of an ever-increasing number of flows: from an analysis of real traffic traces has been indeed estimated the presence of *120K active flows per Gbps of traffic*, leading to over 3000M for 25.6 Tbps switches [47]. Unfortunately, such an increase in switch speed has consistently outpaced the growth of its internal ASIC memory. In Figure 1, we compare the aggregate bandwidth against the internal memory available for the latest generations of state-of-the-art Broadcom switches[2]. Here we can see, that the memory is not keeping up with

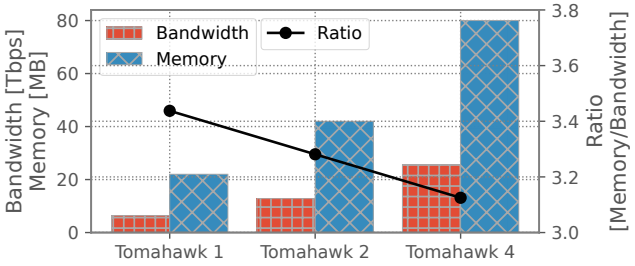

Fig. 1. Memory/bandwidth ratio of different Broadcom Tomahawk switch generations

[1] A detailed discussion is available at https://elegantnetwork.github.io/posts/A-Summary-of-Network-ASICs/
[2] Data are taken from https://people.ucsc.edu/~warner/buffer.html

the bandwidth.
Consequence: there is an increasing need to record *more flows* with *less memory*.

Trend #3: the need for larger flowIDs. The rise of end-host virtualization, consolidating possibly a very large number of diverse services on a single system, and the rapid convergence of cloud-native service access APIs based on the HTTP communication protocol [3, 4] are posing new challenges in flow-level telemetry. Indeed, the increasing generality of higher-layer protocols, such as HTTP, has led operators to explore their feasibility beyond the Web, i.e., for media streaming (i.e., WebRTC[3]), remote procedure calls (i.e., gRPC[4]), data center networking [4] or transport of DNS. Crucially, when everything is encoded into HTTP, basic L2–L4-layer insight into traffic is no longer adequate to understand network behavior [3]. Indeed, the traditional five-tuple is of little use when the destination port is uniformly 80 (HTTP) or 443 (HTTPS) regardless of whether a particular traffic instance is a REST API call or a long-lived media stream.
Consequence: there is a need for storing more fine-grained flow identifiers using additional packet header fields, which in turn puts even more pressure on the memory requirements in the ASIC.

2.1 Limits of the current solutions

In an effort to address the challenges imposed by the aforementioned trends, state-of-the-art solutions commonly use probabilistic data structures [34, 37, 54] to reduce the switch memory requirements at the cost of query accuracy. Despite their theoretical error bounds, existing solutions still suffer to provide comprehensive guarantees for *all flows* in a network. This is because existing algorithms are typically designed to provide guarantees for specific flows (e.g., heavy hitters [14, 48] or super-spreaders [58]) and/or aggregated flow statistics (e.g., cardinality [15] or traffic distribution [31]). As a consequence, when applying these solutions to the entire traffic, the derived bounds are too coarse-grained to work. This leads to a considerable gap: only a small portion of flows actually benefit from the theoretical bounds, while the remaining flows still exhibit poor accuracy.

A recent paper has highlighted this issue and proposed a solution [23], named Near Zero Error (NZE) which, as its name states, reduces the errors but does not completely eliminate them. In more detail, accuracy for small flows is still not guaranteed, especially if many bits are needed for the flow identifiers. Therefore, the problem still remains when taking into consideration trend #3: when adopting increasingly bigger flow identifiers. This has been addressed by the PR-sketch that sends all the flowIDs to the control plane and keeps only the flow detector and the counters on the dataplane [49]. However, the PR-sketch still requires a significant amount of dataplane memory per flow (see Section 5.3 for details), making it less efficient than existing solutions except for very large flowIDs. To better understand this, we conducted an experiment and compared different state-of-the-art algorithms (i.e., Elastic

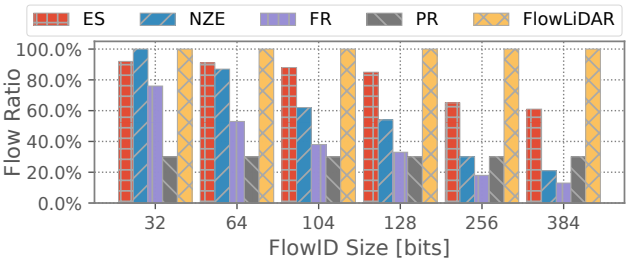

Fig. 2. Fraction of flows that can be monitored with a fixed amount of memory for ElasticSketch (ES), NZE, FlowRadar (FR), PR-Sketch (PR), and FlowLiDAR

[3] https://webrtc.org/
[4] https://grpc.io/

Sketch [54], NZE [23], FlowRadar [34] and the PR-sketch [49]). In Figure 2, we show the fraction of flows that can be accurately tracked when using a fixed amount of memory, i.e., 10 MB and considering 1.2M active flows to be tracked [47].

In this situation, NZE can successfully track *all flows* only if flowIDs are just 32 bits (an IP address). When a more fine-grained flow analysis is needed, larger flow identifiers must be adopted, impacting the ability of NZE to track all flows. Indeed, when using just the standard 5-tuple, the flow coverage can drop to 60%. On the other hand, ElasticSketch is able to track only 90% of flows with high accuracy (<1% relative error) with 32 bits, but its performance degrades more gracefully than NZE[5]. FlowRadar is not able to track all flows for any flowID size and its coverage is lower than that of NZE and Elastic Sketch. When more packet header fields need to be considered, as in the case of tunneled connections requiring VXLAN + 5-tuple or when upper layer protocol headers are needed, the flow coverage can drop further: in the presence of 256-bits flowIDs, the flow coverage of Elastic Sketch, NZE and FlowRadar drop to approximately 60%, 35% and 20%, respectively which is clearly not acceptable. Finally, the PR-sketch flow coverage does not depend on the flowID size as expected but it is below 40%, so worse than existing schemes for small flowID sizes and also not acceptable. Instead, our solution is able to track more than 99% of flows regardless of flowID size.

3 LIGHTWEIGHT DETECTION AND RANGING WITH FLOWLIDAR

This section first presents our overall approach and then each component of FlowLiDAR is described in detail.

3.1 Overall Approach

The concept of FlowLiDAR is illustrated in Figure 3 and makes use of both switch data and control planes. The idea, similarly to [49] is to place all the functions that have to be done per packet in the data-plane while those that are much less frequent are placed in the control-plane. In more detail, flow detection and counting of packets is done in the data-plane while the processing of new flows and the fine grained computation of the number of packets per flow is done in the control-plane. This approach needs to send information between the control and data planes and thus the bandwidth on this interface may be an issue. In the following we discuss why this should not be the case in switching ASICs that have high speed links between both planes and propose variants of our design that can reduce the amount of information sent on that interface.

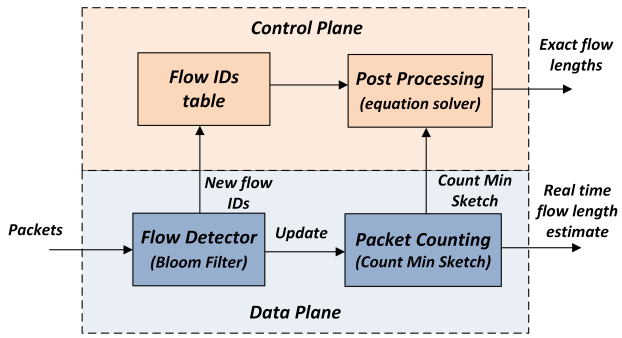

Fig. 3. Block diagram of the proposed FlowLiDAR. The data-plane detects new flows and sends the IDs to the control plane and counts the packets. The control plane stores the flow IDs and periodically computes the exact values of the flow lengths

[5]Elastic Sketch uses two main data structures: a hash table for the heavy part, which is flowID size dependent, and a large count sketch to count the small flows. We chose the memory ratio between the two structures, varying the flowID size to bound the average relative error below 50%.

This split is based on the observation that new flows are only a small fraction of the packets [27]. This has been corroborated by analysing the three different CAIDA traces also used in the evaluation section[6]. The results show that even on a one second window, the average number of packets per flow is in the range of 5-10 and the number increases for larger window sizes. Therefore, instead of storing the flows' IDs in the switch we can just detect new flows and send their IDs to the control plane. This eliminates the need to store the flows' IDs on the data-plane memory thus reducing the memory footprint dramatically. With this idea in mind, what we need to have on the data-plane is the detection of new flows which can be detected using for example a Bloom filter (BF) [34]. This detector has to be in the data-plane as all packets have to be checked. In FlowLiDAR, the detector is a BF that is optimized to reduce its memory footprint [22].

The same reasoning applies to the counters used to count packets, they have to be accessed on every packet and thus are also placed on the data-plane. In more detail, we use a Count-Min Sketch (CMS) for packet counting [43]. This enables us to provide in data-plane flow size estimation using the standard CMS algorithm in real-time. Additionally, snapshots of the BF and CMS are also sent periodically to the control plane and the data structures are reset to start a new measurement epoch.

The control plane stores the FlowIDs and when it receives a snapshot of the CMS and BF computes the exact flow sizes using a more complex processing that models the CMS as a system of equations. Again this is possible as it is done much less frequently than the per packet operations done in the data plane. Even if extracting the exact values is the main aim of FlowLiDAR, we highlight that a further memory reduction is possible at the cost of accepting an approximate resolution of the CMS system of equations. Details about this option are provided in sec. 3.4.3 and evaluated in 5.4. In the following subsections, we describe each of the FlowLiDAR components in more detail discussing also the relations and interactions between the blocks.

3.2 Flow Detector

The flow detector has the mission of identifying new flows and sending their IDs to the control plane. In this subsection we describe three methods to implement the Flow Detector. The first method is based on a standard BF and provides a baseline for the other two methods. A second method, based on a modification of the BF that we called lazy BF updates, reduces the false positive probability of the standard method at the expense of an increase in the bandwidth required by the control plane. When the control plane bandwidth is a bottleneck we can use a third method. This method is based on the idea of sending to the control plane only the FlowIDs of flows that were not present in the previous epoch. To this aim, we use a pair of BF, one storing the FlowIDs of the previous epoch and one that stores those of the current epoch. In all cases, flow detection is done as soon as the first packet of the flow returns a negative on the BF and it is immediately reported to the control plane, a process that takes only a few microseconds in the worst case.

Standard BF. The detection of FlowIDs is done per measurement epoch, so that after taking a snapshot and sending it to the control plane, the detector is reset to start a new epoch. To this end, the flow detector has to check all the packets and thus has to perform simple operations. As in [34],[49], we use a BF to detect new flows. The overall approach is to check each packet on the BF and on a negative, insert the flow on the BF (so that subsequent packets of the flow return a positive), and send its FlowID to the control plane. The use of a BF eliminates the need to store all the FlowIDs to detect new flows thus reducing the memory needed. However, it has the drawback of having false positives so that a few flows may not be detected. For a given number of target

[6]Full names of traces are: equinix-chicago.dirA.20160121-130000.UTC.anon.pcap, equinix-chicago.dirA.20160218-133000.UTC.anon.pcap, equinix-chicago.dirB.20160317-140000.UTC.anon.pcap

flows, the probability of false positives can be made small by appropriately selecting the BF array size m and the number of arrays and hash functions k. The initial flow detection algorithm is shown in Algorithm 1.

Algorithm 1 Initial algorithm for flow detection

```
Reset the BF
Start the epoch timer
for Each packet with FlowID x do
    Query element x in the filter
    if Negative then
        Add x to the filter
        Send FlowID to the control plane
    end if
    Send x to packet counting block
    Timer expired? If yes restart the process
end for
```

The BF maps each FlowID to k independent arrays of m bits. The use of independent arrays per hash function makes it possible to access each position in parallel and facilitates the implementation in a programmable data plane as will be seen in section 4. Additionally, it enables a more advanced flow detection scheme that we implement in FlowLiDAR and is described next.

The fraction of flows that are not detected during an epoch can be estimated by computing the false positive probability of the filter when each new flow arrives and then adding all those probabilities together. The false positive probability of a filter that has k arrays of m bits and on which i elements have been inserted can be approximated by:

$$P(i) \approx (1 - e^{-\frac{i}{m}})^k \tag{1}$$

Then if on an epoch there are n flows, the fraction of false positives can be estimated by adding the probabilities of the second flow $P(1)$, the third flow $P(2)$ and so on until the n^{th} flow obtaining:

$$FPP \approx \frac{\sum_{i=2}^{n} P(i-1)}{n} \tag{2}$$

Lazy updates BF. The advanced BF scheme, that we denote as lazy updates BF is based on the observation that depending on the time between snapshots, most flows may have a single or just a few packets on that period. As an example for the CAIDA traces used in this paper, we reported in Table 1 the percentage of flows with just one, two or three packets for a one second epoch.

Table 1. Percentage of flows having one, two or three packets

1-packet	2-packets	3-packets	more than 3 packets
39%	18%	10%	33%

When that is the case, it may be beneficial not to set all the bits for the new flow in the BF to one but just one at a time. This would reduce the number of ones in the filter and thus its false positive probability. For example, if 40% of the flows have only a single packet, we would be reducing by close to 40% the insertions on the second to k^{th} arrays. In fact, using independent arrays is better in this configuration to reduce the false positive rate similar to what happens in d-left hashing [38]. Deriving the false positive probability for flows in this advanced scheme is more complex but an approximation can be easily obtained. Let us denote by $l(j)$ the fraction of flows that have j or more packets in the epoch. Consider a filter with k

arrays on which i elements have been inserted. Then in the j^{th} array there will be approximately $i \cdot l(j)$ elements inserted. With that assumption, the false positive probability of the filter can be computed as the product of the fraction of bits set to one in each array which is given by $(1-e^{-\frac{i \cdot l(j)}{m}})$ obtaining:

$$P_a(i) \approx \prod_{j=1}^{k} (1 - e^{-\frac{i \cdot l(j)}{m}}) \tag{3}$$

where $l(j)$ accounts for the fact that a fraction of the flows has j or less packets and thus are not inserted on tables $j+1, \ldots, k$ unless they suffer false positives on the previous tables. This approximation would tend to underestimate the false positive probability as there will be false positives. For example, a flow with a single packet may find the bit set on the first BF table and would thus be inserted on the second and so forth. Finally, the fraction of flows that are false positive can be estimated by using $P_a(i)$ instead of $P(i)$ in equation (2).

The price paid when using the advanced scheme is that flows that have more than one packet, may be sent several times to the control plane. Let us consider the bandwidth needed to send the FlowIDs to the control plane. Each FlowID has 13 bytes in IPV4[7]. On average the number of packets per flow is much larger than one, for example even when considering one second windows, the CAIDA traces have 5-10 packets per flow. Considering an average packet size of 500 bytes, the overhead would be below 13/(500*5) so roughly 0.5% which would be acceptable in most cases. Indeed as discussed before, bandwidth grows faster than on-chip memory and thus using a small fraction of the bandwidth in exchange for a large reduction on the memory needed (as FlowIDs no longer need to be stored on-chip) is attractive.

Finally, we can also take advantage of this BF optimization to not send the first packets to the packet counting block this has the side benefit of reducing the load on the CMS as will be seen in the next subsection. The advanced flow detection is presented in Algorithm 2. Note that although the algorithm describes a serial implementation, the for loop over the k arrays has no temporal dependencies and thus can be unfolded and executed in parallel with each value of i corresponding to one of the filter arrays.

Algorithm 2 Advanced algorithm for flow detection using lazy updates for the BF

```
Reset the BF
Start the epoch timer
for Each packet with FlowID x do
    for i = 1 to k do
        if h_i(x) == 0 then
            Set h_i(x) = 1 and i = k
            Send FlowID to the control plane
            break
        else
            if i == k then
                Send x to packet counting block
            end if
        end if
    end for
    Timer expired? If yes restart the process
end for
```

[7]The FlowID is composed of the source and destination IP addresses, the protocol and the source and destination ports.

Differential Flow Detector with a pair of BFs. As discussed before, in switching ASICs we do not expect the control plane bandwidth to be a bottleneck, but there may be other implementations on which that may be an issue. In those cases, one option is to send to the controller only the FlowIDs that are not present in the previous epochs. In this way, only the new flows are sent to the controller, while the old ones are retrieved from the snapshot taken in the previous epoch. In particular, in this configuration, the Flow Detector uses two BFs. In the first BF, called *oldBF* all the FlowIDs of the previous epoch have been inserted. This BF is checked when a packet arrives to avoid sending a FlowID that is already in the snapshot stored in the control plane. The second BF, called *currentBF*, works as the standard BF and stores all the FlowIDs that have packets in the current epoch. A FlowID is sent to the controller only if it is not present in both BFs. At the end of a measurement period, the set of active FlowIDs can be retrieved by merging the FlowIDs sent in the current epoch and the FlowIDs of the previous snapshot that are still active. These can be extracted from the previous snapshot by checking which ones are positive in the *currentBF* indicating that with high probability they are still active. At the end of the measurement epoch, the *oldBF* stores the content of the *currentBF* and the *currentBF* is reset. The differential flow detection is presented in Algorithm 3. We remark that the use of a BF pair has been already explored in literature but mainly to avoid filter overloading as discussed in [7],[39].

Algorithm 3 Algorithm for flow detection using a pair of BFs

```
1: copy currentBF into oldBF
2: reset currentBF
3: Start the epoch timer
4: for Each packet with FlowID x do
5:     Query element x in the currentBF
6:     if Negative then
7:         Add x to the currentBF
8:         Query element x in the oldBF
9:         if Negative then
10:            Send FlowID to the control plane
11:        end if
12:    end if
13:    Send x to the packet counting block
14:    Timer expired? If yes restart the process
15: end for
```

3.3 Packet Counting

The other data-plane function maps each flow to several arrays of counters and increments one counter per array as in a CMS. This enables a fast estimation of the flow size by just taking the minimum of those counters.

Additionally, the CMS contents are sent periodically to the control plane for further analysis. In that analysis, it is beneficial to have as many counters with a value of zero as possible. To achieve it, we exploit the BF used to detect new flows as a counter for the first packets (those sent to the control plane) so that only flows with more than one (or a few if we use the flow detection optimization) packet in the epoch use the counters. This has a large benefit as a significant fraction of the flows no longer need a counter.

Different from the standard CMS, we split the CMS into a set of smaller CMS, each indexed by a master hash function. This choice will introduce a small degradation in the error, but permits to

split the flow analysis presented in subsection 3.4 in a set of disjoint sparse linear systems, thus providing a significant speed-up in the execution of the flow analysis.

3.4 Postprocessing

The control plane, during each epoch, collects the FlowIDs and at the end of the epoch receives the contents of the BF and of the CMS. Then using that information the control plane can compute the exact values of the flow sizes. This is denoted as postprocessing of the information and is done in three stages, some of which are preprocessing for the final stage in the postprocessing:

(1) BF preprocessing.
(2) CMS preprocessing.
(3) CMS equations solving.

each of them is described in the following subsections.

3.4.1 BF preprocessing. An interesting observation is that when lazy updates are used, the BF can be used to identify a fraction of the flows. In more detail, the FlowIDs collected on the control plane can be tested on the snapshot of the BF received from the data plane and those that return a negative have for sure not been added to the CMS. This means that their number of packets corresponds to the number of times that the flowID has been received in the control plane. Therefore, the exact value of the number of packets is obtained for those flows. Additionally, we can remove them from further consideration so simplifying the problem. The BF preprocessing is described in Algorithm 4.

Algorithm 4 BF preprocessing with lazy updates

```
Compute the set of distinct flows D received in the dataplane in the epoch.
Create an empty set C for flows to be processed in the CMS.
for each FlowID x in D do
    Query element x in the BF
    if Negative then
        Set the number of packets x as the number of times it
        was received in the control plane.
    else
        Add x to C
    end if
end for
Use set C in the CMS preprocessing step.
```

3.4.2 CMS preprocessing. Similarly, for any flow that maps to a counter in the CMS with a value of zero, we can get the exact number of packets by counting the number of times that the flowID has been received in the control plane. This has to be the case as the flowID is sent to the control plane when it returns a negative on the filter and after the CMS is updated. Again, these flows can be removed from further consideration. The CMS preprocessing is described in Algorithm 5.

Exact CMS equation solving. Let now $\mathcal{F} = \{f_1, \ldots, f_n\}$ be the set of FlowIDs for which the packet count could not yet be identified. Assume for now that there are no false positives in the BF and hence $\mathcal{F}$ is fully known in the control plane (we comment on the effect of false positives below). Moreover, we have access to the vector $\mathbf{b} = [b_1, \ldots, b_m]^T$ of counters stored in the CMS. The packet count for flow f_j is the number of times that f_j has been received in the control plane plus the number x_j of times that f_j has been added to the CMS. We write $\mathbf{x} := [x_1, \ldots, x_n]^T$ as a vector. The

Algorithm 5 CMS preprocessing

```
Get set C of FlowIDs from previous step.
for each FlowID x in C do
    Query element x in the CMS
    if estimate packet count == 0 then
        Set the number of packets x as the number of times it
        was received in the control plane.
        Remove x from C
    end if
end for
Use set C in the CMS equations solving step.
```

challenge is to compute $\mathbf{x}$ given $\mathbf{b}$, $\mathcal{F}$ and the k hash functions $h_1, \ldots, h_k$ used in the CMS. This problem can be captured by a system of linear equations

$$A\mathbf{x} = \mathbf{b} \tag{4}$$

where $a_{ij} \in \{0, 1\}$ indicates whether $i \in \{h_1(f_j), \ldots, h_k(f_j)\}$, i.e. whether flow f_j has contributed to the counter b_i. Here we assume that $h_1(f_j), \ldots, h_k(f_j)$ are pairwise distinct such that a flow cannot contribute multiple times to the same counter.

The most immediate question is whether $\mathbf{x}$ is uniquely determined by equation (4) or whether there exists $\mathbf{x}' \neq \mathbf{x}$ with $\mathbf{b} = A\mathbf{x} = A\mathbf{x}'$. This is equivalent to the existence of $\mathbf{x}'' \neq 0$ with $A\mathbf{x}'' = 0$, which exists if and only if the columns of A are linearly dependent.

Table 2. Thresholds for CMS equation solving.

k	3	4	5
c_k^*	0.918	0.977	0.992

Assuming that the k hash functions behave like fully random functions, the columns of $A \in \{0, 1\}^{m \times n}$ are *stochastically* independent and contain exactly k ones per column in uniformly random positions. Such matrices have been studied in the literature on cuckoo hashing and random Boolean formulas [12, 13, 44]. Sharp threshold behaviour with respect to the load factor $c = \frac{m}{n}$ has been demonstrated. More precisely, there exists a constant $c_k^* \in (0, 1)$ such that the following holds. For any $c < c_k^* - \varepsilon$ the matrix A has linearly independent columns with probability $1 - n^{-\Omega(1)}$, even when $\mathbb{F}_2 = \{0, 1\}$ is used as the underlying field [12, 13, 44] (this implies independence for underlying fields $\mathbb{Q}$ and $\mathbb{R}$). For any $c > c_k^* + \varepsilon$ there exists with probability $1 - n^{-\Omega(1)}$ a set C of columns of A and a set R of rows of A with $|C| > |R|$ such that all 1-entries within C are within R [12, 16]. This precludes the independence of the column set C over any field. We reproduce the threshold values in Table 2.

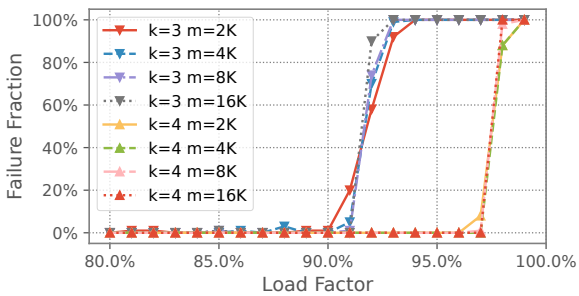

Fig. 4. Percentage of full-rank matrices with different load factors and k values

The behaviour we see in a simulation with 100 trials with $k = 3, 4$ and CMS size of $m = 2K, 4K, 8K, 16K$ as shown in Figure 4 matches these asymptotic predictions quite well.

We remark that the complexity of solving equation (4) is super-linear in m. To improve running times for large m, one could attempt to adopt variants of structured Gaussian elimination as was done in [18] for solving equation (4) over finite fields. Alternatively one could use a smaller load factor exploiting that below the so-called *peeling threshold* equation (4) can be solved in linear time over any group with probability $1 - n^{-\Omega(1)}$ [21, 42].

We rely on a splitting hash function and solve a set of several small systems, instead of a single bigger system solving the various systems in parallel, taking advantage of CPU multi-core architectures. In the evaluation section, we will show how this split reduces the computation time.

On the issue of false positives. In the case where at least one false positive has occurred in the BF we only know a subset $\mathcal{F}_1 \subset \mathcal{F}$ of the remaining FlowIDs, and we do not know $\mathcal{F}_2 = \mathcal{F} \setminus \mathcal{F}_1$. The underlying equation is then $A_1x_1 + A_2x_2 = b$ where A and x are sliced into two parts relating to $\mathcal{F}_1$ and $\mathcal{F}_2$. With no knowledge of A_2 there is no hope of recovering the counts x_2 for $\mathcal{F}_2$. There are, however, several methods for approximately recovering the counts x_1 for $\mathcal{F}_1$ under reasonable assumptions. In an insightful paper by Ting [51], $\epsilon := A_2x_2$ is modelled as a random error vector and our task is to find x_1 that maximizes the likelihood of $\epsilon = b - A_1x_1$. If we assume that the $|\mathcal{F}_2|$ entries of ϵ are independently sampled from a log-concave distribution D, then we obtain a convex optimization problem. For instance, if D is assumed to be a normal distribution, then we recover the linear least squares method where $||A_1x_1 - b||^2$ is to be minimized. This method already yields decent results in practice [33] despite the unfounded assumption on D (e.g. a normal distribution does not guarantee $\epsilon \geq 0$). For even better accuracy, Ting [51] proposes methods for estimating D based on those entries of b to which no key in x_1 has contributed. In our implementation, we compute an approximate value of x_1, called $\hat{x_1}$. We have $\hat{x_1} = A_1^{-1}b$. The approximation error is $e = \hat{x_1} - x_1 = A_1^{-1}\epsilon$. Since we target a small FPR, we will have a small value of $||\epsilon||$, since only a few elements of the vector ϵ are different from zero, thus we expect that also $||e||$ will be small. This is confirmed by our experiments, as we will see in the evaluation section. Furthermore, it is always possible to compare the solution $\mathbf{x}$ provided by the equation solver to the minimum among the k rows of the CMS corresponding to the x_i variable, choosing the minimum among these two values. This guarantees that the error due to the BF will be similar to the approximation of the traditional CMS in the worst case. Finally, we highlight that splitting the system into a set of smaller independent systems further alleviates this problem. In fact, for most of the subsystems, the occurrence of false positives has a negligible impact on the overall error, while for the few subsystems in which this error is significant, we bound the result to those achieved by the traditional CMS approximation.

3.4.3 Approximate CMS equations solving. If the rank r of the matrix is less than the number of variables n the system is underdetermined and has multiple solutions. In particular, the system has a number of free variables that is $l = n - r$. In the following, we describe an algorithm to select the l free variables that minimize the absolute error. This algorithm can be used when the exact CMS equation solving fails. The algorithm, presented as Algorithm 6 selects the system equations with the smallest constant terms b_i and imposes as value of the corresponding variables b_i/n_i, where n_i is the number of variables appearing in the i-th system equation. Fixing the value of these unknowns corresponds to set some additional rows to the $\mathbf{A}$ matrix. The procedure is repeated until the sum of the n_i fixed variables reaches the value of l. The algorithm reduces the underdetermined problem to a linear system with exactly one solution, which can be solved using the standard method used to resolve the sparse linear system of equation (4).

Algorithm 6 Algorithm for the selection of the free variables

```
1: Sort the vector b in ascending order
2: for Each b_i do
3:     Get the variables x_a, ..x_b corresponding to row i of the matrix A that differs from 0
4:     set the values of x_a, ..x_b to b_i/n_i
5:     set l = l − n_i
6:     if l == 0 then
7:         break the loop
8:     end if
9: end for
10: Solve the Ax = b system with the additional equations given by row 4.
```

We remark that this algorithm will select one of the possible solutions for solving the linear system, but we cannot be sure that this is the actual distribution of the number of packets per flow. However, we will show in the evaluation section that the average absolute error

$$AAE = \frac{1}{n}\sum_i |x_i - \hat{x_i}|$$

and the average relative error

$$ARE = \frac{1}{n}\sum_i \frac{|x_i - \hat{x_i}|}{x_i}$$

obtained using this algorithm are better than the AAE and ARE obtained both using the least square method proposed by PR-sketch and the standard CMS algorithm (that for the variables x_i takes the minimum among the buckets addressed by f_i).

4 P4 IMPLEMENTATION

We implemented FlowLiDAR in 500 lines of $P4_{16}$ code[8] for Tofino 2, using the Barefoot SDE 9.7[1]. The results are obtained using a 4x128K BF, with 64 count-min sketches each containing 5 rows of 1K 16-bit counters. The resource costs, as shown in Table 3, are relatively modest and leave plenty of room for any co-located functionality at the programmable switch.

Table 3. Resource footprint imposed by FlowLiDAR in Tofino 2. These numbers are based on a 4x128K BF, and 64 5x1K 16-bit CM sketches

Version	Pipeline Stages	SRAM	sALU	TCAM	Hash Bit
FlowLiDAR	2	5.1%	11.3%	0.3%	2.7%
Lazy FlowLiDAR	3	5.4%	11.3%	0.3%	3.1%

We notice that our prototype uses a non-negligible quantity of Stateful ALUs, though. This is because an efficient implementation without recirculation necessitates a dedicated stateful ALU for each of the 9 hash functions used for the combined BF and CMS dimensions. Here, the indexing in the BF and CMS and the selection of which sketch to apply are all based on the switch-native CRC engine, using custom polynomials that are statically configured at compile-time for a high level of independence between the hash functions.

Finally, it is worth noting that the lazy version of FlowLiDAR requires more stages (+3 compared to +2 stages). This is due to the introduction of a strict dependency between the BF-bits, which forces the compiler to stretch the BF across several hardware stages.

[8]The P4 source code and the simulator are available at this link: `https://github.com/FlowLidar/FlowLidar`.

5 EVALUATION

To evaluate the proposed FlowLiDAR, a software simulator has been developed. The simulator reproduces the behavior of the P4 implementation while providing more flexibility in terms of the number of used hashes and sizing of data structures, thus providing better insights into the behavior of our system. The code has then been used to monitor the flows in three CAIDA traces using our FlowLiDAR with different configuration parameters. In particular, we consider three different traces taken from the 2016 dataset [8]: (C1) 21/01/2016 Minute 13:00, (C2) 18/02/2016 Minute 13:30, and (C3) 17/03/2016 Minute 14:00. The characteristics of the three CAIDA traces are reported in Table 4. Before proceeding to discuss the results, it is important to note that the relative performance of the different sketches will be similar when the link speeds increase from the 10G of the CAIDA traces to faster links such as those currently used in modern data centers.

Table 4. Characteristics of the CAIDA traces used in the evaluation

short name	duration	# of pkts	# of flows	average bit rate	average packet size
C1	60 sec	31M	905K	2.1 Gbps	509B
C2	58 sec	31M	781K	3.1 Gbps	722B
C3	60 sec	34M	579K	4.1 Gbps	898B

In the first experiment, we use one second epochs for measurements and a BF with four arrays of 128K bits and a CMS with 64 arrays of 1K counters of 16 bits. The use of the basic scheme and the lazy update are evaluated and the results are shown in Figures 5-12.

The plots report the percentage of flows with exact results per epoch, the percentage of flows not detected due to FPs, and the bandwidth to the control plane. It can be noticed that FlowLiDAR with the standard BF is able to exactly (with no error) estimate more than 80% of flows. We do not report plots for sake of space but our experiments show that for 95% of flows the error is less or equal to 1. Consequently, the average absolute and relative error of FlowLiDAR are well below those of the traditional CMS evaluation (see Figures 7-8). The fraction of flows with an error greater than one are due to the pollution caused by the untracked flows (Figure 6), which are the flows not detected due to a false positive in the BFs. The plots also show that the use of the lazy update greatly reduces the number of false positives (Fig, 6). This improves both the fraction of flows with zero error, which approaches 100%, and the average absolute and relative errors (Fig, 7-8).

The drawback of the use of lazy updates is the additional required bandwidth, which grows from 60K flowIDs per epoch of the standard BF to 130K flowIDs per epoch (Fig, 12).

5.1 Benefit of lazy update BF

As the lazy updates have additional margin we run the same experiment but reducing the CMS to half, using 32 arrays of 1K counters of 16 bits. The lazy update with 32x1K and 64x1K CMSs are compared and the results are shown in Figures 9-10. In this case the experiment proves that the use of lazy update enables decreasing the size of the CMS without affecting the quality of the results. In fact, also with the 32x1K CMS it is possible to achieve around 100% of the exact results.

In the second experiment, we explore the parameters of our design of the lazy update. So we run the same experiment changing the value of the BF and CMS parameters. In particular, we selected three configurations (4x128K,6x64K,8x32K) for the lazy update BF and two configurations (32x1024, 64x512) for the CMS. The parameters are chosen to get insight on the compromise between the bandwidth required by the lazy BF and the memory saving on the lazy BF due to the fewer number

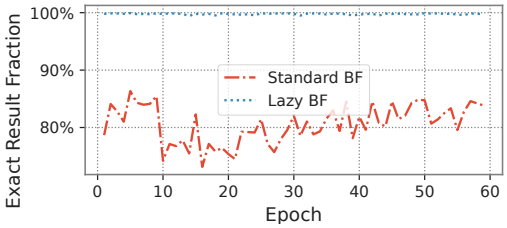

Fig. 5. Percentage of flows with exact result

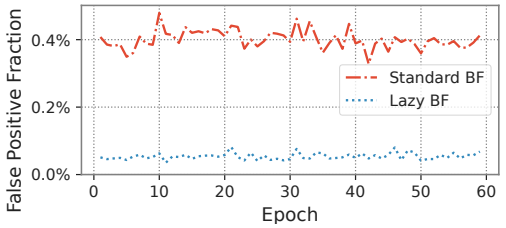

Fig. 6. Flows not detected due to FPs in the BF

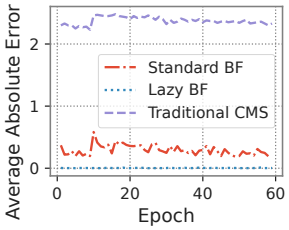

Fig. 7. Average Absolute Error

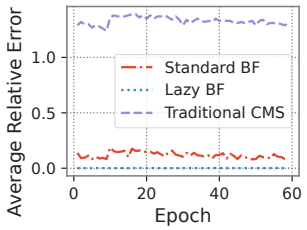

Fig. 8. Average Relative Error

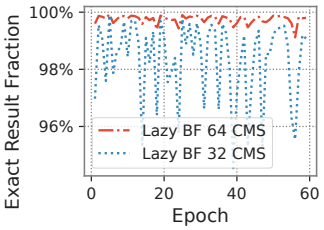

Fig. 9. Percentage of flows with exact result for 32 and 64 CMS and lazy BF

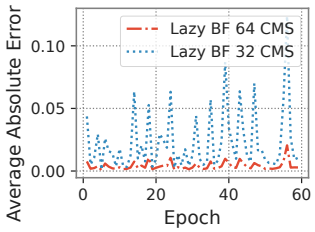

Fig. 10. Average Absolute Error for 32 and 64 CMS

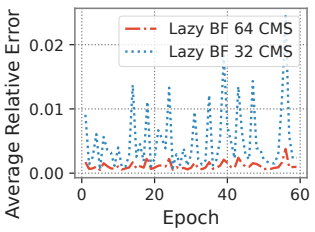

Fig. 11. Average Relative Error for 32 and 64 CMS

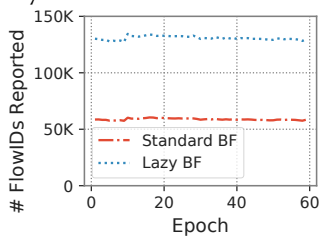

Fig. 12. Bandwidth to the control plane

of bits set to 1 in the lazy BF for the flows with less than k packets. Table 5 reports the false positive rate (FP) and the required bandwidth (BW) for the three lazy BF configurations, the average absolute error (AAE) and the average relative error (ARE) both for the 32x1024 and 64x512 CMS configurations. The results show that the lazy update is able to save 25% of memory (from the 512Kbits of the 4x128K to 384Kbits of the 6x64K) with a negligible penalty for the false positive rate, a 20%-25% of BW overhead and a better value of the AAE and ARE both for the 32 and the 64 CMS configurations. With the 8x32K BF we achieve a saving of 50% with a BW overhead between 25%-35% and a slightly worse AAE.

5.2 Bandwidth and epoch period resolution

One of the possible issues of the FlowLiDAR approach is the need to send to the controller a significant amount of data. In particular, for each epoch it is necessary to send to the controller: (i) all the active flows detected by the BF and, (ii) the snapshot of the CMS stored in the dataplane. Furthermore, if advanced strategies based on the Lazy updates BF or on the BFs pair for differential flow detection are used, also (iii) the snapshot of the BFs must be sent to the controller. Even if the mechanisms to forward these data to the controller can be different, we can suppose that they use the same communication channel such as for example a PCIe connection between the switch and the CPU controller. It is thus important to understand how much bandwidth is required and how the use of different epoch lengths affects this bandwidth. It is obvious that a smaller epoch

Table 5. Analysis of lazy update benefit

metric	k	C1 trace	C2 trace	C3 trace
FP	4	0.0563%	0.0473%	0.0052%
	6	0.0300%	0.0250%	0.0013%
	8	0.1739%	0.1438%	0.0029%
BW (# of flows)	4	130K	129K	72K
	6	154K	151K	88K
	8	163K	160K	97K
AAE (32x1K CMS)	4	0.0194	0.0137	0.00043
	6	0.0031	0.0021	0.00006
	8	0.0202	0.0128	0.00013
AAE (64x512 CMS)	4	0.0176	0.0127	0.00039
	6	0.0030	0.0020	0.00006
	8	0.0186	0.0121	0.00014
ARE (32x1K CMS)	4	0.0052	0.0037	0.00009
	6	0.0008	0.0005	0.00002
	8	0.0060	0.0039	0.00003
ARE (64x512 CMS)	4	0.0047	0.0035	0.00008
	6	0.0007	0.0005	0.00002
	8	0.0057	0.0037	0.00003

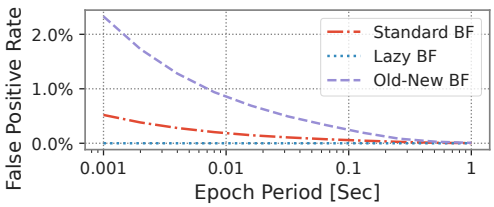

Fig. 13. False positive rate with different BF as a function of the epoch period

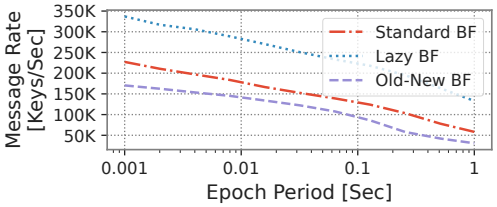

Fig. 14. Number of flows sent to the controller per second as a function of the epoch period

period will provide a better resolution of the network snapshot and thus should be preferable. On the other side, the BW will be directly proportional to the number of epochs in a second, thus at a first look, a too short epoch period could saturate the available bandwidth. However, it is also worth to notice that in a shorter period there will be fewer active flows, thus less pressure to the communication channel. Moreover, a smaller amount of flows also permits to reduce the size of both the BF and the CMS, thus allowing to significantly reduce the overall amount of data to send to the controller.

We performed a set of experiments on the three CAIDA traces to gather insight on the relationship between bandwidth and epoch period. In particular, we selected 10 epoch periods, distributed with a exponential scale between 1ms and 1 second (*i.e* with value 1ms, 2ms, 4ms, 8 ms .. 1024 ms). Also the size of the BF was scaled in the same way, starting from a BF size of 1Kb and doubling the size at each step. This choice allows a fixed contribution to the bandwidth that is of 1Mb/sec ≈ 128KB/sec, which is fairly small. With the above configurations, the FP rate is around 2% for the differential BF, less than 1% for the standard BF, and less than 0.05% for the lazy updates. Figure 13 shows the results in terms of FP rate for the three options taken into account.

The second contribution is due to the size of the CMS. If we target to achieve an exact solution, the CMS load should be less than 0.97 when k=4, thus the number of counters in the CMS should be slightly greater than the number of active flows. Considering a 2B counter, in the best case in which we almost fill the CMS up to 97%, the required bandwidth is directly proportional to the

number of flows sent to the controller. The third contribution is simply the number of active flows that are sent to the controller.

In Figure 14 we reported the amount of flows sent to the controller. It is possible to see that, as expected the number of flows sent per second decreases for higher epoch periods, but also if we want to run FlowLiDAR at a high resolution of 1 ms, the amount of data to send to the controller is still manageable. From the above data, we can estimate the overall required bandwidth as follows. If we suppose that the FlowID is 16B, such as the standard 5-tuple of 104 bits plus some additional information, and considering the 2B for each CMS counter, we can estimate the overall bandwidth for the controller with the standard BF as $C_{BW} = 128KB + (16 + 2) \cdot n_f$. For the lazy updates BF the actual number of flowIDs sent to the controller is $\hat{n}_f > n_f$, while the number of counters in the CMS is reduced, since the lazy updates avoid the insertion in the CMS of the flows with less than 4 packets. The data reported in Figure 14 shows a $\hat{n}_f \approx 1.5 \cdot n_f$. Furthermore, we can estimate a 50% reduction in the CMS size and thus the BW for the lazy updates can be computed as $C_{BW} = 128KB + (24+1) \cdot n_f$. For the case of the differential flow detection we can reduce the amount of flows sent to the controller of around 25%, corresponding to a BW of $C_{BW} = 128KB + (12+2) \cdot n_f$.

The above-presented evaluation shows that the FlowLiDAR system requires a bandwidth of 6.4MB/sec (3.6 MB/sec) in the worst case of 1 ms resolution with lazy updates (differential BFs) and 1.5MB/sec (0.94 MB/sec) in the case of 1 second resolution. Supposing that the connection between the control plane and the data plane is provided by a PCIe interface, also the worst case of 6.4MB/sec is fully sustainable using only a fraction of the available PCIe bandwidth. Even if we consider a 20X speedup to mimic the behavior of a 100 Gbps link, as done as an example in [19], the worst case requires only around 1 Gbps of PCIe throughput. Applying the same speedup, in Figure 13 and Figure 14 the x-axis should be scaled by 20x to estimate the FPR and bandwidth for a fully used 100 Gbps link.

5.3 Comparison with other solutions

We compared our solution with FlowRadar [34], the NZE sketch [23], the PR sketch [49] and the ElasticSketch [54]. If FlowRadar has sufficient memory, it is able to provide an exact result for almost all the monitored flows. Instead, with insufficient memory, the IBLT decoding process fails, no FlowIDs can be recovered, and thus no flow estimation can be done. Therefore, for the comparison between FlowRadar and FlowLiDAR, we estimated the minimum amount of memory needed to achieve 99% of exact results. Instead, for NZE, PR-sketch, and ElasticSketch, we fixed the same amount of memory and evaluated different metrics, namely the required bandwidth, AAE, ARE, and percentage of flows with no error. For FlowLiDAR, we used a lazy BF of (4x128 Kbits) and (32x1Kx16bits) CMSs, with an overall memory of 128 KB. For NZE we used the code of the NZE repository allocating for the BF in the NZE sketch the same size as our lazy BF, and the amount of memory used by our CMS corresponds to the sum of the CMS and hash tables used in the NZE sketch. In detail, we allocate 32KB to the NZE CMS and 32KB to the hash table. For the PR-sketch the configuration is similar to FlowLiDAR: 64KB for the BF and 32Kx16bits for the CMS.

We remark that the sizing of the lazy BF of FlowLidar and of the BF of NZE and PR-sketch is related to the number of undetected flows, *i.e.* flows that are not monitored by the system. In order to monitor more than 99% of flows, for a trace with around 60K flows, we need at least 64KB of memory for the standard BF used in PR-sketch (see equation (1)).

This corresponds to a ratio between BF and CMS different from the one used in the original PR-sketch paper. Using the default ratio of 12.5% leads to a ratio of undetected flows of around 20%.

For ElasticSketch, we allocated 25% of memory to the heavy part (32KB) and 75% of memory to the light part (96KB), following the configuration proposed in [54]. Note that since the original paper of ElasticSketch does not mention the case in which all the monitored FlowIDs are sent to the controller at the end of the measurement epoch, we do not report its BW usage. Following the configuration of [34], FlowRadar requires around 1.4 MB to correctly decode traces C1 and C2, and around 800 KB for trace C3. Thus FlowLiDAR provides a memory saving between 6x and 10x with respect to FlowRadar.

In Figure 15, we show the comparison with NZE, PR-sketch, and ElasticSketch for one trace with an epoch time of 1 second. However, the results are similar for other traces and configurations and consistently show that our solution provides better results at the expense of a slight increase in the number of flowIDs sent to the control plane. In particular, we have much better results both in terms of ARE and AEE and in terms of flows with no error[9]. This is mainly due to two reasons: first of all, in FlowLiDAR, small flows are directly counted by the control plane and do not use the CMS counters; second, we also avoid the use of a hash table in the dataplane, which permits doubling the size of the CMS. Both improve the possibility of extracting the exact results using the resolution method described in section 3.4.

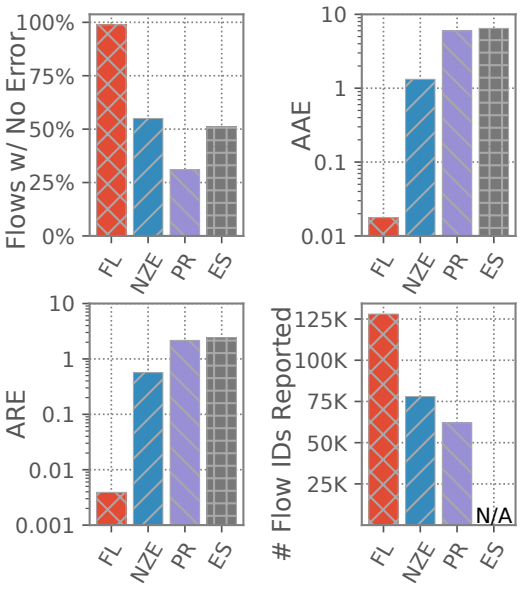

Fig. 15. Comparison between FlowLiDAR (FL), NZE, PR-Sketch (PR), and ElasticSketch (ES)

Finally, since the PR-sketch is conceptually close to FlowLiDAR, we conducted an additional comparison on memory efficiency presented in Figure 16. The fraction of flows tracked without any estimation errors was recorded (data refers to the C1 trace) when varying amounts of memory was allocated for filtering and sketching. The PR-sketch does indeed achieve higher error-free rates as more memory is allocated, but at a much lower rate than FlowLiDAR. As an example, to achieve a target error-free rate of 90%, PR-sketch would require *8x as much memory as FlowLiDAR*. This clearly shows the benefits of the innovations introduced by our design. The better performances of FlowLiDAR over PR-sketch are mainly due to the use of lazy filtering, which reduces the number of monitored flows, thus increasing the memory size range in which

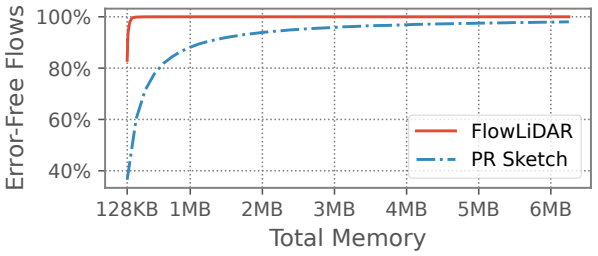

Fig. 16. Comparing FlowLiDAR and PR-sketch in terms of error-free flows at different allocated memory sizes

[9]Note that the performance characteristics of PR-sketch, as well as FlowLiDAR, depends on the overall configuration of the data structure (e.g., the ratio of memory used for filtering vs sketching). Here, we chose the ratio to be the same in both systems to provide a fair comparison.

the exact resolution can be used. However, even in the case of an underdetermined system the use of an ad-hoc approximate resolution provides better results than the least square method used in PR-sketch, as discussed in the next subsection.

5.4 FlowLiDAR approximate resolution

As mentioned in section 3.4.3, when the number of flows is higher than the number of CMS rows, the system is underdetermined, and there are multiple possible solutions. We performed a set of experiments reducing the size of the CMS to understand the quality of the approximate solution provided by Algorithm 6. In particular, we used the lazy update BF with $k = 4$ and reduced the overall memory of the CMS from the value of 512 Kbits (the 32x1K configuration discussed in section 5.1), which is able to provide the exact resolution, down to 32Kbits of a 32x64 CMS configuration. The collected data proves that even with a very small amount of memory, it is possible to have a good estimation of the flow size. We compared our results with the ones achieved using the least square method proposed in PR-sketch[10] and with the estimated values obtained using the standard CMS estimation based on the minimum value. We remark that the use of the least square method simply picks one of the possible solutions (all of them have the L^2-norm equal to zero), thus it does not give any guarantee on the actual error between the proposed solution and the actual values. Instead, our algorithm selects the free variables of the system to fix among the smallest ones, thus minimizing the error between the free variables and the actual values. In Table 6 are reported the achieved percentage of exact values of the different configurations, the AAE and ARE of the FlowLiDAR approximate resolution compared both with the results achieved with the least square method and those obtained using the standard CMS estimation based.

Table 6. Performance of FlowLiDAR approximate CMS resolution

	size (bits)	#exact	#exact lstsq	AAE	AAE lstsq	AAE std	ARE	ARE lstsq	ARE std
32x1K (exact)	512K	99.0%	99.0%	0.019	0.019	3.468	0.0052	0.0052	1.97
32x512	256K	65.0%	60.8%	5.11	5.75	12.9	0.58	1.17	7.57
32x256	128K	63.3%	58.8%	9.11	18.0	42.6	1.32	3.68	24.31
32x128	64K	63.2%	58.8%	21.93	50.4	121	3.94	10.3	68.54
32x64	32K	63.1%	58.8%	45.41	129.9	315	8.57	26.5	177

The table shows that the approximate resolution is still able to provide more than 60% of exact results. These values are mostly due to the use of the lazy update BF that counts the number of packets in the flow based on the number of occurrences of the FlowID sent to the control plane. The main benefit of the approximate resolution appears on the obtained AAE and ARE values, which are significantly smaller than the ones achievable using the standard evaluation of the CMS. In particular, when the memory available for the CMS is very small, the approximate resolution provides a much better estimation than the standard one. For example, comparing the 32x64 configuration that requires 32Kbits, it has around a 3x better AAE and ARE with respect the least square method used in PR-sketch, a 7x better AAE and a 20x better ARE with respect to the one achievable using the traditional CMS.

[10]We can see this as an improved PR-sketch since it first exploits the benefit of the lazy update mechanism and after uses the least square method.

5.5 Processing time for equation solver

Another aspect of the FlowLiDAR implementation to take into account is the processing time needed for CMS equation solving. It is well known that the processing time grows more than quadratically [9] and thus it is important to reduce the size of the CMSs used by FlowLiDAR. On the other hand, the use of smaller CMSs has a slightly negative impact on the probability to exactly solve the CMS equation. In more detail, since a single hash function is used to select the CMS for a given flow, the load factor of each CMS will vary. Therefore some CMSs will be overloaded, and if they go over the resolution threshold the exact CMS resolution will fail. We remark that this is not a dramatic event, since in case of failure FlowLiDAR uses as fallback the estimation of size based on the minimum. To better investigate this aspect, we first perform a set of experiments on an 8 cores, 16 threads Intel i7-10700K CPU clocked at 3.80 GHz to evaluate the processing time of a CMS exact resolution using a single core, varying the number of rows of the system equation. The results are presented in Table 7.

Table 7. Processing Time for Exact CMS resolution

CMS size	256	512	1K	2K	4K
Processing time (ms)	0.6	2.8	15	100	680

From Table 7 we can identify a CMS size that allows a line rate decoding using the parameters selected in the previous section. In particular, for the shortest epoch period of 1 ms, 2 CMS of 256 elements are sufficient to store the active flows in one epoch (that are less than 300 in the three CAIDA traces used in the simulations), and can also be decoded in a time interval less than the epoch period using 2 CPU threads. For longer periods the scenario is less challenging, since we can exploit multiple cores to perform the exact resolution of different CMS equations in parallel. Furthermore, a greater epoch period permits to increase the size of the single CMS. For example, for the 1 second resolution it is possible to deploy 64 CMS of 1K, which can be solved by a single thread in around 960 ms.

6 RELATED WORK

The use of modern switches' programmability to implement flow monitoring has been widely studied both only to identify heavy hitters (see e.g. [5, 40, 55]) or all the flows. This last case is the one we target with our FlowLiDAR scheme. Our scheme, different from existing solutions such as FlowRadar [34], TurboFlow [50], FlowMap [53] or Hashflow [59], does not require storing the flow identifiers in the dataplane memory. Instead, FlowLiDAR sends to the control plane the FlowID of each active flow drastically reducing the amount of memory needed and thus making it possible to monitor a much larger number of flows. The idea of sending the FlowIDs to the control plane was also exploited in the NZE sketch [23] and in the PR-sketch [49]. The NZE sketch splits the traffic in two subsets: the elephants are directly stored in a hash table that provides a key-value map where FlowID and packet counts are stored, while for the mice the FlowID is stored in the control plane and the packet count is based on the use of a sketch structure. However, the NZE sketch requires a careful sizing to split the available memory between the hash table and the sketch and is very sensitive to the FlowID size since larger sizes reduce the number of keys that can be stored in the hash table. The PR-sketch instead sends all FlowIDs to the control plane making its dataplane memory usage independent of the FlowID size. However, it requires a much larger number of counters in the CMS and a larger filter than FlowLiDAR needing thus much more dataplane memory.

FlowRadar [34] uses a Bloom filter to detect new flows and stores the flow identifiers and counters on an Invertible Bloom Lookup Table (IBLT) [45]. The content of the IBLT is periodically sent to

the controller that inverts the table to extract the flows. FlowRadar requires additional memory to ensure that the IBLT can be inverted (around 20%) and also to achieve a low false positive rate on the Bloom filter. Also in FlowRadar, the required memory depends on the flowID size. In terms of accuracy, when the number of flows is below the limit for the IBLT to be invertible, most flows are identified with no errors, while if the number of flows is above the threshold the IBLT completely fails in retrieving the *<flowID, counter>* tuples. Instead, our FlowLiDAR can rely on the approximate resolution when the number of flows exceeds the threshold.

FlowMap [53] introduces some modifications to FlowRadar by replacing the IBLT with an independent hash table that stores only the flow identifiers and a separate two-level hash structure that stores the counters. The extraction in FlowMap is implemented as a linear programming problem that reduces the memory overhead compared to that of an IBLT but is more expensive computationally. To alleviate this issue, the counters are divided in groups by the two-level hashing. This enables solving smaller linear programming problems, which makes the extraction process faster. FlowLiDAR uses a similar approach to split the CMS equations solving.

TurboFlow [50] and HashFlow [59] use hash tables to store the flow identifiers and the associated counters. TurboFlow handles collisions on the hash table by evicting the flows to the controller. HashFlow instead uses two tables such that on a collision on the first table elements are placed on the second, and collisions on the second cause the flow with fewer packets on the second table to be discarded. The observation that network traffic is heavily skewed and many flows have small values is also exploited in Panakos [57], where a mix of bitmap, Count-Min, and SpaceSaving data structures is used. Our choice is to allocate to the control plane the tail of the flow size distribution, reserving data plane memory only for flow with a certain number of packets. Finally, a solution able to provide exact counting in a distributed monitoring system is presented in [11].

7 CONCLUSIONS

In this paper, we have presented FlowLiDAR a scheme to scale switch in data plane flow monitoring to millions of flows while providing flow size estimates that are exact with high probability. FlowLiDAR uses a Bloom filter to identify new flows and a sketch to estimate their size and introduces several innovations. The first one is not storing the FlowIDs in the data plane but sending them to the controller, so reducing the data plane memory needed. Additionally, the sketch to provide estimates is solved as a linear programming problem to improve accuracy and finally a lazy update mechanism is used in the Bloom filter that reduces the false positives and the number of flows stored in the sketch. These techniques are efficiently combined to drastically reduce the amount of data plane memory while achieving excellent accuracy. FlowLiDAR has been implemented and compared with state of the art alternatives. The results show that FlowLiDAR requires at least 6x less memory than FlowRadar. Compared with NZE, the PR-sketch and the Elastic Sketch configured with the same amount of memory, FlowLiDAR improves the AAE and ARE by up to 10x, 100x and 100x, respectively. Moreover, while FlowLiDAR is able to successfully track 98.7% of existing flows, other techniques can only reconstruct up to 60% of them.

8 ACKNOWLEDGEMENTS

We thank our shepherd, Arpit Gupta, and the anonymous reviewers, for valuable comments and feedback. The work of Pedro Reviriego was partly done while he was at Universidad Carlos III de Madrid and was supported by the FUN4DATE project, PID2022-136684OB-C22, and the ENTRUDIT project TED2021-130118B-I00, funded by the Spanish Agencia Estatal de Investigación (AEI) 10.13039/501100011033 and by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101016663 (B5G-OPEN).

The work of Gianni Antichi was partially supported by the UK EPSRC project EP/T007206/1 and by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on “Telecommunications of the Future” (PE00000001 - program “RESTART”)

REFERENCES

[1] Anurag Agrawal and Changhoon Kim. 2020. Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 1–32.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[3] Gianni Antichi and Gábor Rétvári. 2020. Full-Stack SDN: The Next Big Challenge?. In *Symposium on SDN Research (SOSR)*. ACM.

[4] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. 2021. Leveraging Service Meshes as a New Network Layer. In *Workshop on Hot Topics in Networks (HotNets)*. ACM.

[5] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2020. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1172–1185.

[6] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-band Network Telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 662–680.

[7] Giuseppe Bianchi, Elisa Boschi, Simone Teofili, and Brian Trammell. 2010. Measurement data reduction through variation rate metering. In *2010 Proceedings IEEE INFOCOM*. IEEE, 1–9.

[8] Caida. 2016. The CAIDA UCSD Anonymized Internet Traces. http://www.caida.org/data/passive/passive_2016_dataset.xml.

[9] Timothy A Davis, Sivasankaran Rajamanickam, and Wissam M Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. *Acta Numerica* 25 (2016), 383–566.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon’s Highly Available Key-Value Store. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery.

[11] Vitalii Demianiuk, Sergey Gorinsky, Sergey I Nikolenko, and Kirill Kogan. 2020. Robust distributed monitoring of traffic flows. *IEEE/ACM Transactions on Networking* 29, 1 (2020), 275–288.

[12] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. 2010. Tight Thresholds for Cuckoo Hashing via XORSAT. In *Proc. 37th ICALP (1)*. 213–225. https://doi.org/10.1007/978-3-642-14165-2_19

[13] Olivier Dubois and Jacques Mandler. 2002. The 3-XORSAT Threshold. In *Proc. 43rd FOCS*. 769–778. https://doi.org/10.1109/SFCS.2002.1182002

[14] Cristian Estan and George Varghese. 2002. New Directions in Traffic Measurement and Accounting. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[15] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frederic Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *Conference on Analysis of Algorithms (AofA)*.

[16] Nikolaos Fountoulakis and Konstantinos Panagiotou. 2012. Sharp Load Thresholds for Cuckoo Hashing. *Random Struct. Algorithms* 41, 3 (2012), 306–333. https://doi.org/10.1002/rsa.20426

[17] Jiaqi Gao, Nofel Yaseen, Robert MacDavid, Felipe Vieira Frujeri, Vincent Liu, Ricardo Bianchini, Ramaswamy Aditya, Xiaohang Wang, Henry Lee, David Maltz, Minlan Yu, and Behnaz Arzani. 2020. Scouts: Improving the Diagnosis Process Through Domain-Customized Incident Routing. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[18] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. 2016. Fast Scalable Construction of (Minimal Perfect Hash) Functions. In *Proc. 15th SEA*. 339–352. https://doi.org/10.1007/978-3-319-38851-9_23

[19] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. 357–371.

[20] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings*

of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM). ACM.

[21] George Havas, Bohdan S. Majewski, Nicholas C. Wormald, and Zbigniew J. Czech. 1993. Graphs, Hypergraphs and Hashing. In *Proc. 19th WG*. 153–165. https://doi.org/10.1007/3-540-57899-4_49

[22] J. Hill, M. Aloserij, and P. Grosso. 2018. Tracking Network Flows with P4. In *IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*.

[23] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. 2021. Toward Nearly-Zero-Error Sketching via Compressive Sensing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 1027–1044.

[24] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. 2020. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 404–421.

[25] Intel. 2021. Intel Deep Insight Network Analytics Software. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/network-analytics/deep-insight.html. Accessed: 2022-10-04.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Symposium on Operating Systems Principles (SOSP)*. ACM.

[27] Piotr Jurkiewicz, Grzegorz Rzym, and Piotr Boryło. 2021. Flow length and size distributions in campus Internet traffic. *Computer Communications* 167 (2021), 15–30. https://doi.org/10.1016/j.comcom.2020.12.016

[28] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Symposium on SDN Research (SOSR)*. ACM.

[29] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*.

[30] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 90–106.

[31] Abhishek Kumar, Minho Sung, Jun (Jim) Xu, and Jia Wang. 2004. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. In *Special Interest Group for the Computer Performance Evaluation (SIGMETRICS)*. ACM.

[32] Jan Kučera, Ran Ben Basat, Mário Kuka, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. 2020. Detecting Routing Loops in the Data Plane. In *Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM.

[33] Gene Moo Lee, Huiya Liu, Young Yoon, and Yin Zhang. 2005. Improving sketch reconstruction accuracy using linear least squares method. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*. 24–24.

[34] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*.

[35] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM.

[36] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[37] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*.

[38] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo. 2019. Optimizing Bloom Filter: Challenges, Solutions, and Comparisons. *IEEE Communications Surveys and Tutorials* 21, 2 (2019), 1912–1949.

[39] Bruce M Maggs and Ramesh K Sitaraman. 2015. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review* 45, 3 (2015), 52–66.

[40] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory*. Springer, 398–412.

[41] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 15–28.

[42] Michael Molloy. 2005. Cores in random hypergraphs and Boolean formulas. *Random Struct. Algorithms* 27, 1 (2005), 124–135. https://doi.org/10.1002/rsa.20061

[43] F. Pereira, N. Neves, and F. M. V. Ramos. 2017. Secure network monitoring using programmable data planes. In *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*.

[44] Boris Pittel and Gregory B. Sorkin. 2016. The Satisfiability Threshold for k-XORSAT. *Combinatorics, Probability & Computing* 25, 2 (2016), 236–268. https://doi.org/10.1017/S0963548315000097

[45] Salvatore Pontarelli, Pedro Reviriego, and Michael Mitzenmacher. 2014. Improving the Performance of Invertible Bloom Lookup Tables. *Inf. Process. Lett.* 114, 4 (apr 2014), 185–191. https://doi.org/10.1016/j.ipl.2013.11.015

[46] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[47] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostic, and Marco Chiesa. 2023. A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches. In *Networked Systems Design and Implementation (NSDI)*. USENIX.

[48] Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter A. Dinda, Ming-Yang Kao, and Gokhan Memik. 2007. Reversible Sketches: Enabling Monitoring and Analysis over High-Speed Data Streams. In *Transactions on Networking, Volume: 15, Issue: 5*. IEEE Press.

[49] Siyuan Sheng, Qun Huang, Sa Wang, and Yungang Bao. 2021. PR-Sketch: Monitoring per-Key Aggregation of Streaming Data with Nearly Full Accuracy. *Proc. VLDB Endow.* 14, 10 (jun 2021), 1783–1796. https://doi.org/10.14778/3467861.3467868

[50] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference*. 1–16.

[51] Daniel Ting. 2018. Count-min: Optimal estimation and tight error bounds using empirical error distributions. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2319–2328.

[52] Nguyen Van Tu, Jonghwan Hyun, and James Won-Ki Hong. 2017. Towards onos-based sdn monitoring using in-band network telemetry. In *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 76–81.

[53] Xiong Wang, Hanyu Liu, Jun Zhang, Jing Ren, Sheng Wang, and Shizhong Xu. 2019. FlowMap: A Fine-Grained Flow Measurement Approach for Data-Center Networks. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 1–7.

[54] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*.

[55] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: an accurate algorithm for finding Top-k elephant flows. *IEEE/ACM Transactions on Networking* 27, 5 (2019), 1845–1858.

[56] Minlan Yu. 2019. Network telemetry: towards a top-down approach. *ACM SIGCOMM Computer Communication Review* 49, 1 (2019), 11–17.

[57] Fuheng Zhao, Punnal Ismail Khan, Divyakant Agrawal, Amr El Abbadi, Arpit Gupta, and Zaoxing Liu. 2023. Panakos: Chasing the Tails for Multidimensional Data Streams. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1291–1304.

[58] Qi Zhao, Abhishek Kumar, and Jun Xu. 2005. Joint Data Streaming and Sampling Techniques for Detection of Super Sources and Destinations. In *Conference on Internet Measurement (IMC)*. USENIX Association.

[59] Zongyi Zhao, Xingang Shi, Zhiliang Wang, Qing Li, Han Zhang, and Xia Yin. 2021. Efficient and Accurate Flow Record Collection With HashFlow. *IEEE Transactions on Parallel and Distributed Systems* 33, 5 (2021), 1069–1083.

[60] Yu Zhou, Jun Bi, Tong Yang, Kai Gao, Jiamin Cao, Dai Zhang, Yangyang Wang, and Cheng Zhang. 2020. Hypersight: Towards scalable, high-coverage, and dynamic network monitoring queries. *IEEE Journal on Selected Areas in Communications* 38, 6 (2020), 1147–1160.

[61] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. 2020. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 76–89.

[62] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*. 479–491.

Received August 2023; revised October 2023; accepted October 2023