Sequential Zeroing: Online Heavy-Hitter Detection on Programmable Hardware

Turkovic, Belma; Oostenbrink, Jorik; Kuipers, Fernando; Keslassy, Isaac; Orda, Ariel

Publication date
2020

Document Version
Accepted author manuscript

Published in
IFIP Networking 2020 Conference and Workshops, Networking 2020

Sequential Zeroing: Online Heavy-Hitter Detection on Programmable Hardware

Belma Turkovic[1], Jorik Oostenbrink[1], Fernando Kuipers[1], Isaac Keslassy[2], and Ariel Orda[2]
[1]Delft University of Technology, The Netherlands
[2]Technion, Israel
Email: {B.Turkovic-2, J.Oostenbrink, F.A.Kuipers}@tudelft.nl, {isaac, ariel}@ee.technion.ac.il

***Abstract*— Flows that have exceeded a given percentage of the last sliding window of N packets, denoted as *heavy-hitter flows*, require special handling, since they may disrupt the service of other flows or may be indicative of malicious traffic. However, even when equipped with a programmable switch, it is unclear how to detect heavy hitters on a per-packet basis, while obeying the stringent switch memory access rates. For instance, existing solutions, such as HashPipe, cannot detect heavy hitters without halving the line rate and do not support sliding windows.**

To the best of our knowledge, this paper is the first to present heavy-hitter detection solutions that provide per-packet granularity at line-rate performance. We realize this by introducing (1) *Modulo sketching*, a novel counting algorithm that reuses counters and limits the impact of smaller flows beyond early processing stages; and (2) *Sequential Zeroing*, a new approach to extending interval-based schemes to sliding window measurements. Our solutions are extensively evaluated, both via simulations and experiments on a Netronome SmartNIC, and demonstrate significant performance gains over the state-of-the-art.

I. Introduction

This paper is about providing the ability to detect, online and at high line-rate, whether each packet going through a programmable switch is part of a heavy-hitter flow, which is a flow that has exceeded a threshold number of packets in the sliding window of the last N packets. For network operators, such an ability is of crucial importance to enable fine-grained Denial-of-Service (DoS) mitigation, traffic anomaly detection, flow-size-aware routing, Quality-of-Service (QoS) management, and load balancing [4]–[6], [22].

Scope. We consider enabling per-packet heavy-hitter detection in the data plane of programmable switches, which rely on network programming languages such as P4 [8]–[10], [25]. P4 defines registers, *i.e.,* stateful memory blocks that the switch can read from, modify, and/or write to, while packets are being processed. When deployed in a network, a P4 program works in coordination with a control plane, which configures the run-time rules and action variables. P4-programmable hardware can be classified as having either (1) *local memory*, as in Barefoot [24], where packets are processed through a pipeline of several hardware stages. Each stage has its own separate memory and processing resources. To maintain a high processing throughput, typically only one read-modify-write action is allowed per register array; or (2) *shared memory*, as in Netronome [1], where concurrent memory accesses to the same register array are allowed. Since memory accesses consume most of the processing cycles in programmable hardware and may lead to race conditions, they should be minimized. In this paper, we formulate our algorithms so they could be implemented in both hardware models.

Motivation. Common heavy-hitter detection algorithms (*e.g.,* Space-Saving [18], CSS [6], WCSS [6], and Memento [4]) violate programmable hardware constraints, as they are not organized in consecutive simple stages, require too many memory accesses per processed packet and/or use actions not supported by programmable hardware. Simple sketching schemes, like Count-Min (CM) [13], have no mechanism to count over the sliding window of the last N packets, and in fact do not even provide a clear implementation for counting over periodic intervals, as they do not contain a mechanism to simultaneously reset the whole data structure online [26]. Even existing data-plane solutions that were developed for P4, such as HashPipe [22] and PRECISION [5], come with several limitations: most significantly, (1) they also have no mechanism to count over sliding windows; and (2) even when counting over periodic intervals, they cannot compute online a count estimate for each packet, unless they recirculate each packet through the pipeline twice, thereby halving the line rate (see Appendix). In addition, (3) they also do not provide a simultaneous memory flushing implementation for counting over periodic intervals; and (4) they intrinsically need flow identifiers for each counter, thus requiring additional memory.

Contributions. *We present a body of solutions for heavy-hitter detection on programmable network hardware, in which we aim at minimizing the false-positive and false-negative rates.*

In Sec. II, we start by considering the easier problem of detecting heavy hitters over a fixed *interval* of N packets. To do so, we introduce *Modulo Sketching*, a new sketching approach. Its most salient feature is that it relies on *conditional sketching*, a sketching approach that uses several consecutive stages of counter arrays and attempts to filter out the non-heavy-hitter packets by stopping them at the first stages. As a result, heavy-hitter packets are nearly the only ones to reach the last stages, thus (1) reducing potential collisions between different flows, which in turn reduces the need for a large

memory size; and also (2) reducing memory access rates, since non-heavy-hitter packets almost never access later stages. Therefore, this approach is particularly adapted to the common pipeline structure of programmable switches. It stands in contrast to previous P4-based algorithms like HashPipe and PRECISION, which need to go through all stages in order to evaluate the size of a packet's flow (see Appendix for details).

Next, in Sec. III, we consider the case of heavy hitters over *sliding windows*, which is our main goal. We attempt to leverage our interval-based Modulo sketch by generalizing it to sliding windows. Unfortunately, unlike intervals, sliding windows also need deletions, which introduce additional memory accesses. In addition, implementing a perfect sliding window would consume a large portion of the limited switch memory, as it would require us to remember the full packet order. We thus suggest two different approaches to efficiently approximate a sliding window: (1) *Sequential Window*, which relies on control-plane intervention, and (2) *Zeroing Window*, an in-data-plane counter zeroing technique. We finally combine both to yield the *Sequential Zeroing* algorithm in the data plane.

Finally, in Sec. IV, we evaluate the performance of our new algorithms and demonstrate how they outperform existing approaches. We run our evaluations both through simulations and through experiments on a Netronome SmartNIC. In particular, we illustrate on CAIDA traces how our final schemes can achieve negligible false-negative rates and low false-positive rates while providing an estimation for each packet at line rate using the data plane, even when assuming a small memory consumption of 55kB.

II. Interval Measurement

As a first step towards our goal of determining heavy hitters over the last N packets, we look at the easier problem of detecting heavy hitters over a fixed *interval* of N packets. In other words, our initial goal is to determine for each incoming packet whether its flow has exceeded a threshold number H of packets within this interval.

We would like to obtain a heavy-hitter detection scheme that satisfies two major criteria: (1) it should not consume too much memory and (2) it should have a low memory access rate. In addition, as commonly considered in the literature, we assume that false negatives (failures to detect heavy-hitter flows) carry a higher penalty than false positives [19], [21].

Conditional sketching. To satisfy these criteria, we introduce the concept of *conditional sketching*, a sketching approach that relies on several consecutive pipelined stages of counter arrays. Since we want to reduce the overall memory access rate, our key idea is to stop non-heavy-hitter packets in early stages. Thus the overall number of operations is significantly reduced, as heavy-hitter flows are nearly the only flows to reach the last stages.

Modulo Sketch. We introduce the *Modulo sketch* algorithm to efficiently implement the concept of conditional sketching. As mentioned, each packet goes through several stages to update its counters. However, following conditional sketching, most packets will stop updating counters at the early stages of the stage pipeline. The packets may then continue over several additional stages without any counter update simply to estimate the packet's flow size.

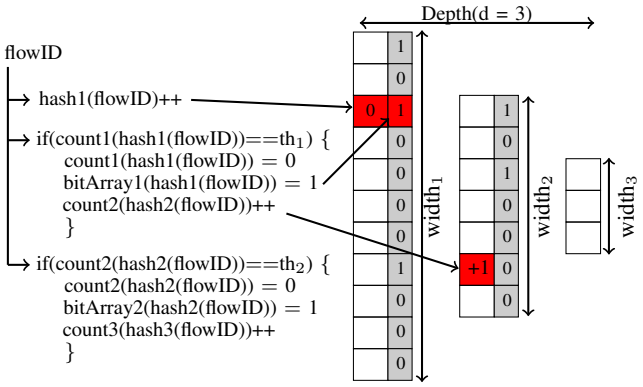


Fig. 1. *Modulo* sketch with $d = 3$ stages. The flowID of an incoming packet first hashes into a first-stage (red) counter and increments it. If the counter value is $th_1 - 1$ and it is incremented to $th_1 \equiv 0 \mod th_1$, we reset the counter, set its associated bit to 1, hash the flowID into a second-stage (red) counter, and increment that counter as well. If the result is below the threshold th_2, we stop here. As the counter's associated bit is 0, there is no need to continue to the third stage, and we can evaluate flowID as *not* a heavy hitter.

Intuitively, Modulo sketch works like a clock, as it zeroes the counter of the *seconds* (first stage) when incrementing the counter of the *minutes* (second stage). It proceeds in the same way by resetting the *minutes* (second stage) when incrementing the *hours* (third stage), and so on. Thus, Modulo manages to increase its efficiency by both (1) stopping most counter updates in early stages and (2) reusing the counters.

Architecture. As Fig. 1 illustrates, the Modulo sketch consists of d successive stages of counters, where d is the depth of the sketch. Each stage i holds $width_i$ counters of c_i bits each. We define the threshold th_i of each stage $1 \leq i \leq d-1$ such that

$$H > \prod_{i=1}^{d-1} th_i, \tag{1}$$

where H is the heavy-hitter threshold that was defined above.

In addition, to enable the conditional sketching, we introduce a bit array at each stage but the last, *i.e.,* each counter in stage $1 \leq i \leq d-1$ is provided an additional initially-null bit that determines whether the packet should continue to the next stage.

Algorithm. As shown in Fig. 1, upon a packet's arrival, its flowID is first hashed onto a counter in the first stage. It then increments this counter. Next, for any stage $1 \leq i \leq d-1$ but the last, the first time that the resulting counter value reaches $th_i \equiv 0 \mod th_i$, we set its associated bit to 1, indicating that the threshold has been reached at least once, and therefore that all subsequent packets hashing to this counter should continue to the next stage $i+1$ in order to estimate their flow size. Therefore each packet goes through all stages with a set bit, until it reaches a stage where it hashes to a counter with a null bit, in which case it can stop. Since $H > \prod_{i=1}^{d-1} th_i$ (Eq. (1)), any packet that stops before the last stage is considered as

a non-heavy-hitter. More generally, the flow size of a packet that sees a counter value v_i at each stage i can be estimated as $v_1 + th_1 \cdot (v_2 + th_2 \cdot (\cdots + th_{d-1} \cdot v_d))$, which needs to be compared to H.

Threshold details. To fully utilize all the bits of all the counters, we define the threshold th_i of each stage $1 \leq i \leq d-1$ to be $th_i = 2^{c_i}$. We also allocate enough bits to the last-stage counters so that they never overflow even if a single flow uses N packets, *i.e.,* $c_d = \left\lceil \log_2 \left(\frac{N+1}{\prod_{i=1}^{d-1} th_i} \right) \right\rceil$ bits per counter.

Properties. The main advantage of Modulo is its *reduced memory consumption* at high scales. If N packets are added to the sketch, at most N/th_1 packets reach the second stage to update it. More generally, at most $N/\prod_{j=1}^{i-1} th_j$ packets will reach stage i. Therefore we have an *exponentially decreasing* load further down the stages, yielding a particularly scalable architecture. For instance, when doubling the window size N and the heavy-hitter threshold H, Modulo would only need to double a single threshold. Again, we can increase the width of the first stages at the expense of smaller widths for the late stages.

Note that the Modulo sketch presents the drawback of allowing a small number of false negatives. Specifically, the packet that resets a counter is the one that increments the next-stage counter. For instance, a first flow F_1 may increment a first-stage counter to 63, but then the packet of another flow F_2 may arrive, reach the threshold of 64, reset this counter, and increment its hashed second-stage counter, thus in a sense *stealing* the entire counter value of 64.

III. Sliding window Measurement

The goal of this paper is to compute online heavy hitters over *sliding windows*. Since in the previous section, we considered the problem over *intervals* as a first step, we would now like to provide ways to generalize interval-based sketching schemes to sliding windows.

However, sliding windows involve both additions and deletions at each packet arrival, and therefore cause many challenges to overcome: (1) we want to delete the last packet from the counting structure without keeping in memory the list of packets, and therefore without remembering what the last packet is, as this would significantly increase the memory consumption and the number of memory accesses; (2) the logic of a conditional sketch-based structure like Modulo breaks down with deletions: *e.g.,* we mentioned above the example of a flow F_1 contributing 63 packets out of a counter threshold of 64, and another flow F_2 contributing the last packet and consequently incrementing its hashed second-stage counter. If we want to delete an F_1 packet from the structure, we would not know how to update the second-stage counters; (3) finally, the counter increments (due to packet arrivals) and decrements (due to packet deletions) are not allowed to occur in two different counters of the same stage, since there is a bound of one memory access per stage, and therefore there needs to be some scheduling of the memory accesses.

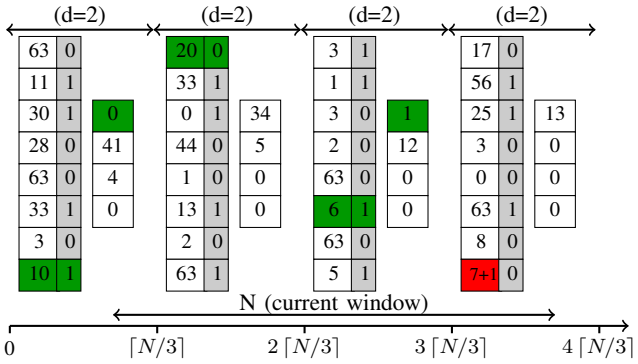


Fig. 2. *Sequential Window* algorithm with Modulo sketch, using $k = 3$. The Sequential Window architecture comprises $k+1 = 4$ sub-intervals, each implementing a Modulo sketch with $d = 2$ stages for $\lceil N/k \rceil$ packets, thus covering the entire window of N packets. An incoming packet would only update its count in the last sketch (in red), while reading and summing its count estimate from all 4 sketches (in green and red).

First, in Sec. III-A we generalize interval-based approaches to support the sliding window concept. Next, in Sec. III-B we propose a sliding window approach to reset the counting sketches directly in the data plane, while the packets are processed. Finally, we combine the advantages of the two previously described approaches to create a sliding window solution that can maintain accuracy over time without any intervention from the control plane.

A. Sequential Window

Architecture. As Fig. 2 illustrates, given some integer parameter k, our *Sequential window* scheme periodically implements an interval-based sketching scheme such as Modulo or Count-Min of depth d every sub-interval of $\lceil \frac{N}{k} \rceil$ packets. Therefore, each window of N packets can be covered by at most $k+1$ consecutive interval sketches.

Algorithm. At the first incoming packet, we start by counting in the first counting sketch corresponding to the first sub-interval. Then, every $\lceil \frac{N}{k} \rceil$ packets, we keep advancing to the next counting sketch. When we are done with the last sketch, we can reset the first one using the control plane, since it counts packets that were received over N packets ago. Every $\lceil \frac{N}{k} \rceil$ packets, we then keep resetting the next sketch in the sequence to start a fresh sketch for the next sub-interval. Finally, to estimate a count for a given packet, we simply sum the estimates provided by all the sub-intervals (shown in green in Fig. 2).

Threshold details. Since we scaled down the N-packet intervals to $\lceil \frac{N}{k} \rceil$-packet sub-intervals, we want to similarly scale down the thresholds, and therefore update Eq. (1) by using a locally-scaled-down heavy-hitter threshold $\lfloor H/k \rfloor$.

Properties. The Sequential Window approach has several advantages. First, by avoiding any deletions and including enough sub-intervals to cover the entire sliding window of N packets, the Sequential Window scheme does not introduce additional false negatives.

Second, the value k helps to trade off the performance against the total number of stages, thus targeting different

hardware platforms as well as different window sizes. For instance, increasing k will decrease the number of packets processed by each sub-interval, and therefore reduce the number of unique flows and the number of hash collisions in each stage, thus increasing accuracy. Also it will decrease the number of packets that are counted outside the window, and therefore further decrease the number of false positives. On the other hand, it will also need more stages in the implementation.

Third, the Sequential Window approach is easily implementable on any type of programmable hardware. As illustrated in Fig. 2, it only requires up to $d \cdot (k+1)$ memory accesses per packet, *i.e.,* $d \cdot (k+1) - 1$ reads and 1 read-modify-write. Thus, on hardware with shared memory, e.g. Netronome, the sketch can easily be tuned by the choice of k and d to avoid a drop in throughput. Moreover, for other types of hardware, it does not violate the one-access-per-stage rule.

However, there are several disadvantages to this approach. First, the total number of consumed stages increases by a factor of $k+1$ compared to any interval-based sketch: $(k+1) \cdot d$ stages in total. Second, by counting over a larger window than the actual sliding window, we introduce false positives. Third, and most significantly, all the counters in the oldest sub-interval need to be reset at the same time *using the control plane*. Thus, this approach is not entirely done in the data plane. This control-plane resetting may be an issue on a fast link with a small window. For instance, Barefoot Tofino switches can process packets at 6.5Tbps. Assuming a window of $N = 2^{16}$ minimally-sized packets of 64B and $k+1 = 8$, then every 81ns the hundreds or thousands of counters of a sub-interval would need to be reset, which at best increases the processing resources from the control plane used by the algorithm and at worst is simply impossible, depending on the hardware platform.

B. Zeroing Window

Overview. We now look for an alternative way of transforming an interval-based sketch like Modulo or Count-Min into a sliding-window-based sketch. A significant challenge is that we need to delete old packets that are not in the sliding window anymore, but on the other hand we do not want to allocate memory space to remember old packets. Instead, our first key idea is *to loop through all counters and zero them once every N packets*, thus ensuring that packets that have left the sliding window do not influence our counts anymore. In addition, our second key idea is to make sure that we can do it in the *data plane*, and do not require the massive intervention of the control plane anymore.

Initial algorithm. Our *Zeroing Window* algorithm is relatively straightforward. Consider a given interval-based sketching algorithm like Modulo or Count-Min. Then, for each stage i of width $width_i$ in the sketching algorithm, Zeroing Window defines a zeroing period $m_i = \lfloor N/width_i \rfloor$, and essentially resets the next counter at every m_i^{th} packet that is added to the sketch. Specifically, it resets counter j at packets $j \cdot m_i \mod N$. For instance, if $N = 2^{16} = 65,536$ and $width_i = 1,000$, it resets the first counter at packet 65 of the window, the second counter at packet 130, and so on, until the last counter at packet 65,000. It then resets the first counter again at packet $N + 65$, etc.

Data-plane implementation. The above algorithm is simple, but it violates our rule that each stage should be accessed at most once per packet, since it may want to increment a counter as well as reset another one within the same stage. Therefore, as Fig. 3(a) illustrates, we suggest a data-plane implementation of the Zeroing Window algorithm. We split each stage into two equal-sized and independent sub-stages. Then, we want to apply the same zeroing scheme to each sub-stage. Assuming that the hashing functions are uniformly distributed, each sub-stage is only accessed at most half the time by the inserted packets. Therefore, whenever a counter needs to be reset, it can simply wait for the next time its sub-stage is free. In the worst case this waiting time is unbounded, and in the case where all packets are independent it is a geometrically distributed variable of expected value below 2. In practice, we never encountered any issue with this waiting time.

Properties. The main benefit of the Zeroing Window scheme is that it manages to operate in the data plane. However, because it resets arbitrary counters in the sketches, its disadvantage is that it also significantly increases the false negative rate, which we consider as more costly than the false positive rate. Therefore, we also consider several ways of reducing these false negatives for different sketching algorithms, at the expense of increasing the false positives:

Zeroing the Count-Min sketch. When removing values from the CM-sketch, or as in our case resetting some of the counters to 0, one should apply the median instead of the minimum [13]. For instance, if the $d = 3$ counts are 100, 110 and 120, and 120 is zeroed, the median estimate of 100 is more accurate than the minimum estimate of 0.

Zeroing the Modulo sketch. We now do not stop at the first time that a bit is set to 0 (Modulo sketch), but estimate the flow size using the counts over all stages.

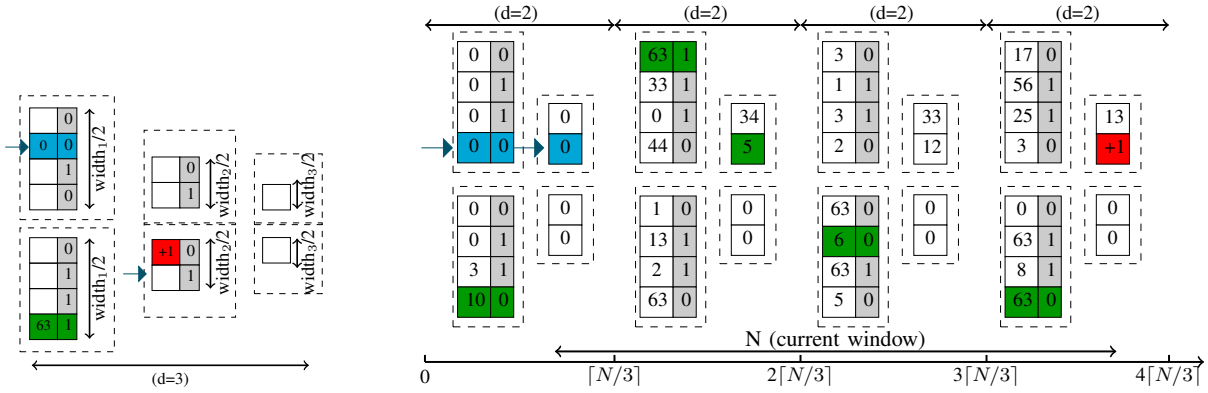
C. Sequential Zeroing: Zeroing the Sequential Window.

We finally introduce a last scheme, denoted *Sequential Zeroing*, which combines the Zeroing approach with the previously-described Sequential window. As Fig. 3(b) illustrates, Sequential Zeroing removes outdated flow counts from the last sub-interval of Sequential Window by applying the Zeroing Window algorithm to this sub-interval. Namely, it can do it automatically in the data plane, without any intervention from the control plane, by splitting its stages into sub-stages and applying the scheduling described above.

IV. Evaluation

A. Experiment setup

We conducted our evaluations, first using simulations in Python, and then experiments with a Netronome Agilio CX SmartNIC [1].



(a) Zeroing window for the Modulo sketch.

(b) Sequential Zeroing.

Fig. 3. Illustrations of the *Zeroing Window* technique that resets counters in the dataplane. **(a)** Zeroing Window for a Modulo sketch of depth $d = 3$ stages. Each stage is subdivided into 2 equal-size sub-stages (in dashed rectangles), and we are only allowed one memory access per sub-stage. In addition, assume we want to reset the counters with blue arrows. The incoming packet checks its hashed (green) counter in the first stage, which is equal to the first-stage threshold $th_1 - 1 = 63$, and then resets it and increments a (red) second-stage counter. As a result, we can reset the (blue) first-stage counter in the top sub-stage, since there is no memory access to this sub-stage. However, we cannot reset the second counter with an arrow, and therefore will wait for the next time that this sub-stage is not accessed by a packet. **(b)** *Sequential Zeroing* scheme, which applies the Zeroing Window technique to the last sub-interval of the Sequential Window. We use the example of Fig. 2 with $k + 1 = 4$ sub-intervals, each implementing a Modulo sketch of depth $d = 2$. Again, each stage is subdivided into 2 equal-size sub-stages (in dashed rectangles). Assume we want to reset the last (blue) counters of both of the top sub-stages in the left sub-interval that contains the oldest packets. We can indeed do so without violating memory constraints.

Hashing. The 5-tuple flow identifier (flowID) consists of the source IP, destination IP, layer 4 protocol, source port, and destination port. All our implementations used the CRC16 hash function. Different hash functions were created by appending seed values to the flow identifiers.

Traces. We classified heavy hitters as flows whose packet counts were above a threshold $H = \lfloor N/1000 \rfloor$, initially considering an interval size of $N = 2^{16}$ packets. Packets were obtained from (1) 40 different traces collected from an ISP backbone link at the Equinix data-center in Chicago in January 2016, made available by CAIDA [23], and (2) 10 different traces collected from university campus data centers (UNI1 and UNI2 dataset), made available by [7]. We observed similar results using both datasets, even though they have significantly different flow-size distributions (the university traces have more heavy hitters), and therefore only present the CAIDA results due to space constraints. In the hardware evaluation, all the traces were replayed at the rate of N packets per second.

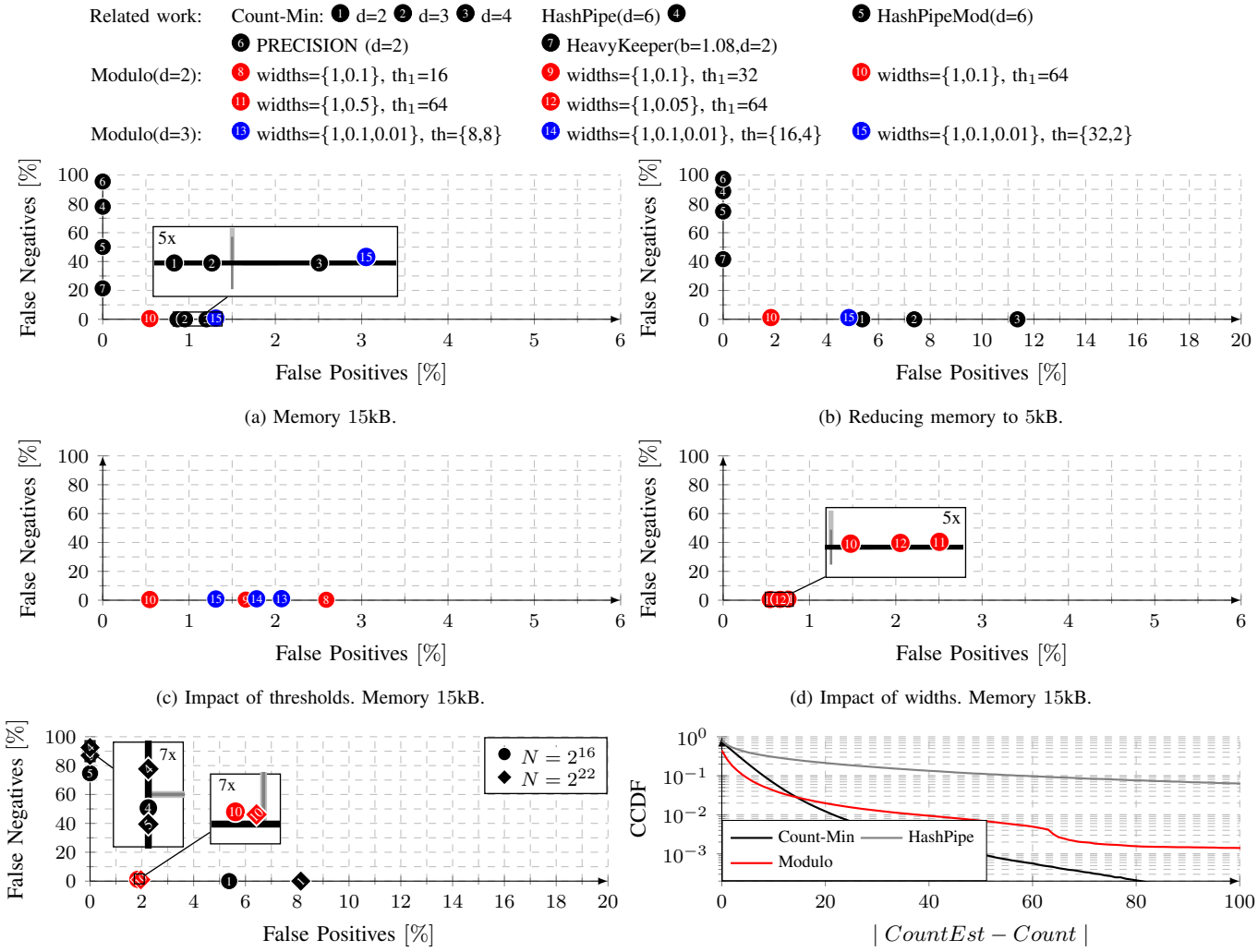
Metrics. We evaluated all schemes on the percentages of false negatives (percentage of heavy hitter packets that are not reported) and false positives (percentage of non-heavy hitter packets that are reported). As previously mentioned, following literature, we assume that the penalty of false negatives is significantly higher than that of false positives. We also measured the distribution of the absolute count estimation error.

Comparison baselines. For interval-based evaluations, all of our solutions were compared against the following baseline solutions: (1) *Count-Min (CM)* sketch [13], (2) *HashPipe* [22], (3) *HashPipeMod* which implements HashPipe using 2B flowID fingerprints rather than 13B flowIDs (see Appendix), (4) *HeavyKeeper* [28], and (5) PRECISION [5]. For the sliding-window evaluations, since we are not aware of any other P4 solution that implements sliding windows, we considered a set of solutions that combine all interval-based solutions with a periodic resetting of all the counting stages every N packets. In all evaluations, we fixed a given allocated total memory for a fair comparison.

B. Interval measurements

Memory vs. performance. Fig. 4(a) and 4(b) show the performance of our solutions for two different memory quotas. As expected, assigning more memory improves the accuracy for all solutions, as the width of the counter arrays can be increased, which reduces the number of hash collisions. We can see how for small memory allocations, other approaches start to break down, while the Modulo sketch manages to achieve acceptable heavy hitter detection. The main reason is a more efficient memory usage: the total number of counters used in our scheme is much higher than in the other approaches, since our counters do not need to count up to N and therefore are smaller. This memory efficiency makes our solution uniquely suitable for programmable hardware. While not included due to a lack of space, we found that our solutions still outperformed the state of the art even for higher memory allocations (*e.g.,* 300 kB), although the outperformance is reduced.

Tuning of Modulo. Choosing a higher value of th_1 that filters more flows and prevents them from reaching the last stages of the pipeline improves accuracy (Fig. 4(c)). Also, most counters should be placed in the first stages, and only a few of them in the last stages (Fig. 4(d)). This way, due to larger widths in early stages, the probability of collisions in the first stages is also lowered and the number of false positives reduced. However, this comes at a cost: the probability of a collision between two flows reaching the last stages is



(a) Memory 15kB.

(b) Reducing memory to 5kB.

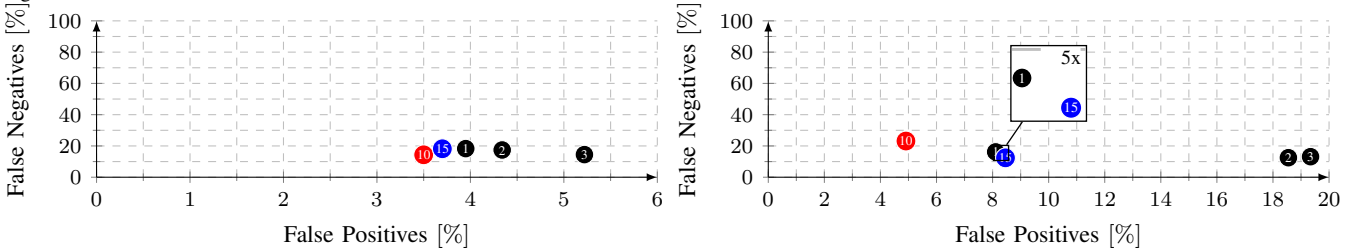
(c) Impact of thresholds. Memory 15kB.

(d) Impact of widths. Memory 15kB.

(e) Impact of interval size. Memory 5kB.

(f) CCDF of count estimation error. Memory 15kB.

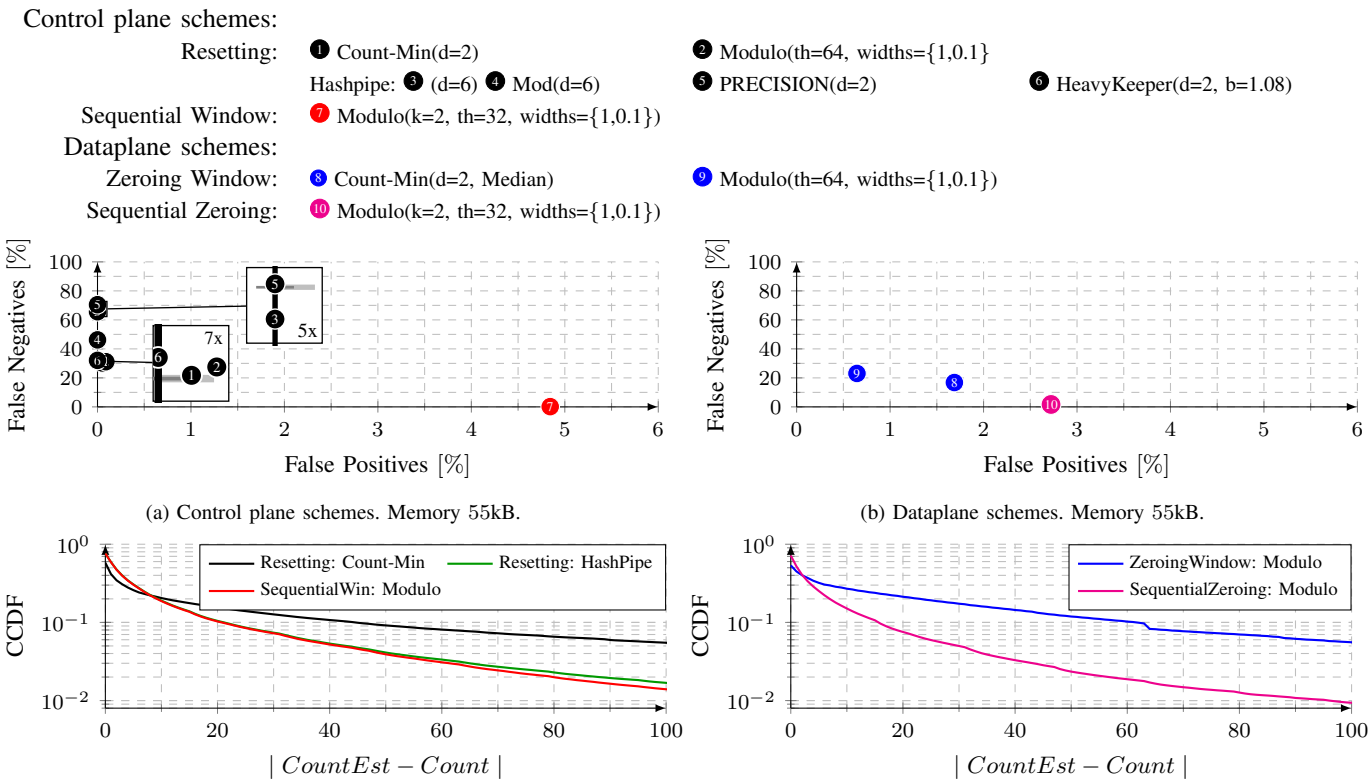
Fig. 4. *Interval-based* simulation of all schemes using 40 different CAIDA traces. **(a)** False-positive vs. false-negative rates of all schemes given 15kB of memory and an interval of $N = 2^{16}$ packets. HashPipe and our modified HashPipeMod with flowID fingerprints exhibit many false negatives, which we try to avoid (note that they would also run at half the line-rate). Count-Min (CM) performs better, especially with $d = 2$ stages. Our Modulo scheme performs best, especially with $d = 2$. **(b)** The smaller memory of 5kB displays comparable results, although CM begins to underperform. (Note that even with larger memories, CM never performs better than our schemes.) **(c)** Varying the thresholds shows that it is better to set high thresholds at the first stages to filter out the small flows. **(d)** Varying the widths of each stage shows a relatively small sensitivity; though it seems better to decrease the width in the second stage by an order of magnitude in order to exploit the smaller number of collisions. **(e)** Increasing the interval size N further increases the gap in the false-positive rate between Modulo and the other schemes, since the exponential counting of Modulo starts to play a larger role (we used the best parameters from (a) for each algorithm). **(f)** Plotting the CCDF of the count estimation error, *i.e.,* the probability that the absolute difference between the estimated flow size and the real flow size of each packet exceeds some value, shows that Modulo may not fit alternative goals. For instance, it has a larger chance than CM of yielding a high estimation error.



(a) Hardware evaluation. Memory 15kB.

(b) Hardware evaluation. Memory 5kB.

Fig. 5. Netronome SmartNIC hardware experiments for *interval-based* schemes using 40 different CAIDA traces. **(a)** and **(b)** correspond to the simulation settings of Fig. 4(a) and 4(b), respectively. On Netronome SmartNICs, the speed of the control plane is the main limiting factor. Intuitively, resetting the interval counts is not immediate. While an RPC call is initiated every second for every array, these actions are not executed instantaneously, resulting in an increased number of false negatives. Hashpipe and PRECISION cannot provide an online count estimate (see Appendix). Similarly, as programmable hardware does not support floating point operations, nor loops to implement fixed point math, HeavyKeeper cannot be implemented.



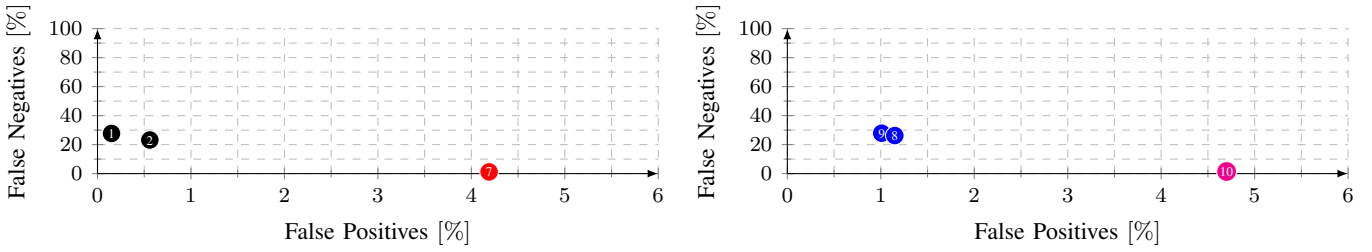
(a) Control plane schemes. Memory 55kB.

(b) Dataplane schemes. Memory 55kB.

(c) Control plane schemes. Count estimation Error. Memory 55kB.

(d) Dataplane schemes. Count estimation Error. Memory 55kB.

Fig. 6. *Sliding window* simulation using 40 different CAIDA traces. **(a)** False-positive vs. false-negative rates of schemes *that need the control plane for deletions*, given 55kB of memory and a window of $N = 2^{16}$ packets. Simply periodically resetting interval-based schemes yields significant false negative rates, because at the start of each interval these schemes do not take into account packets that appeared previously. The *Sequential Window* approach that implements several sub-intervals of interval-based schemes yields significantly lower false negative rates, yet the false positive rates are slightly high. **(b)** Same as (a) with schemes that can perform resets in the data plane and do not need the control plane. *Zeroing Window*, which periodically resets each counter, attempts to reach some compromise between false positives and negatives, but still yields a non-negligible rate of false negatives. *Sequential Zeroing*, which applies the Zeroing Window approach to the last sub-interval of Sequential Window, achieves a good performance with low false negative rates and reasonable false positive rates. **(c)** and **(d)** Looking at the CCDF of the count estimation error, we find that the Sequential Zeroing schemes perform best again, even though they do not need control-plane intervention.



(a) Hardware evaluation. Control plane schemes. Memory 55kB.

(b) Hardware evaluation. Dataplane schemes. Memory 55kB.

Fig. 7. Netronome SmartNIC experiments for *sliding window* schemes using 40 different CAIDA traces. HashPipe and PRECISION cannot provide an online count estimate (see Appendix). And, since programmable hardware does not support floating point operations, nor loops to implement fixed point math, HeavyKeeper cannot be implemented. **(a)** and **(b)** correspond to the simulation settings of Fig. 6(a) and Fig. 6(c), respectively. **(a)** On programmable hardware, the Sequential window was able to maintain high accuracy. **(b)** Our Sequential Zeroing approach displays a negligible rate of false negatives. Its slight increase in false positives when compared to simulations may be due to race conditions in hardware.

increased, leading to an increased probability of high count over-estimation (Fig. 4(f)).

Interval sizes. Fig. 4(e) shows that Modulo performs very well with longer interval sizes, suffering only from a small decrease in accuracy (0.12 percentage points of false positives) given the same memory consumption ($5kB$).

Count estimation error. Fig. 4(f) shows the complementary cumulative distribution function (CCDF) of the absolute count estimation error, *i.e.,* the probability of exceeding a given value. HashPipe is clearly outperformed by all other approaches, which is also reflected by its large number of false negatives (see Fig. 4(b) and 4(a)). Count-Min has a higher probability than Modulo of being mistaken in the flow count estimation, but a slightly smaller probability of being significantly mistaken (by more than 17).

Hardware experiments. Fig. 5(a) and 5(b) run experiments

on Netronome SmartNIC, using the same settings as the simulations of Fig. 4(a) and 4(b), respectively. We assume that the control plane takes time resetting intervals, and runs a full reset of all counters each time in a somewhat naive way. As expected, we find that this indeed impacts the performance of all schemes.

C. Sliding-window measurements

Control-plane solutions. Fig. 6(a) and Fig. 6(c) show that our Sequential Window outperforms all resetting solutions that rely on an interval-based scheme and reset it periodically. Sequential Window keeps a low percentage of false positives (as low as 4.19% with just $55kB$),

Data-plane solutions. Fig. 6(b) shows that our Zeroing algorithm in combination with Modulo always outperforms a solution that resets all counts using the control plane, by almost halving the percentage of false negatives. Moreover, Sequential Zeroing, which combines our two window approaches, Sequential window and Zeroing Window, outperforms all other schemes, by dramatically lowering the percentage of false negatives (from over 20% to 1.67%) while keeping a reasonable false positive percentage. Packets from the extra sub-interval are gradually removed, and the count estimation error, as a result, is reduced (Fig. 6(d)).

Hardware experiments. Fig. 7(a) and 7(b) show the results of our experiments on a Netronome SmartNIC, using the same settings as the simulations of Fig. 6(a) and 6(b), respectively. Again, our new schemes significantly outperform the others.

V. Related work

Algorithms relating to heavy-hitter detection can be divided into three groups: (1) *sampling* algorithms, (2) *sketch-based* algorithms, and (3) *counting* algorithms.

Sampling algorithms. Sampling algorithms (NetFlow [12], Sflow [20], Sample&Hold [16]) are currently widely deployed and used by network operators. In these algorithms, nodes usually maintain current flow statistics that are periodically sent to a remote point for further analysis. However, they do not determine for *each* packet whether it belongs to a heavy hitter, which is needed for fine-grained control.

Sketch-based algorithms. Sketch-based algorithms such as ours use specialized data structures called sketches that hash and count all packets in the switch hardware. In exchange for some count overestimation or underestimation, this approach can achieve a considerably lower memory usage, which makes it especially suitable for programmable hardware. Unfortunately, existing algorithms (Count-Min Sketch [13], UnivMon [17], Count Sketch [11], Probabilistic lossy counting [14], CountMax [29], Elastic sketch [27], Cold Filter [30], HeavyKeeper [28]) were not designed for P4-programmable switches and often cannot be directly implemented without modifications or loss of accuracy. For example, to estimate a count of an item, Cold Filter calculates a minimum of d hashed counters in each stage, violating the constraint of one memory access per register array present on modern programmable hardware. Moreover, other meta-algorithms, such as Elastic sketch that relies on the Count-Min sketch, are orthogonal to our approach and could benefit from using our sketches with higher accuracy.

Counting algorithms. Counting algorithms (HashPipe [22], PRECISION [5], Space-Saving Algorithm [18], CSS [6]) maintain a data structure consisting only of heavy-hitter flows and corresponding counts. The Space-Saving algorithm requires either maintaining a sorted list or finding an item with the minimum counter value. Unfortunately, both are either not supported by existing programmable hardware or exceed the available processing budget. CSS uses TinyTable [15], which also violates the available processing budget. Hashpipe [22] is explained in the Appendix and PRECISION [5] is similar. Both were designed for P4, but cannot operate at line-rate.

Sliding window approaches. (WCSS [6], SWAMP [2], [3], Memento [4]) remove the oldest entries from the counting data structure so that only information about the last N processed packets is present at the switch. SWAMP [2], [3] maintains an additional array with flow identifiers from the last N packets. Every time a new packet arrives the oldest entry from the array is removed and replaced with a new flow identifier. However, depending on the selected window size, memory consumption is very high. Ben-Basat et al. present two different solutions in [4], [6] optimized for memory consumption with constant query time. However, their use of TinyTable [15] is unsuitable for programmable network hardware.

VI. Conclusion

In this paper, we introduced the first heavy-hitter detection algorithm for programmable switches that provides per-packet granularity at line-rate performance.

To do so, we first introduced the *conditional sketching* technique that filters most small flows in early stages, and illustrated it by developing an interval-based sketching algorithm called *Modulo* sketch. Next, we addressed the problem of enabling such conditional sketching to work over *sliding windows*. Specifically, we started with the *Sequential Window* algorithm that is based on sub-intervals and needs the control plane. We then presented the *Zeroing Window* technique that periodically resets each counter in any interval-based sketch and which works fully in the data plane. Last, we combined both techniques to obtain the in-data-plane *Sequential Zeroing* scheme. In our evaluations, we showed how our techniques significantly improve the accuracy of our estimation when compared to several baseline algorithms inspired by the literature, and implemented our schemes on a Netronome SmartNIC.

Beyond bringing sliding-window heavy hitter detection to the dataplane, we believe the techniques introduced in this paper, such as (1) zeroing through ping-ponging the memories, (2) sequential windows, and (3) counter reuse through modulo, can also benefit dataplane measurement applications in general.

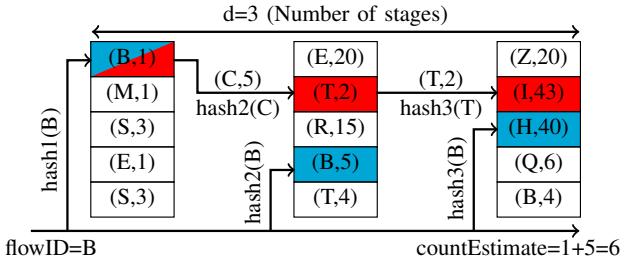


Fig. 8. HashPipe algorithm. Based on [22].

APPENDIX A
HASHPIPE IMPLEMENTATION

HashPipe. HashPipe [22] consists of d consecutive stages, each with its own counter array and its own hash function. In addition to flow counts, HashPipe also stores the corresponding flowIDs (see Fig. 8).

Upon a packet's arrival, its flowID is hashed to produce an index and compared to the flow identifier $flowID_1$ currently stored at that index in the first stage. If the identifiers do not match, the $flowID_1$ and count are *evicted* from the first stage and replaced by the new flowID and count 1. If the identifiers do match, the count is simply increased by 1. When a flow identifier $flowID_i$ (and count) is evicted from stage i, HashPipe will try to store it in the next stage $i+1$ by following the same process until the last stage is reached. In addition, to compute the count estimate, counts of all matching pairs from each stage $(flowID, *)$ need to be summed. Fig. 8 illustrates this: flow insertion uses the red counts, while the count estimation uses the blue counts. As each array can only be accessed once, each packet would need to go through the pipeline twice to get an estimate for each packet, halving the throughput. Since PRECISION is similar to HashPipe, it suffers from the same throughput reduction.

In this paper, we have implemented two versions of HashPipe: (1) the original algorithm storing the full flow identifier (13B for a 5-tuple); and (2) *HashPipeMod*, a modification of the original algorithm that we introduce for a fairer comparison. HashPipeMod stores a fingerprint of the flow identifier (2B) instead of the full flow identifier. Thus, HashPipeMod has lower memory consumption than HashPipe, at the cost of introducing false positives (since a non-heavy-hitter flow may obtain the same fingerprint as a heavy-hitter flow).

ACKNOWLEDGEMENTS

We thank Koen Langendoen for his helpful insights and comments. This work was partly supported by SURFnet, the Israel Science Foundation (grant No. 1119/19) and the Technion Hiroshi Fujiwara Cyber Security Research Center.

REFERENCES

[1] About Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/. [Online; accessed 16-January-2020].

[2] ASSAF, E., BEN-BASAT, R., EINZIGER, G., AND FRIEDMAN, R. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. *CoRR abs/1712.01779* (2017).

[3] BASAT, R. B., EINZIGER, G., FRIEDMAN, R., AND KASSNER, Y. Poster abstract: A sliding counting bloom filter. In *INFOCOM WKSHPS* (May 2017), pp. 1012–1013.

[4] BASAT, R. B., EINZIGER, G., KESLASSY, I., ORDA, A., VARGAFTIK, S., AND WAISBARD, E. Memento: Making sliding windows efficient for heavy hitters. *arXiv preprint arXiv:1810.02899* (2018).

[5] BEN-BASAT, R., CHEN, X., EINZIGER, G., AND ROTTENSTREICH, O. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (2018), IEEE, pp. 313–323.

[6] BEN-BASAT, R., EINZIGER, G., FRIEDMAN, R., AND KASSNER, Y. Heavy hitters in streams and sliding windows. In *INFOCOM* (2016), pp. 1–9.

[7] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *ACM IMC '10* (2010), pp. 267–280.

[8] BIFULCO, R., AND RÉTVÁRI, G. A survey on the programmable data plane: Abstractions architectures and open problems. In *Proc. IEEE HPSR* (2018), pp. 1–7.

[9] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (2014), 87–95.

[10] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM CCR 43*, 4 (2013), 99–110.

[11] CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming* (2002), Springer, pp. 693–703.

[12] CLAISE, B. Cisco systems NetFlow services export version 9. Tech. Rep. 2070-1721, Internet Engineering Task Force, 2004.

[13] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms 55*, 1 (Apr. 2005), 58–75.

[14] DIMITROPOULOS, X., HURLEY, P., AND KIND, A. Probabilistic lossy counting: an efficient algorithm for finding heavy hitters. *SIGCOMM Comput. Commun. Rev. 38*, 1 (2008), 5–5.

[15] EINZIGER, G., AND FRIEDMAN, R. Counting with tinytable: Every bit counts! In *ICDCN* (2016).

[16] ESTAN, C., AND VARGHESE, G. New directions in traffic measurement and accounting. *SIGCOMM Comput. Commun. Rev. 32*, 4 (Aug. 2002), 323–336.

[17] LIU, Z., MANOUSIS, A., VORSANGER, G., SEKAR, V., AND BRAVERMAN, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 101–114.

[18] METWALLY, A., AGRAWAL, D., AND EL ABBADI, A. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory* (2005), Springer, pp. 398–412.

[19] MUTHUKRISHNAN, S., ET AL. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science 1*, 2 (2005), 117–236.

[20] PHAAL, P., PANCHEN, S., AND MCKEE, N. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. Tech. Rep. 2070-1721, Internet Engineering Task Force, 2001.

[21] RONG, Q., ZHANG, G., XIE, G., AND SALAMATIAN, K. Mnemonic lossy counting: An efficient and accurate heavy-hitters identification algorithm. In *IEEE IPCCC* (2010), pp. 255–262.

[22] SIVARAMAN, V., NARAYANA, S., ROTTENSTREICH, O., MUTHUKRISHNAN, S., AND REXFORD, J. Heavy-hitter detection entirely in the data plane. *ACM SOSR* (2017), 164–176.

[23] The CAIDA UCSD anonymized internet traces - 2016, 2018. Available at http://www.caida.org/data/passive/passive_dataset.xml.

[24] Tofino: World's fastest P4-programmable Ethernet switch ASICs. https://barefootnetworks.com/products/brief-tofino/. [Online; accessed 16-January-2020].

[25] TURKOVIC, B., KUIPERS, F., VAN ADRICHEM, N., AND LANGENDOEN, K. Fast network congestion detection and avoidance using p4. In *CoNEXT* (New York, NY, USA, 2018), NEAT '18, ACM, pp. 45–51.

[26] WU, X., AND LUO, Y. Network Measurement with P4 and C on Netronome Agilio. Available at https://www.slideshare.net/Open-NFP/network-measurement-with-p4-and-c-on-netronome-agilio.

[27] YANG, T., JIANG, J., LIU, P., HUANG, Q., GONG, J., ZHOU, Y., MIAO, R., LI, X., AND UHLIG, S. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM* (2018), pp. 561–575.

[28] YANG, T., ZHANG, H., LI, J., GONG, J., UHLIG, S., CHEN, S., AND LI, X. Heavykeeper: An accurate algorithm for finding top-k elephant flows. *IEEE/ACM Transactions on Networking 27*, 5 (2019), 1845–1858.

[29] YU, X., XU, H., YAO, D., WANG, H., AND HUANG, L. Countmax: A lightweight and cooperative sketch measurement for software-defined networks. *IEEE/ACM Transactions on Networking* (2018).

[30] ZHOU, Y., YANG, T., JIANG, J., CUI, B., YU, M., LI, X., AND UHLIG, S. Cold filter: A meta-framework for faster and more accurate stream processing. In *ACM SIGMOD* (2018), pp. 741–756.