Practical Heavy-Hitter Detection Algorithms for Programmable Switches

Rani Abboud
Technion
Haifa, Israel
srani@cs.technion.ac.il

Roy Friedman
Technion
Haifa, Israel
roy@technion.ac.il

***Abstract*—Programmable switches enable offloading various network functions, such as anomaly detection and traffic engineering, to the same switches that perform packet routing. A fundamental component of many such applications is detecting heavy hitters (largest flows).**

Realizing such data plane algorithms requires taking into consideration all types of limited hardware resources of the switch, including the recirculation bandwidth, number of stages, and memory. This motivates solutions that avoid recirculation, fit into a minimal number of stages, and are memory-frugal.

We introduce a novel sketch for heavy hitter detection, CMSIS, that supports both online detection and offline retrieval of heavy hitters and achieves high accuracy while incurring low resource consumption. We implemented CMSIS in P4 for the Tofino 2 target. This implementation of CMSIS requires no recirculation and consumes 25% less pipeline stages than state-of-the-art alternatives that do not perform recirculation, while its memory consumption is competitive with prior works.

***Index Terms*—heavy-hitter detection, programmable switches, in-network computation**

I. Introduction

The functionality of programmable switches [1], [2] can be programmed, e.g., using the P4 domain specific language [3], while maintaining high throughput. The main objective of a network switch is to receive and forward packets in the network, and thus, it must operate in line rate even when it is programmable. Therefore, the programming model of programmable switches is restricted, such as in the Reconfigurable Match Tables (RMT) [4] model.

Specifically, RMT is a pipelined architecture for switching hardware, which allows the data plane to be changed without making any modifications to the chip hardware. To maintain the high packet processing rate, implementations of the RMT model impose several restrictions on the user-defined switch program, including limitations on branching, memory access, and the number of pipeline stages. These restrictions can make implementing network applications quite challenging.

Sketches are probabilistic approximation algorithms that only store a small synopsis of the data [5]. Sketches can be utilized in programmable switches to implement useful network functionalities and execute them inside the data plane [6]. These include, e.g., load balancing [7]–[9], anomaly detection [10]–[14], and traffic engineering [15]–[17]. A primary task used by many such applications is to identify the *heavy hitters*, which are the most popular flows. Formally, a heavy hitter flow is a flow that contributes at least a θ fraction (e.g., 0.1%, 0.05%) of the total network traffic, i.e., appeared at least $\theta \cdot N$ times, where N is the total number of packets (events) encountered so far. Heavy hitter detection sketching algorithms attempt to keep track of flows with the largest number of appearances while being resource frugal.

Most works in this area focus on minimizing SRAM usage [18]–[23]. Some previous work has even considered memory usage as low as 5KB [20], which is significantly lower than a typical memory allocation unit on Tofino [1]. Most state-of-the-art algorithms that are implementable on programmable switches employ packet recirculation [18], [19]. That is, sending the packet again through the switch, in this case, to overcome the hardware restriction that does not allow a pipeline stage to access the memory of an earlier stage. Recirculation may reduce the algorithm's throughput and restrict the ability to incorporate several different applications on the same switch hardware, as we report below.

In reality, the compiler may package multiple functionalities onto the same switch as long as their combined resource requirements can be satisfied. This motivates a more holistic view regarding hardware resource utilization, including the number of pipeline stages and recirculation bandwidth. Further, it is important to note that there is a non-negligible minimal memory allocation unit per pipeline stage, which means that focusing only on the theoretical overall memory requirements of a given solution is misleading.

Another aspect of heavy hitter detection algorithms one should consider is *online detection* vs. *offline retrieval*. In the former, packets are labeled in real-time as they pass through the switch, whether they belong to a heavy hitter flow or not, while the latter enables the control plane to occasionally examine the data plane's internals and identify the list of heavy hitter flows. As we elaborate later on, each of these is useful for certain applications and situations.

***Our Contributions**:* We present CMSIS (Count-Min Sketch with Identifier Sampling), our novel sketching algorithm for heavy hitter detection and frequency estimation. CMSIS is tailored for the RMT architecture, on which most popular programmable switches are based. CMSIS completely

ISBN 978-3-903176-63-8 © 2024 IFIP

avoids recirculation, supports both online detection and offline retrieval, and requires fewer pipeline stages than prior competing ideas. CMSIS is the first heavy hitter detection algorithm for programmable switches that allows tuning its sensitivity while the P4 program is running, which can be useful in DoS mitigation, as discussed in Section IV-C. We also describe an online detection only light-weight version called CMS+Threshold that consumes even fewer pipeline stages.

We also propose a simple mechanism for updating the heavy hitters threshold in real-time, according to the total number of packets. This allows us to strictly follow the definition of heavy hitter flows ($\theta \cdot N$) throughout the algorithms' lifetime, unlike previous work [19], [22], [23] which compare flow count estimates to a predefined threshold. The real-time threshold calculation is described in Section IV-E.

We have implemented CMSIS and CMS+Threshold in P4 on Tofino 2 [2], where CMSIS only consumes 6 pipeline stages, and CMS+Threshold only requires 3 stages. Table I compares CMSIS and CMS+Threshold with other heavy hitter detection algorithms. CMSIS and CMS+Threshold consume 25% fewer pipeline stages than their state-of-the-art alternatives that do not employ recirculation (FCM+TopK and FCM-Sketch, respectively [22]). We make our P4 implementation of CMSIS and CMS+Threshold publicly available [24].

Sketch	Tofino impl.	Avoids recirculation	Online detection	Offline retrieval	# stages
HashPipe [21]	X	✓	X	✓	-
PRECISION [18]	✓	X	✓	✓	11
dSketch [19]	✓	X	✓	X	4
FCM-Sketch [22]	✓	✓	✓	X	4
FCM+TopK [22]	✓	✓	✓	✓*	8
CMS+Th	✓	✓	✓	X	3
CMSIS	✓	✓	✓	✓	6

TABLE I: Comparison of several heavy-hitter detection algorithms targeting programmable switches.

II. Related Work

State of the art heavy hitter detection algorithms such as RAP [25] and Space-Saving (SS) [26] are not implementable on programmable switches as these assume non-restricted access to memory, which violates the programmable switching hardware's limitations. For RAP and SS, the unrestricted memory access is needed for finding the minimal value between all counter values. dW-RAP is a d-way variant of the RAP algorithm that is more hardware-friendly as it accesses only d elements and calculates the minimal value among them. One might be tempted to try implementing dW-RAP on programmable switches using d pipeline stages. However, one of the RMT model restrictions is that code executed inside one stage cannot access memory allocated to another stage. Therefore, we can only calculate the minimal value out of the d values after passing through the dth stage, where we no longer have access to previous stages. Thus, we cannot update the minimal value in case it is located in a previous stage.

*Limited support. Can only handle 32-bit and 64-bit identifiers on Tofino 1 and Tofino 2, respectively. See discussion of FCM+TopK in Section II.

PRECISION [18] applies probabilistic *recirculation* as a workaround and achieves very good accuracy. Recirculation is the act of recirculating the packet again through the pipeline after it has completely passed through it (at least) once. A drawback of recirculation is that it negatively impacts the throughput of the switch as the recirculated packet passes again through the pipeline instead of allowing new packets to go through, and this also increases latency. In addition, PRECISION requires a substantial amount of hardware resources.

dSketch [19] also relies on recirculation but consumes fewer hardware resources than PRECISION. It consumes only 4 pipeline stages, while PRECISION needs at least 11.

HashPipe [21] was the first attempt to implement heavy hitter detection for programmable switches. It avoids recirculation, but is not implementable on today's programmable switches either [18]. HashPipe, as reported in [21], was implemented for BMv2 [27], a P4 software switch that is less restrictive than real programmable switches. Univ-Mon [28], a framework for flow monitoring, was also implemented on BMv2 and is not trivial to implement on real programmable switches. Similarly, ElasticSketch [23] cannot be implemented on actual programmable switches without modifications that significantly lower its accuracy.

Sequential Zeroing [20] presents an online heavy hitter detection algorithm for SmartNICs [29], which have a less restrictive model than programmable switches. However, their algorithm is not implementable on Tofino as it requires performing operations that are more complex than those supported today.

FCM-Sketch [22] is a sketch for network measurement tasks, including heavy hitter detection and flow size estimation, in programmable switches. FCM-Sketch was also presented as a possible alternative for CMS (Count-Min Sketch [30]) as a component in different in-network measurement algorithms. The work in [22] also presents another sketch, FCM+TopK, that combines FCM with a modified version of ElasticSketch [23], which keeps track of heavy flows' identifiers. However, their algorithm cannot support identifiers wider than 32-bits (on Tofino 1). Note that an IPv4 5-tuple or a single IPv6 address cannot be stored by their algorithm, even on Tofino 2. FCM+TopK must store the flow identifiers together with a counter inside the same register to perform atomic operations on both values simultaneously. This causes the flow identifier to be limited to half the largest register width supported by the hardware (32 bits in Tofino 1 and 64 bits in Tofino 2). While storing a 32-bit (or 64-bit) hash of the flow identifier is possible, offline retrieval of heavy flows' identifiers this way requires additional communication.

Previous work that targets the $\theta \cdot N$ heavy hitter definition [19], [20], [22], [23] compares the count estimations to a predefined static threshold (e.g., 10K in [22]). However, this does not follow the online heavy hitter detection definition, especially when not working with sliding windows (e.g., FCM [22], ElasticSketch [23], Modulo Sketch [20], and adaptions of CMS [19]). In contrast, we calculate the threshold ($\theta \cdot N$) in real-time, so the threshold grows together with the

growing stream size N. We elaborate on this in Section IV-E.

III. Motivation

A. Resource Usage of Data Plane Applications

When designing data plane applications for programmable switches, we would like these applications to have the lowest resource consumption possible to allow additional applications to run on the same hardware. Previous work has focused mainly on minimizing the amount of SRAM dedicated to these algorithms [18]–[21] and reducing the number of pipeline stages [19], [20]. In reality, P4 compilers usually place functions of multiple data plane applications on the same pipeline stages, and these are not dedicated each to a single application. The compiler is limited by how many functionalities it can stack into a single stage depending on the amount of hardware resources (e.g., memory, registers accessed, hash calculations) used by the data plane algorithms. Therefore, the number of pipeline stages by itself does not indicate how heavy the data plane application is. Other hardware resources should also be considered, including the recirculation bandwidth.

Recirculation Considered Wasteful: As mentioned earlier, PRECISION [18] and dSketch [19] employ packet recirculation to overcome the switch's restriction that a pipeline stage cannot access memory areas allocated to other stages. As every pipeline in the switch processes one packet during each clock cycle, recirculated packets affect the switch's throughput. Also, in dSketch [19], the number of recirculations in a given time interval is only bounded by the number of unique flows in that interval. Therefore, a DDoS attack could cause a significant portion of the packets to be recirculated, potentially cutting the throughput in up to half.

In addition, the dSketch paper [19] claims that under full network load of 6.4Tbps (in Tofino 1), recirculating 2% of the packets consumes 50% of the switch's recirculation port's throughput. However, we claim that recirculation bandwidth should be treated as a switch resource like any other. That is, consuming 50% of this bandwidth is reasonable when the heavy hitter detection functionality is the only one requiring recirculation on the switch. However, when multiple functionalities are placed on the same switch, and the recirculation bandwidth needs to be shared among them, consuming half of the recirculation port's bandwidth for a single application is too costly. We can set additional recirculation ports by converting standard ports into recirculation ports, but this will also significantly affect the switch's throughput and utilization.

B. Online Detection vs. Offline Retrieval

Online detection data plane algorithms give an estimation or a label for a packet as it passes through the switch. This allows the data plane to take an action, e.g., drop the packet or forward it to a specific port, based on the online algorithm's decision, before the packet departs from the switch. Examples of such algorithms include dSketch and Sequential Zeroing. Online detection algorithms can save memory since they do not need to store flow identifiers.

In contrast, *offline retrieval* algorithms store enough information about suspected heavy hitter flows and possibly other flows so that the control plane can read these data structures and decide which flows are indeed heavy hitters. For example, HashPipe does not support online detection (explained in [20]), and only allows offline retrieval. To be precise, in order to allow online detection, HashPipe needs to recirculate every packet (whose identifier does not already exist in the first stage), which would cut the switch's throughput in half in the worst case, making it impractical. PRECISION was also presented as an offline retrieval algorithm but unlike HashPipe, it can easily be adapted to provide online estimations, as every packet accesses all entries potentially belonging to its flow while it traverses the sketch's data structure before deciding whether to recirculate the packet. Elastic Sketch [23] also attempts to keep track of the top flows' identifiers.

Offline retrieval by itself does not allow reacting to heavy flows in real-time. In that sense, offline retrieval may seem less useful. However, offline retrieval enables the control plane to inspect for heavy hitter flows, which is useful for many applications, e.g., network security. This is impossible in online-only heavy hitter detection algorithms [19], [20], [22], [30], as they do not store flow identifiers.

Our main algorithm, CMSIS, supports both online detection and offline retrieval. This is possible because we store the identifiers of flows suspected to be heavy hitters in CMSIS's data structure. These can later be read by the control plane.

IV. Architecture

A. Warm-up: Primitive CMSIS

When designing an algorithm for programmable switches that does not use recirculation, we must respect the restriction that a stage i will never be able to access or send any information to an earlier stage $j < i$. We design our sketch such that all information flows in one direction.

The primitive version of our algorithm, presented here as a warm-up exercise, consists of two parts: Part *(i)* stores count estimations, and Part *(ii)* stores flow identifiers, as shown in Figure 1. Data is structured as rows where each flow is mapped to a single row using a hash function. Each row holds a frequency estimation counter that is incremented on every access, and a structure that holds three slots for flow identifiers. A flow's identifier is inserted into the first slot in the row with a relatively small probability (e.g., $\frac{1}{128}$). When this happens, the previous value in the first slot replaces the value in the second slot, and the value of the second slot will replace the value of the third slot (i.e., shifting).

Part *(i)* provides the frequency estimation for a flow while Part *(ii)* functions as a filter that filters out flows that are not heavy hitters but are mapped to the same row. Since these non heavy hitter flows will rarely appear, and we insert a flow identifier with a low probability, identifiers of non heavy hitter flows will most likely appear at most once in their respective row. A flow will be labeled as a heavy hitter if its frequency estimation is large enough (threshold calculations

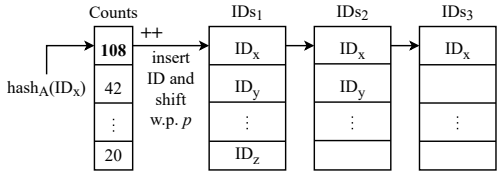


Fig. 1: The primitive version of CMSIS.

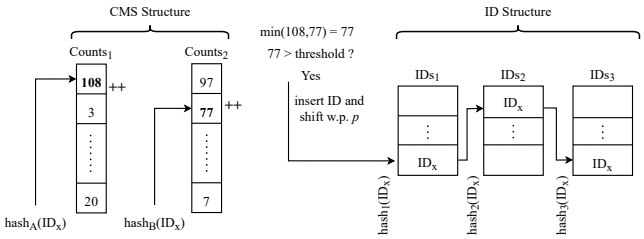


Fig. 2: CMSIS. It includes a 2-way Count-Min sketch used to estimate a flow's frequency and an ID structure that stores identifiers of flows suspected as heavy hitters.

are discussed in Section IV-E) and its identifier appears in the majority of occupied entries in its respective row.

The main shortcoming of this algorithm is that it stores a large number of identifiers of flows that are not heavy hitters. In particular, a row with an extremely low frequency estimation would still occupy three identifier entries. Since most flows are not heavy hitters, most of the data structure's identifiers' entries will hold irrelevant data, turning the algorithm to be extremely inefficient memory-wise.

B. CMSIS: Count-Min Sketch with Identifier Sampling

To avoid the problem described above, our actual algorithm is based on a 2-way CMS [30] and maintains a data structure that stores identifiers of suspected heavy hitter flows and a threshold calculator, as illustrated in Figure 2.

CMSIS uses five different pairwise independent hash functions when processing packets' identifiers. The functions $hash_A$ and $hash_B$ are used to access the 2-way CMS structure, while the functions $hash_1$, $hash_2$, and $hash_3$ are used to access the ID structure in stages 1, 2, and 3 respectively. The ID structure contains significantly fewer entries than the CMS structure, typically 128, 256, or 512 ID entries per stage (depending on the value θ, which affects the expected amount of heavy hitters). Only identifiers of flows suspected as heavy hitters will be inserted into this structure. A different hash function is used at each ID stage to reduce the chance that two heavy hitters are mapped to the exact same entries (i.e., row) thereby preventing each other from occupying a sufficient number of entries in the ID structure to be deemed heavy hitters. Using three ID stages accounts for occasional insertions of non heavy hitter flows while minimizing hardware resource usage. Each additional ID stage requires at least one additional pipeline stage and yields diminishing accuracy returns.

Once a packet passes through CMSIS it receives a frequency estimation through the 2-way CMS, the minimum of $Counts_1[hash_A(ID_x)]$ and $Counts_2[hash_B(ID_x)]$. The heavy hitter threshold is calculated as in Section IV-E below.

Flows whose frequency estimation is larger than the current threshold will enter the ID structure with probability $p = \frac{1}{n}$ such that n is a power of 2 (e.g., $\frac{1}{128}$). The insertion probability p is restricted to this representation to allow performing probabilistic actions in the data plane by generating random numbers that are $(\log_2 n)$-bits long. Arbitrary-range random number generators are not available on Tofino. We have found that values of $p \in \{\frac{1}{64}, \frac{1}{128}, \frac{1}{256}\}$ all work well, and the chosen p has an insignificant effect on accuracy.

The identifier insertion is done while moving the previous value of the entry in the 1st ID stage to the 2nd ID stage and moving the previous entry in the 2nd ID stage to the 3rd ID stage. The previous entry of the 3rd ID stage is dropped. Flows whose identifier is inserted will automatically be labeled as heavy hitters. Flows whose identifier is not inserted, will traverse through the relevant entries in the ID stages and will count the number of times their identifiers were found. A flow with a frequency estimation that is larger than the current threshold and with a number of ID matches that is larger than the required number of matches (typically 1 or 2 out of 3, see Section IV-C) will be labeled as a heavy hitter.

The benefit of this extended design is that it greatly reduces the chance of non heavy hitters entering the ID structure. Consequently, it significantly lowers the chance for non heavy hitters to be labeled as heavy hitters. This is obtained without preventing flow identifiers of heavy hitters from being inserted. Thus, we can obtain good accuracy with much less memory than with the primitive version we discussed in Section IV-A.

Algorithm 1 lists CMSIS's pseudocode. We implemented CMSIS in P4 [3] for the Tofino 2 target, and it fits into 6 pipeline stages. Note that 3 of the 6 stages employed by the CMSIS algorithm are very light, with the last stage consisting mainly of a simple if-statement that does not consume any SRAM resources. This means that other applications can potentially be packaged to use these stages as well.

Although CMSIS was designed for a packet count-based definition of Heavy Hitters, we believe that it can be easily adapted to work with a bandwidth-based definition.

C. Tuning the Sensitivity of CMSIS

CMSIS is capable of indicating its confidence level of the flow's label. This is based on the number of times that a flow's identifier is matched in our data structure, e.g., 1 out of 3 or 2 out of 3 times. This allows the sensitivity of the algorithm to be tuned according to the application's needs. For example, an application that prefers a lower false-negative rate (at the cost of a higher false-positive rate) might want to label a flow (whose count passes the threshold) as a heavy hitter even if it was matched only once. A different application that prefers a lower false-positive rate at the cost of a higher false negative rate can require 2 (or even 3) matches to label a flow whose count passes the threshold as a heavy hitter.

In various applications such as anomaly detection and load-balancing, minimizing false-negatives is more crucial, even at

Algorithm 1 The CMSIS Algorithm

```
1: procedure CMSIS(flow_id)
2:     hh_label ← false
3:     Counts_1[hash_A(flow_id)] ← Counts_1[hash_A(flow_id)] + 1
4:     Counts_2[hash_B(flow_id)] ← Counts_2[hash_B(flow_id)] + 1
5:     estimation ← min(Counts_1[hash_A(flow_id)],
                        Counts_2[hash_B(flow_id)])          ▷ Count-Min
6:     threshold ← Threshold()
7:     if estimation ≥ threshold then
8:         insert ← true w.p. p otherwise false      ▷ Insert flow_id to the ID
   structure with probability p
9:         if insert = true then
10:            new_val ← flow_id
11:            for i in {1,2,3} do
12:                swap(IDs_i[hash_i(new_val)], new_val)
13:            hh_label ← true
14:        else
15:            #matches ← 0    ▷ Count appearances of flow_id in ID structure
16:            for i in {1,2,3} do
17:                if IDs_i[hash_i(flow_id)] = flow_id then
18:                    #matches ← #matches + 1
19:            if #matches ≥ required_matches then
20:                hh_label ← true
21:    output hh_label, estimation
```

the cost of a higher false-positive rate [19]. Yet, note that in the case of anomaly detection, every flow that is incorrectly labeled as a heavy hitter will be passed to a network scrubber. This will cause higher delay times for these falsely labeled as positive flows. As a result, it may be useful to control the sensitivity of the heavy hitter detection algorithm to ensure minimal delays for more legitimate flows at the cost of a slightly higher number of missed heavy hitters.

CMSIS's sensitivity can be tuned from the control plane in real-time by changing the number of required matches. This can be extremely useful as a mechanism for adapting the algorithm to different situations. For example, when the system suspects it is under a DoS attack, the sensitivity of the algorithm can be set to 'high' by setting the required number of matches to 1. A 'high' sensitivity will ensure that the number of missed heavy hitters is minimal. During regular times (no DoS), the sensitivity of the algorithm can be set to 'medium' by setting the required number of matches to 2. 'Medium' sensitivity will ensure a low number of false-positives, which cause legitimate flows to be treated as heavy hitters, thus reducing the number of unfortunate legitimate flows that are delayed by mistake. Changing the sensitivity level does not require re-loading the program to the switch, as it only requires changing a single parameter/register value, which can be changed from the control plane.

In Section V-B we evaluate the four different variants of CMSIS: CMSIS-Mi requires i ID matches to label a flow as a heavy hitter, where $i \in \{0, 1, 2, 3\}$.

D. CMS+Threshold: CMS with online threshold calculation

CMS+Threshold is a lightweight variant of CMSIS that only supports online detection, does not store any flow identifiers, and therefore consumes less SRAM and uses a significantly lower number of pipeline stages. Similarly to CMSIS, it is based on a 2-way CMS to provide the frequency estimation. In addition, it calculates the heavy hitter threshold in real-time. The frequency estimation is compared to the threshold, and

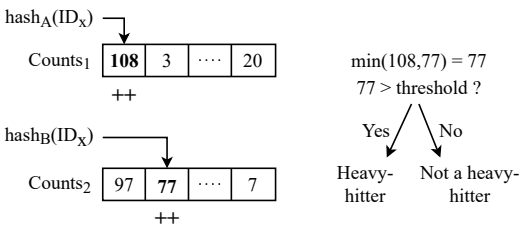


Fig. 3: CMS+Threshold: the lightweight variant of CMSIS. It includes a 2-way Count-Min sketch used to estimate a flow's frequency, which is then compared to the real-time threshold.

then the label is given accordingly, as described in Figure 3. CMS+Threshold fits into 3 pipeline stages, which are lightly used as there is no ID matching nor heavy register usage.

This simple version, other than being very resource-efficient, is superior to other variants in terms of the false-negative rate. The false-negative rate is 0 due to CMS's one-sided error property, meaning that every heavy-hitting flow will, by definition, have a count estimation that is larger than the current threshold and, therefore, be labeled as a heavy hitter. This comes at the cost of a somewhat higher false-positive rate, which can be acceptable in many applications, such as anomaly detection and load balancing [19].

E. Real-time Threshold Calculation

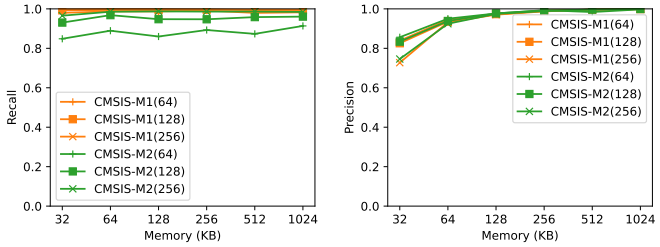
Recall that a heavy hitter flow is a flow that contributes at least a θ fraction of the total network traffic, i.e., appears at least $\theta \cdot N$ times, where N is the total number of packets (events) encountered so far, and θ is a parameter.

As already mentioned, previous work [19], [20], [22], [23] compares the count estimations to a predefined static threshold, e.g., 10K in [22]. Consider the case where a flow f has appeared 9K times when the total number of packets (events) is 100K. When $\theta = 0.1\%$, the flow f is definitely a heavy hitter, by definition, as $9K > 0.1\% \cdot 100K = 100$. However, these algorithms do not label f as a heavy hitter as it has not reached their predefined value of 10K. In addition, when there is a static predefined threshold, more and more flows will reach the threshold as the interval grows, and these will be labeled as heavy hitters even though they have not necessarily contributed a θ fraction of the total traffic.

To truly support online detection, we calculate the threshold ($\theta \cdot N$) in real-time so it grows together with the window N.

Modulo-counting is an efficient and flexible technique for real-time threshold calculation, in which a register consists of two parts: *low* and *high*. Both parts are initially set to zero. The *low* part of the register is incremented by 1 on every packet arrival until it reaches a value of $\frac{1}{\theta}$ when it is set to 0, and then the *high* part is incremented by 1. The heavy hitter threshold is equal to the value of the *high* part of the register.

This technique can be implemented as an atomic register operation, easily fitting into a single pipeline stage. Recall that in both variants of our algorithm, the first pipeline stage consists of the modulo-counting threshold calculation.



(a) Recall. (b) Precision.

Fig. 4: Changing the number of entries in each ID stage in CMSIS – CAIDA '19. $\theta = 0.1\%$.

V. Evaluation

We used the CAIDA 2016, 2018, and 2019 [31]–[33] traces while defining flow identifiers as the packets' IP 5-tuples. The evaluation is performed by feeding the traces, one packet after the other, to the respective implementation. We then inspect the labels and estimations given to each packet for each algorithm and compare them to the ground truth, which is the flow's real count and whether its real count is larger than the current heavy hitter threshold ($\theta \cdot N$). For each algorithm we count the number of true-positives, true-negatives, false-positives, and false-negatives. We then calculate the False positive rate (FPR), False negative rate (FNR), Recall, Precision, F1-score, and Mean squared error (MSE) for each algorithm.

A. Tuning CMSIS Memory Budget Allocation

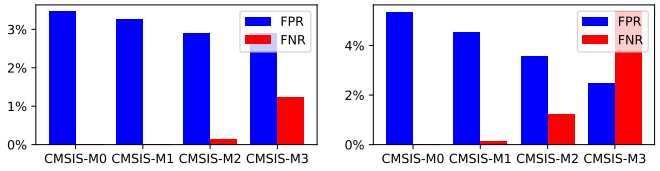
We conducted Python simulations to study the impact of the number of entries in the ID structure of CMSIS when given a total memory budget (the rest is used for CMS counters). We used a subset of CAIDA 2019 [33] that contains 29M packets, and tested the CMSIS-M1 and CMSIS-M2 variants. Recall that CMSIS-Mi requires i ID matches to label a flow, whose estimation is larger than the current threshold, as a heavy hitter.

Figure 4a and Figure 4b indicate that for $\theta = 0.1\%$, 128 entries in each ID stage produces a good balance between Recall and Precision. A higher number of entries would produce a higher Recall for CMSIS-M2 in higher memory ranges at the cost of lower Precision in lower memory ranges. The reason is that a higher number of entries in the ID structure comes at the cost of lower memory for the CMS structure.

Similarly, for $\theta = 0.05\%$, the same balance between Recall and Precision is achieved by allocating 256 entries in each ID stage of CMSIS. The higher number of entries required here is expected as a smaller θ implies more heavy hitters.

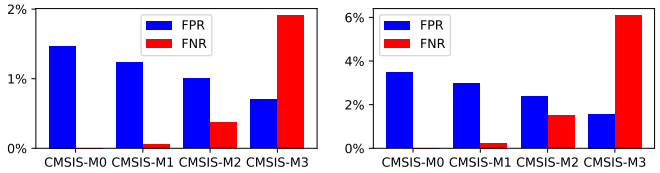
B. Evaluation on the Tofino Model

We implemented CMSIS on the Tofino Model, a component of Intel® P4 Studio [34], that emulates Tofino's hardware. Note that the potential throughput of algorithms executing on Tofino is not a concern, given Tofino's inherent design for processing packets at line-rate, particularly when considering algorithms that do not involve recirculation.



(a) $\theta = 0.1\%$. $128 \cdot 3$ ID entries. (b) $\theta = 0.05\%$. $256 \cdot 3$ ID entries.

Fig. 5: How the required number of matches in CMSIS affects FNR and FPR. CAIDA '16. Evaluated on Tofino Model.



(a) $\theta = 0.1\%$. $128 \cdot 3$ ID entries. (b) $\theta = 0.05\%$. $256 \cdot 3$ ID entries.

Fig. 6: How the required number of matches in CMSIS affects FNR and FPR. CAIDA '18. Evaluated on Tofino Model.

We compile the P4 implementation of CMSIS using Intel Tofino's P4 compiler for the Tofino 2 target [2] and then run the compiled program on the Tofino Model. The hardware resource consumption of CMSIS on Tofino 2 is outlined in Table II, along with a comparison to FCM+TopK and PRECISION. In Table III, the hardware resource consumption of CMS+Threshold on Tofino 2 is presented alongside FCM, an alternative online detection algorithm. The percentages presented in Tables II and III are per pipeline.

We define CMSIS to contain 16K counter entries in each CMS way and either 128 or 256 ID entries per ID stage, depending on the value of θ, according to Section V-A. The insertion probability was set to $p = \frac{1}{128}$. The actual SRAM consumption on Tofino 2 is 1.5%.

CMSIS-M0 ignores the ID structure of CMSIS, and is equivalent to CMS+Th which compares the frequency estimation of CMS to the current threshold (as in Section IV-D).

We used CAIDA Anonymized Internet Trace 2016, 2018, and 2019 [31]–[33], truncated to their first 20 million packets. We start the accuracy evaluation after the first 1M packets to allow the sketches to build up. In particular, early heavy hitters that are labeled while the threshold value is still very low (in this case, less than 1000 or even 500, for $\theta = 0.1\%$ and $\theta = 0.05\%$, respectively) are not particularly interesting.

The results on the three traces were similar. For brevity, we show only the results for the CAIDA 2016 and 2018 traces.

Figures 5 and 6 indicate that the CMSIS-M1 and CMSIS-M2 provide the best balance between the false negative and false positive rates. Thus, we focus on these two variants of CMSIS in our evaluations in Section V-C.

C. Python Simulations

We compared CMSIS with other heavy hitter detection algorithms like PRECISION and HashPipe using Python sim-

Resource	CMSIS	FCM+TopK	PRECISION
TCAM	0%	0%	1.3%
Hash Bits	1.6%	2.2%	3.3%
Stateful ALUs	11.3%	12.5%	13.8%
VLIW Actions	3.0%	1.6%	4.7%
Match Crossbar	7.5%	2.3%	4.5%
Gateway	6.9%	3.4%	12.5%
Physical Stages	6	8	13

TABLE II: Hardware resource consumption of CMSIS, FCM+TopK, and PRECISION on Tofino 2. FCM+TopK cannot store flow identifiers wider than 64-bits on Tofino 2.

Resource	CMS+Th	FCM
TCAM	0%	0%
Hash Bits	0.7%	1.1%
Stateful ALUs	3.8%	7.5%
VLIW Actions	1.3%	1%
Match Crossbar	1.2%	1.3%
Gateway	1.9%	1.9%
Physical Stages	3	4

TABLE III: Hardware resource consumption of CMS+Threshold and FCM on Tofino 2.

ulations [35]. As before, we used the CAIDA 2016, 2018, and 2019 [31]–[33] traces, but this time truncated to the first 100 million packets. We observed similar results in all three traces, so we provide the results only for CAIDA 2018 for brevity.

We have implemented all these algorithms in Python, mimicking their behavior, and added a real-time threshold calculator for each of them. Each algorithm compares its count estimation with the current threshold value ($\theta \cdot N$). A flow that exists in the data structure of PRECISION or HashPipe, and its count estimation is above the threshold, is labeled as a heavy hitter. For FCM+TopK, FCM-Sketch, and CMS+Threshold, any flow with a count estimation that is above the current threshold is labeled as a heavy hitter. Note that in reality, HashPipe cannot provide an online estimation without making major modifications to the algorithm [20]. In our simulations, we assume that HashPipe performs recirculation to be able to provide online estimations. For PRECISION, we have simulated a recirculation delay of 20 packets, although any amount of delay between 0 and 100 has an insignificant effect on the algorithm's accuracy, according to its authors [18].

Note that FCM+TopK cannot store 16B flow identifiers in reality (as mentioned in Section II), but we still simulate their algorithm as if they could store 16B identifiers. Also note that the entries in the TopK component of their sketch are 24B as we need 16B to store flow identifiers, and the TopK structure also holds two counters where each is 4B wide. We define these counters as 4B to prevent them from overflowing.

Since it is not clear how the memory of FCM+TopK should be divided between its two components (FCM and TopK), we adopt the 128KB and above configurations from [22], where we can allocate 4096 entries to the TopK structure.

As in prior work, the memory consumption of each algorithm was estimated while assuming that flow identifiers are stored in 16B entries (as in the implementation of [18]) and counters are 4B entries. Given M bytes of memory, the algorithms were given the following number of counters:

- CMS+Threshold: $\frac{1}{2} \cdot \frac{M}{4}$ counters in each of the two ways.
- CMSIS: given $c \in \{128, 256\}$ entries in each ID stage, the CMS structure contains $\frac{1}{2} \cdot \frac{M - c \cdot 16 \cdot 3}{4}$ counters in each of the two ways.
- PRECISION: $\frac{1}{2} \cdot \frac{M}{4+16}$ entries in each of the two ways.
- HashPipe: $\frac{1}{2} \cdot \frac{M}{4+16}$ entries in each of the two ways.
- FCM-Sketch[1]: $c = \frac{1}{2} \cdot \frac{M}{1+\frac{2}{8}+\frac{4}{64}} = \frac{8}{21}M$ 1B entries in the first stage, $\frac{c}{8}$ 2B entries in the second stage and $\frac{c}{64}$ 4B entries in the third stage, for each of two trees.
- FCM+TopK[2]: the TopK component gets 4096 entries that are 24B wide. The remaining memory ($M - 4096 \cdot 24$) is allocated for the FCM component.

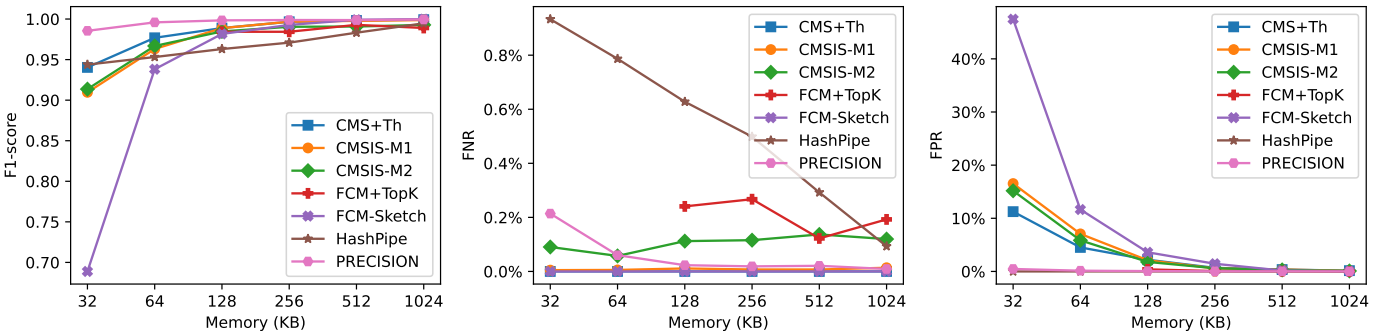
This memory consumption estimation discounts allocation overheads imposed by the hardware, so the actual memory consumption of the algorithms is slightly higher. We also ignore small structures, e.g., for modulo counting, as their sizes are negligible compared to the registers' sizes.

The results of our measurements are shown in Figures 7 and 8. The main insights include:

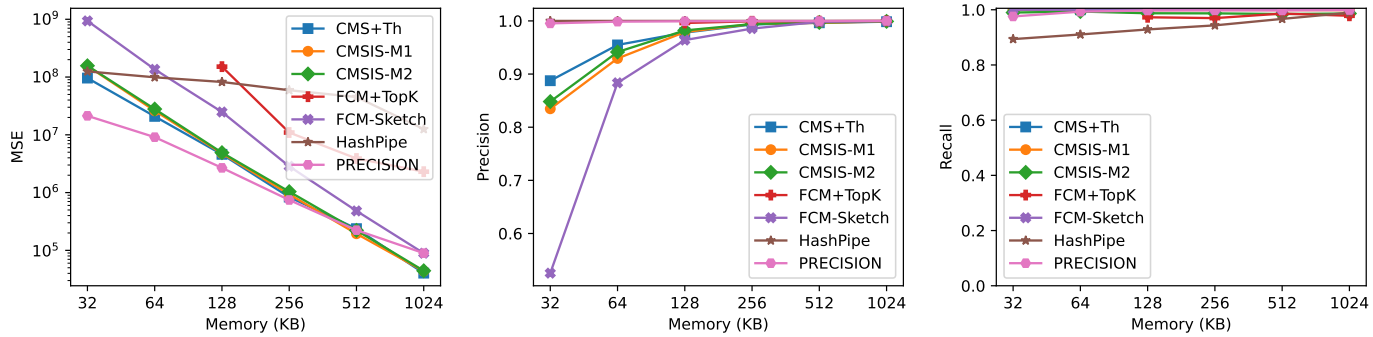
1) CMSIS is competitive with state-of-the-art algorithms in all metrics when given the same amount of memory while being more resource-efficient in terms of the number of pipeline stages and the recirculation bandwidth.
2) CMSIS-M1 performs better than its alternatives which support offline retrieval (PRECISION, FCM+TopK, and HashPipe) in terms of Recall and FNR while consuming almost half the number of stages that PRECISION requires and performing no recirculation. This shows that CMSIS can serve as a more resource-efficient alternative to these algorithms. In addition, CMSIS-M2 outperforms both FCM+TopK and HashPipe in these metrics.
3) CMSIS-M2 reaches a Recall and FNR plateaus where Recall is stable at 98.5% and 96.5% for $\theta = 0.1\%$ and $\theta = 0.05\%$ respectively, even with large amounts of memory (see Figure 8f and Figure 7f). This is because dominant heavy hitters could collide in some stages of the ID structure with other less dominant heavy hitters, preventing them from holding two spots and thus preventing these less dominant flows from being labeled as heavy hitters. We have found that increasing the number of entries in the ID structure allows CMSIS-M2 to overcome this plateau, as shown in Section V-A.
4) CMS+Threshold performs better than both variants of CMSIS in the lower memory ranges in all metrics. This is because CMS+Threshold is an online-only detection algorithm, which does not store any flow identifiers and, therefore, can utilize more of its memory for counter entries, reducing the amount of collisions between different flows. CMSIS, on the other hand, has to allocate a portion of its memory for storing flow identifiers.

[1]The 8-ary variant, which maintains two trees where stages 1-3 hold 1B, 2B, and 4B entries respectively, was recommended in the original paper [22].

[2]In the FCM paper [22], when they present FCM+TopK, they use 4K entries in the TopK structure. They do not explain how the memory should be divided between the FCM-Sketch and the TopK components.

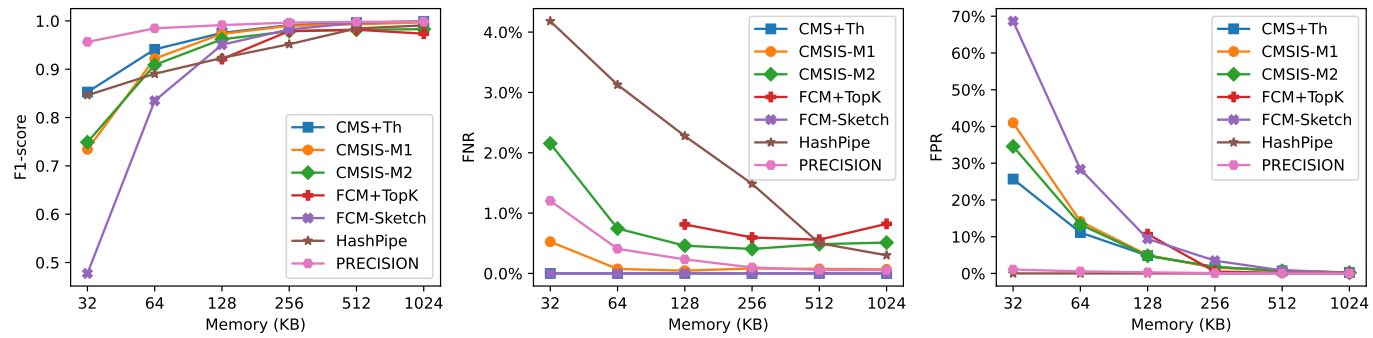


(a) F1-score (b) FNR (c) FPR

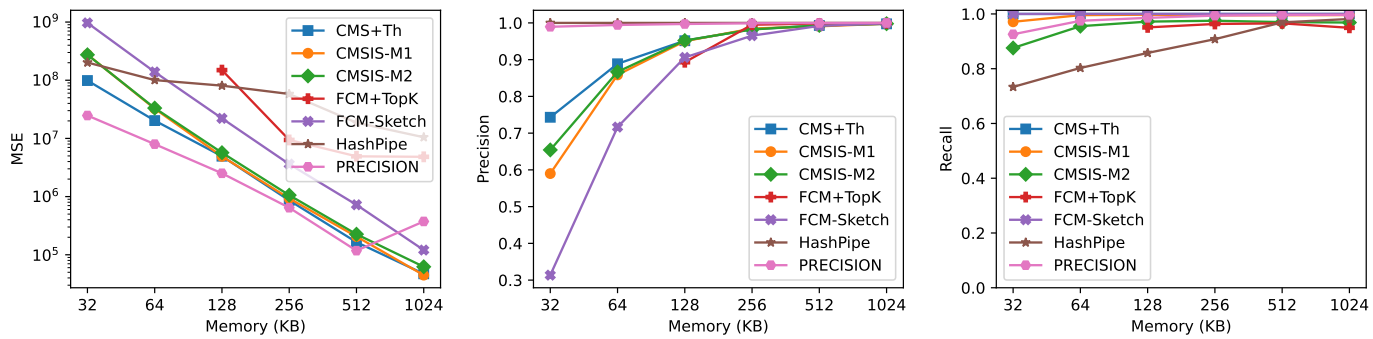


(d) MSE (e) Precision (f) Recall

Fig. 7: CAIDA '18. $\theta = 0.1\%$.



(a) F1-score (b) FNR (c) FPR



(d) MSE (e) Precision (f) Recall

Fig. 8: CAIDA '18. $\theta = 0.05\%$.

If CMS+Threshold and CMSIS were given the same number of counters in their CMS structure, then CMSIS would have a much lower FPR, as shown in Section V-B when comparing CMSIS-M0 with CMSIS-Mi for $i > 0$.

5) CMS+Threshold performs better than its direct online-labeling alternative, FCM-Sketch, in terms of FPR, Precision, and MSE, while Recall and FNR are similar. Further, CMS+Threshold has a simpler structure and consumes fewer pipeline stages.
6) Both CMS+Threshold and FCM-Sketch have perfect Recall and FNR due to their one-sided error.
7) HashPipe has the worst Recall, FNR, and MSE, even though it is not directly implementable on programmable switches available today.

Due to space constraints, offline retrieval evaluation is omitted. Our experiments demonstrated that CMSIS achieves near-optimal Recall comparable to PRECISION and outperforms FCM+TopK and HashPipe, for $\theta = 0.1\%$.

VI. Conclusion

In this paper, we have introduced CMSIS, a novel data plane heavy hitter detection algorithm for programmable switches that supports both online and offline detection. CMSIS avoids recirculation altogether, yet its memory usage is competitive with related work. We implemented CMSIS in P4 for the Tofino 2 target and compared it to state-of-the-art alternatives. This implementation consumes fewer pipeline stages than alternatives that do not perform recirculation. In addition, CMSIS is the first algorithm targeting programmable switches that supports real-time tuning of its sensitivity. Further, it is also the first such algorithm that calculates its heavy hitters' threshold dynamically. We have also shown an adaptation of CMS that performs only online heavy hitter detection while consuming fewer hardware resources than previous work.

References

[1] "Intel® Tofino™," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[2] "Intel®Tofino™ 2," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[4] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," *ACM SIGCOMM CCR*, vol. 43, no. 4, pp. 99–110, 2013.

[5] I. Haber. (2020, March) Count-Min Sketch: The Art and Science of Estimating Stuff. [Online]. Available: https://redis.com/blog/count-min-sketch-the-art-and-science-of-estimating-stuff/

[6] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends," *IEEE Access*, vol. 9, 2021.

[7] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs," in *Proc. of the ACM SIGCOMM*, 2017, pp. 15–28.

[8] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav *et al.*, "CONGA: Distributed Congestion-Aware Load Balancing for Datacenters," in *Proc. of the ACM SIGCOMM*, 2014, pp. 503–514.

[9] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable Load Balancing Using Programmable Data Planes," in *Proceedings of the Symposium on SDN Research*, 2016, pp. 1–12.

[10] X. Z. Khooi, L. Csikor, D. M. Divakaran, and M. S. Kang, "DIDA: Distributed in-network defense architecture against amplified reflection DDoS attacks," in *IEEE NetSoft*, 2020, pp. 277–281.

[11] A. Laraba, J. François, I. Chrisment, S. R. Chowdhury, and R. Boutaba, "Defeating protocol abuse with p4: Application to explicit congestion notification," in *IFIP Networking Conference*, 2020, pp. 431–439.

[12] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi *et al.*, "Flow Event Telemetry on Programmable Data Plane," in *Proc. of the ACM SIGCOMM*, 2020, pp. 76–89.

[13] I. Nevat, D. M. Divakaran, S. G. Nagarajan, P. Zhang, L. Su, L. L. Ko, and V. L. Thing, "Anomaly Detection and Attribution in Networks with Temporally Correlated Traffic," *IEEE/ACM ToN*, vol. 26, no. 1, 2017.

[14] P. Bereziński, B. Jasiul, and M. Szpyrka, "An Entropy-Based Network Anomaly Detection Method," *Entropy*, vol. 17, no. 4, 2015.

[15] H. Mostafaei, M. Shojafar, and M. Conti, "TEL: Low-latency failover traffic engineering in data plane," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4697–4710, 2021.

[16] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine Grained Traffic Engineering for Data Centers," in *Proc. of the 7th Conference on Emerging Networking Experiments and Technologies*, 2011, pp. 1–12.

[17] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Pfabric: Minimal Near-Optimal Datacenter Transport," *ACM SIGCOMM CCR*, vol. 43, no. 4, pp. 435–446, 2013.

[18] R. B. Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Designing Heavy-Hitter Detection Algorithms for Programmable Switches," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, 2020.

[19] X. Z. Khooi, L. Csikor, J. Li, M. S. Kang, and D. M. Divakaran, "Revisiting Heavy-Hitter Detection on Commodity Programmable Switches," in *7th IEEE NetSoft*, 2021, pp. 79–87.

[20] B. Turkovic, J. Oostenbrink, F. Kuipers, I. Keslassy, and A. Orda, "Sequential Zeroing: Online Heavy-Hitter Detection on Programmable Hardware," in *IFIP Networking Conference*, 2020, pp. 422–430.

[21] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 164–176.

[22] C. H. Song, P. G. Kannan, B. K. H. Low, and M. C. Chan, "FCM-sketch: Generic Network Measurements with Data Plane Support," in *Proc. of CoNEXT*, 2020, pp. 78–92.

[23] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic Sketch: Adaptive and Fast Network-Wide Measurements," in *Proc of the ACM SIGCOMM*, 2018, pp. 561–575.

[24] "CMSIS," https://github.com/RaniAbboud/CMSIS-tofino2.

[25] R. B. Basat, X. Chen, G. Einziger, R. Friedman, and Y. Kassner, "Randomized Admission Policy for Efficient top-K, Frequency, and Volume Estimation," *IEEE/ACM ToN*, vol. 27, no. 4, 2019.

[26] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient Computation of Frequent and top-K Elements in Data Streams," in *Proc. of International Conference Database Theory (ICDT)*, 2005, pp. 398–412.

[27] "P4 Language Consortium. P4 Switch Behavioral Model," https://github.com/p4lang/behavioral-model.

[28] Q. Xiao, Z. Tang, and S. Chen, "Universal Online Sketch for Tracking Heavy Hitters and Estimating Moments of Data Streams," in *IEEE INFOCOM*, 2020, pp. 974–983.

[29] "Netronome, "Agilio CX SmartNICs"," https://www.netronome.com/products/agilio-cx/.

[30] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: the Count-Min Sketch and its Applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[31] Anonymized Internet Traces 2016. [Online]. Available: {https://catalog.caida.org/dataset/passive_2016_pcap}

[32] Anonymized Internet Traces 2018. [Online]. Available: {https://catalog.caida.org/dataset/passive_2018_pcap}

[33] Anonymized Internet Traces 2019. [Online]. Available: {https://catalog.caida.org/dataset/passive_2019_pcap}

[34] "Intel® P4 Studio," https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-studio.html.

[35] "Python Simulations Source Code," https://github.com/RaniAbboud/Switch-HH-Simulations.