Revisiting Heavy-Hitter Detection on Commodity Programmable Switches

Xin Zhe Khooi[†], Levente Csikor[†], Jialin Li[†], Min Suk Kang[*], Dinil Mon Divakaran[‡]
[†]National University of Singapore, [*]KAIST, [‡]Trustwave

2021 IEEE 7th International Conference on Network Softwarization (NetSoft) | 978-1-6654-0522-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/NetSoft51509.2021.9492531

***Abstract*—Existing in-network heavy-hitter detection algorithms suffer from several shortcomings. On the one hand, most of the algorithms perform monitoring in intervals and reset the data structures in between; consequently, a notable amount of heavy hitters (HH) spanning across the intervals go undetected. On the other hand, the algorithms consume substantial hardware resources, potentially hindering other data plane functionalities to be integrated on the same device.**

In this work, we revisit the state-of-the-art in-network approaches in this regard and identify that they fall short in overcoming the aforementioned issues. In particular, we investigate whether it is possible to design a heavy-hitter detection algorithm that provides high accuracy without consuming substantial resources, thereby making it feasible to integrate with concurrent applications. To this end, we propose `dSketch`, a time-decaying algorithm for in-network heavy-hitter detection. Trace-driven simulations and evaluations on the Intel Tofino-based commodity switches show that `dSketch` significantly improves the detection rate of HHs by 5-10% while being resource- and operation-efficient in contrast to state-of-the-art approaches. Moreover, we show that `dSketch` can be integrated with standard switch functionalities such as `switch.p4` with additional resources spared, offering itself as a compelling solution for switch data plane designers.

***Index Terms*—Heavy-hitter detection, in-network monitoring, time-decay, resource efficient**

I. Introduction

Due to the ever-increasing traffic and application demands, the capability of real-time per-flow monitoring has become an indispensable requirement for network management. Timely detection of heavy hitters (HH) — flows contributing proportionally high amounts to the overall network traffic — is useful for a wide variety of network applications such as traffic engineering [1], [2], caching specific flow table entries [3], load balancing (of latency-critical applications) [4], per-flow inter-packet metrics gathering [5], network anomaly [6], [7] and attack detection [8], [9]. Once HHs are detected, corresponding actions, e.g., rerouting the relevant flows, can be promptly carried out.

Traditionally, heavy-hitter detection relies on coarse-grained metrics *sampled* via protocols such as NetFlow [10] or via intelligent gathering of per-flow statistics [11]–[13] using remote controllers. The goal is to identify deviations from average traffic behavior, e.g., by means of historical data, and to make timely decisions accordingly. To avoid a huge control plane overhead, the sampling rate is usually set to low (e.g., 1 out of 1000 packets [14]); however, this is insufficient to obtain fine-grained metrics [15] in a timely manner.

978-1-6654-0522-5/21/$31.00 ©2021 IEEE

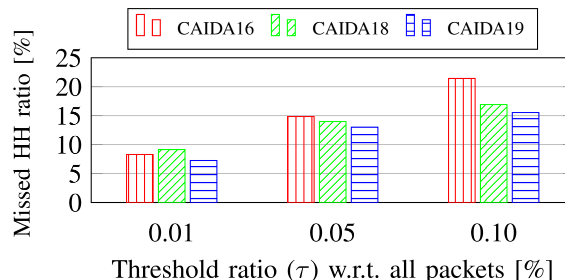


Fig. 1: The ratio of missed HH using interval-reset algorithms with unbounded hash tables (c.f. §IV) on different real-world traffic traces from CAIDA. In every second, the number of HH within the last interval size (of 8 sec) is obtained as ground truth, and compared to the algorithm's report.

The rise of programmable switches, commonly known as PISA (Protocol Independent Switch Architecture) switches, has enabled faster and more accurate methods to monitor *all* packets entirely in the data plane [16], while maintaining terabits per second packet processing performance [17]–[19].

However, designing and implementing data plane algorithms and data structures for PISA switches are *challenging*. These switches have limited hardware resources, restricted set of operations they can perform, and stringent memory access rules, hindering any ideal *but* complicated and resource-hungry application to run. Therefore, several recently proposed *in-network heavy-hitter detection algorithms* rely on probabilistic data structures [20]–[22], [22]–[26] to provide an acceptable trade-off between performance and resource consumption. Unfortunately, this trade-off leads to several major shortcomings.

First, most of these algorithms use an interval-reset approach, i.e., flow monitoring is divided into static time intervals and flow statistics are reset at the end of each interval. A consequence of this approach is that certain heavy flows spanning across intervals can go undetected. An analysis on three traffic traces collected from ISP backbone links shows (cf. Fig. 1) that the ratio of missed HHs when using an interval-reset algorithm is *significant*. Here a flow is considered heavy if it exceeds τ percent of the total number of packets within an interval of 8 seconds. Observe that the ratio of heavy flows going undetected can be up to 10%, 16% and 22% when τ is set at 0.01%, 0.05% and 0.1%[1]. HHs that go undetected can compromise the goal of the control applications. For instance,

[1]These thresholds are commonly used for heavy-hitter detection [22], [23], [26], and, in this case, they mean ∼700 HHs (cf. §IV).

a load balancer would fail to distribute the network load properly, leading to persistent congestion in the network (see more details in §II-B).

Second, existing in-network algorithms are pushing the limits of switch resource consumption in exchange for better results. The number of pipeline stages and memory accesses, and the amount of (per-stage) memory required, however, directly affect the feasibility of integrating an application into today's programmable switches [27]. For instance, a fully functional data plane program designed for standard L2/L3 switching and routing (e.g., `switch.p4`) can easily take 10 or more stages (of the typically available 12 [28]). Moreover, more than half of the stages are densely utilized, leaving little available resources (e.g., memory) for other application logic to be merged in those stages. Unfortunately, synthesizing the state-of-the-art in-network HH detection algorithm PRECISION [21] alone requires at least 11 pipeline stages with substantial resource consumption, rendering integration of `switch.p4` with PRECISION infeasible (see details in §II-B). From a network operational perspective, the bread-and-butter switch functionalities always get prioritized over further extensions, making PRECISION impractical in any real network deployment.

In this work, we investigate the challenges of designing and implementing an in-network heavy-hitter detection algorithm that *(i)* reduces the number of HHs that current interval-reset solutions miss by design, and *(ii)* significantly reduces the overall hardware resource consumption (e.g., pipeline stages, memory) to enable integration with common switch functionalities, i.e., with `switch.p4`.

To this end, we propose `dSketch`, a novel time-decaying algorithm for accurate HH detection in the data plane, and implement it in the P4 language [17]. To address *(i)*, `dSketch` decays the counts instead of resetting them (§III-A). With a decaying function, `dSketch` retains a portion of historical counts for potential heavy flows throughout the time, while inactive/small flows are garbage-collected automatically (§III-B). We carry out trace-driven simulations and evaluations on Intel Tofino-based programmable switches (§IV) and show that `dSketch` significantly reduces the number of undetected HHs (i.e., high true positive rate) by $5-10\%$ over the state-of-the-art approaches (§IV-A). Addressing *(ii)*, `dSketch` uses only 4 pipeline stages and 4.7% of the switch SRAM (§III-C). This leaves ample resources for other concurrent data plane applications (§IV-B). Besides open-sourcing `dSketch` [29], we show its feasibility by integrating with the state-of-the-art switching function, `switch.p4`.

This paper makes the following contributions:

- We identify and elaborate on the shortcomings of existing heavy-hitter detection approaches (§II).
- We carefully design `dSketch` to overcome these limitations (§III).
- We evaluate the performance of `dSketch` and compare it to the state-of-the-art proposals (§IV). We show that `dSketch` can detect 5-10% more HH, with low false-positive rate and negligible packet recirculation (1-2%).

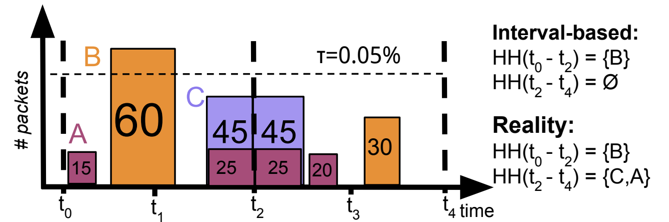


Fig. 2: Heavy flows A and C spanning across intervals go undetected with interval-reset algorithms. Here, $\tau = 0.05\%$ corresponds to 50 packets.

- We implement `dSketch` on an Intel Tofino-based programmable switch in P4 and make the code publicly available [29]. Furthermore, we show that our resource-efficient `dSketch` can be easily integrated with a complex, fully functional data plane program, such as `switch.p4`, positioning itself as an attractive network monitoring extension to already deployed in-network switching functions.

II. Motivations

Only a small number of flows account for the predominant proportion of the network traffic [30], and these important *elephant flows* or *heavy hitters (HH)* define some important network characteristics. The detection and identification of these flows are crucial to provide a high Quality of Service (QoS) and efficient load-balancing (see more details below).

Next, we discuss why and to what extent the interval-reset-based algorithms fall short of keeping track of the HH accurately. Then, we elaborate on further shortcomings of existing approaches designed explicitly for commodity programmable switches and define our design objectives accordingly.

A. Missing Heavy Hitters

Consider Fig. 2, where three different flows A, B, and C are shown as time passes within two intervals. In the network, an interval-reset algorithm is deployed, which resets the counts periodically at every t_i ($i = 2k, k \in \mathbb{N}$). The network operator's aim is to detect all heavy flows and carry out the necessary actions, e.g., rerouting. A flow is considered heavy, if the number of packets reaches the threshold $\tau = 0.05\%$ within an interval, i.e., within $2t$. Observe, in the first interval ($t_0 - t_2$), only flow B reaches the threshold, and the algorithm correctly reports it. Then, in the second interval ($t_2 - t_4$), both flows A and C go completely undetected due to the reset at t_2, however, they both exceed the threshold within a period of an interval size, i.e., within ($t_1 - t_3$). This renders interval-reset algorithms to be inefficient even in this simple use case.

On the other hand, every monitoring algorithm trying to realize patterns in a non-deterministic, forever-changing traffic inevitably misses some important observations (e.g., not detecting a HH) and/or can draw false conclusions (e.g., falsely reporting a small flow as heavy). Here, we argue that in the scope of heavy-hitter detection, while *keeping the number of falsely reported small flows (i.e., false positives) at a tolerable*

level, *minimizing the missing HH* (i.e., false negatives) is a superior aim for effectiveness. To underpin this claim, we briefly elaborate on some use cases for which heavy-hitter detection algorithms are typically deployed.

Anomaly detection. Since anomaly detection identifies possible threats based on the deviations from actual patterns and traffic load, detecting heavy hitters is a critical part of any security system. When heavy flows are detected (e.g., port scanners, DDoS attack), they are sent towards a network scrubber (e.g., IDS/IPS, firewall) that further processes the corresponding flows (e.g., filters out the ones found to be malicious [31]). Clearly, accidentally double-checking some benign small flows is acceptable (as they will not be filtered anyway), however, letting a significant amount of malicious flows simply pass through the network can pose serious security issues.

Load-balancer. HH can easily cause congestion that significantly reduces the overall QoS. Hence, we need to detect all heavy flows and carefully reroute them to optimize network utilization. Similarly to the anomaly detection, accidentally rerouting some small flows (e.g., false positives) does not introduce any negative side-effects, however, not rerouting a significant amount of HH may let the congestion remain.

Further applications such as traffic prioritization [32], [33], flow caching [3], in-network key-value stores [34], etc. share the same basic findings, i.e., minimizing false-negatives is crucial for efficient operation.

Accordingly, we define our first objective as follows.

OBJECTIVE 1. *Minimize the heavy flows going undetected while keeping the false-positives reasonably low.*

B. Hardware Resource Limitations

Besides the accuracy of an algorithm, we have further crucial requirements to satisfy to realize in-network heavy-hitter detection. Today's commodity programmable switches are PISA [35] switches based on the RMT [27] architecture, which enforces strict regulations due to the nature of its feed-forward packet processing pipeline. This can be described by having *(i)* fixed number of pipeline stages (e.g., 12 [28]), *(ii)* fixed amount of memory (SRAM) per stage (e.g., $\sim$1.4MB [27]), *(iii)* fixed number of parallel memory accesses per stage, and *(iv)* single-stage memory access [21], [35]. Next, we discuss *(i)-(iii)* in more detail. We later address *(iv) in §III-C*.

In a nutshell, the number of pipeline stages *(i)* required by an algorithm can be correlated with its length of the chain of dependencies. For instance, consider the canonical Count-Min Sketch (CMS [25]) with 4 rows for heavy-hitter detection. Fig. 3 depicts the mapping of CMS to the PISA switches' pipeline stages. First, whenever a packet arrives, we hash its flow ID (e.g., 5-tuple, source IP) in order to get 4 indexes to access the counters in every row (Stage 1). Next, in Stage 2, we update (i.e., increase) the corresponding counters in every row, and the new values are returned. Then, in Stage 3, we compare each pair of returned values [36] to get minimums, while in Stage 4, we calculate the real minimum out of the two obtained counts from Stage 3. Lastly, the obtained minimum count is checked against the threshold (Stage 5) to decide whether a flow is heavy (and to perform corresponding actions).

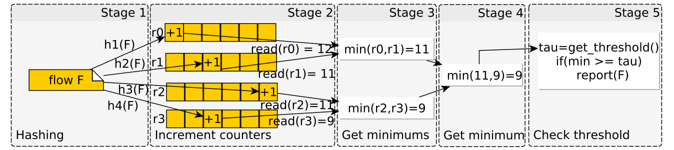


Fig. 3: The required number of different pipeline stages to keep track of heavy hitters with a CMS with four rows.

Whenever an algorithm *(ii)* requires more memory than a stage has, or *(iii)* performs more parallel lookups or memory accesses that allowed within one stage, the program will need to be split across multiple stages. For instance, if CMS in Fig. 3 *(i)* needs more counters per row and/or *(ii)* requires 6 rows, then, we need to split Stage 2 into further stages. Moreover, we may need more than two steps in order to get the final minimum count, resulting in at least 6 stages in total. On top of that, applications that depend on the output of the CMS further increases the length of the chain of dependency, requiring more stages to realize.

Therefore, HH detection algorithms have to be carefully designed to *not consume excessive amount of resources single-handedly* in order to enable integration with other data plane functionalities (e.g., switching and routing, anomaly-detection, DDoS defense). As a common baseline, we look at integrating algorithms with the commodity switch implementation, `switch.p4`, which eventually leads to our second objective.

OBJECTIVE 2. *Minimize resource consumption to foster integration with standard switch functionalities.*

III. dSKETCH: DESIGN AND IMPLEMENTATION

Next, we discuss in detail how the above-mentioned objectives can be satisfied. First, we introduce time-decaying algorithms and how they can overcome the limitations of interval-reset algorithms. Then, we formalize our novel algorithm `dSketch` and discuss in detail its implementation complying with the restrictions of PISA switches.

A. Time-decaying Algorithms

As discussed in §II, the fundamental cause of interval-reset algorithms missing the HHs is the periodical resets. To overcome this, we have to maintain the individual states (i.e., flow counts) *across intervals*. Consequently, we have to introduce the notion of time to the algorithm and carefully maintain the flows and their counts as time passes, i.e., preferring counts in the most recent interval over older ones.

One particular trick is to employ sliding windows. Some recent proposals for heavy-hitter detection (e.g., [37], [38]), however, are designed for software-based approaches only. This is due to the need for dynamic memory allocation, as well as requiring excessive memory accesses in order to obtain the heavy flows (e.g., iterating through the whole sliding window

to retrieve the counts). Thus, none of them is realizable on current commodity programmable switches [18], [19].

On the other hand, there already exist approaches like ConQuest [24] (originally designed for micro-burst detection, but we adapt it for heavy-hitter detection (§IV)) that attempts to realize sliding windows in the dataplane. ConQuest maintains multiple instances of the same data structure for different intervals and aggregates the counts accordingly. While satisfying **OBJECTIVE** 1, due to the significantly increased resource consumption, it goes against **OBJECTIVE** 2 (§IV-B). Thus, while interval-reset algorithms are inaccurate because of the periodic loss of information, sliding window-based approaches naïvely operating on top of multiple (data structure instances for multiple) intervals easily ends up in significant resource consumption.

So, how do we track flow information across multiple intervals without maintaining extra state? — We decay the counts as time passes instead. There have been several works on efficiently answering streaming queries using time-decaying algorithms, from which, exponential decay has been widely used in the domain of streaming data aggregation [39]–[42]. As time-decaying functions rely on floating point operations, to the best of our knowledge, they have yet to be explored in the realm of commodity programmable switches. This is mainly due to the lack of support for floating point operations in PISA switches [35], [36]. Hence, we attempt to achieve approximate time-decaying functions in the dataplane by utilizing the available primitive arithmetic and logical operations available; we found that binary shifting is the closest approximation to exponential decay. Hence, we propose Approximate Exponential Decay (AED) as a possibility to realize the basics of time-decaying in commodity programmable switches (initially presented briefly in [43]). Note, by decaying we approximate the operation of a canonical sliding window. Particularly, we dynamically manage an independent sliding window for each flow, i.e., a heavy flow will be retained as long as its count is sufficiently high and new, but at the price of introducing a slight estimation error in the true counts.

With time-decaying, we satisfy **OBJECTIVE** 1 (see in §IV) by letting only a negligible amount of HH go undetected.

1) Keeping Track of Time: Since for time-decaying we need to be aware of the time, first, we discuss how we can keep track of time in the data plane. High-resolution clocks (typically up to 48-bit timestamps with ns precision [44]) are available in PISA switches and can be accessed during packet processing, thereby making it possible to introduce the notion of time to our algorithms. Instead of the explicit timestamps, however, we define broader observation phases to keep track of intervals for which we extract n bits of the timestamps, e.g., 8 bits (bit 40 to 33) to obtain $\sim$8.6-second periods. By having these explicit but shortened timestamps, we can easily keep track of the intervals.

B. `dSketch`: The Algorithm

Following our findings in §III-A, we select Count-Min Sketch [25] as our foundation due to its lightweight property, and propose `dSketch` (time-decaying sketch) on top.

Algorithm 1 Update in `dSketch`, *Input:* incoming packet's data F, set of independent hash functions $\mathcal{H}_e$, Ω actual discretized timestamp, γ interval difference after garbage collection kicks in.

```
 1: procedure UPDATE(F, Ω)
 2:   F.count ← ∞                          # init count for flow F
 3:   for i ← 1 to e do                    # for each row
 4:     c, ω ← DSKETCH[H_i(F)]             # get counter and timestamp
 5:     Δ_Ω ← Ω − ω                         # timestamp difference
 6:     if Δ_Ω ≠ 0 then                     # not updated recently
 7:       if Δ_Ω ≥ γ then                   # counter is too old
 8:         c ← GC(c)                       # garbage collection
 9:       else
10:         c ← AED(c, Δ_Ω)                 # decay counter
11:       end if
12:     end if
13:     c ← c + 1                           # Increment counter
14:     DSKETCH[H_i(F)] ← c, Ω              # update counter
15:     F.count ← MIN(c, F.count)           # get min count for F
16:   end for
17:   return F.count                        # estimated count for F
18: end procedure
```

Instead of merely focusing on a sole observation phase (i.e., an interval) or maintaining multiple instances of the same sketch, we decay the (true) counts of the flows as time passes. To reach this, first, we introduce the notion of time by augmenting every counter with an 8-bit auxiliary field to maintain discretized timestamps (cf. §III-A1). The stored timestamps keep track of in which interval the counter was last updated. To retain counts for the most recent intervals only, we perform the approximate exponential decay function whenever the discretized timestamp advances. If a particular counter has not been updated for a while, i.e., within the last 2 intervals, then the garbage-collection process will zero it out. This mechanism protects the data structure from overestimation.

In Alg. 1, we discuss how `dSketch` works; F is the flow ID of the actual packet, Ω marks the actual interval (i.e., the current global discretized timestamp), while γ equals to the maximum timestamp difference to Ω within a certain flow's count is *not* zeroed out but decayed. To put it briefly, the function DSKETCH in this algorithm reads and writes the sketch data structure (like in CMS [25]). For every row in `dSketch` (line 3-16), we retrieve the corresponding counter value (c) and its last updated timestamp (ω) by hashing F (line 4). Then, we compare Ω and ω (line 6) and apply the corresponding actions, i.e., either garbage-collection or applying the approximate exponential decay function (AED) accordingly to the timestamp difference (line 6-12). Flows that have not been updated within the last γ intervals will be garbage collected. Lastly, we increment c by 1 and update the counter together with the latest global timestamp, Ω (line 13-14). While updating every row, we keep track of the minimum counts which will represent the actual count of flow F. Depending on the outcome of UPDATE, HH are reported to the control plane via push-based mechanisms [45], [46].

C. dSketch: The Implementation

Next, we discuss in detail how the proposed algorithm of dSketch (§III-B) can be implemented in P4 and what necessary steps we need to make to satisfy **OBJECTIVE** 2.

Taking a closer look at Alg. 1, we can immediately identify that line 6-14 violates the memory access patterns of PISA [21], [27], [35]. For an easier comprehension, we break down line 4-14 into separate parts for analysis by arranging them into stages[2] as follows.

Stage n: First, we retrieve c and ω (and its difference from Ω) from the memory. This takes one pipeline stage (line 4-5).

Stage $n+1$: Then, based on the difference, Δ_Ω, we apply different conditional actions to the counter value c (line 6-12).

Note, there is a dependency between the operations on c resulting in the two different pipeline stages n and $n+1$ (cf. Fig. 3 in §II-B). This becomes a problem when we increment and assign the new value back to the counter with the latest global timestamp (c, ω) later in a subsequent stage $n+2$ (line 13-14) as we would access the same memory region again but from a different stage, i.e., first at stage n, then at $n+2$. This violates the single-stage memory access property of the PISA switches [21], [27], [35] making Alg. 1 infeasible to be implemented directly on commodity programmable switches.

To overcome this, we utilize packet recirculations (similarly to [21]) to update the counters with the decayed values (i.e., for line 14). The revised algorithm is shown in Alg. 2, and the corresponding stages are discussed below.

Stage n: When *regular* packets are processed, we increment counter c of flow F immediately, and retrieve c and ω (and its difference from Ω) from the memory (line 8-11). On the other hand, if we receive a *recirculated* packet (containing the new and decayed values as metadata) we update the corresponding counter and timestamp values accordingly (line 4-5).

Stage $n+1$: If we need to decay a particular count (line 13-20), instead of trying to write it back to the memory (which we cannot), we use the new values as metadata (stored in form of an additional custom header), clone the packet, and then recirculate the cloned copy. Note, by cloning we avoid packet reordering, and we maintain application-level performance as the original packets are routed as usual, while the cloned packet is dropped after it finishes its second round (line 6).

Observe that recirculation is only needed if a particular flow's count needs to be decayed or zeroed out. Therefore, the number of recirculations dSketch needs is equal to the number of different flows within an interval in the worst-case, i.e., if all flows needs to be decayed or zeroed out.

Packet recirculation is a common practice for applications to access a given memory region more than once. For instance, HH detection [20], [21], packet scheduling [47], payload parking for network functions [48] or even as simple as MPLS packet processing recirculate packets (multiple times) to re-access specific memory blocks. However, this ultimately penalizes the available switch backplane capacity; hence, it has to be minimized to maintain line-rate throughput. We find that dSketch recirculates only 2.3%, 2.6%, and 1.8% of the packets on CAIDA16, CAIDA18, and CAIDA19 traces, respectively in contrast to the already negligible amount of ∼3% of PRECISION [21]. Note, under full network load of 6.4 Tbps, recirculating 2% of the packets only consume ∼50% of the switch's shared recirculation port's throughput, which we believe to be reasonable. Hence, for brevity, we do not discuss recirculation further in §IV.

[2] For brevity, we do not discuss the previous and subsequent stages that do not violate any restriction.

Algorithm 2 Update in dSketch, *Input:* same as Alg. 1

1: **procedure** UPDATE(F, Ω)
2: $F.count \leftarrow \infty$ *# init count for flow F*
3: **for** $i \leftarrow 1$ to e **do** *# for each row*
4: **if** $F.recirculated$ **then**
5: $\text{DSKETCH}[\mathcal{H}_i(F)] \leftarrow F.tmp_{c_i}, \Omega$
6: $\text{DROP}(F)$ *# drop cloned packet*
7: **else**
8: $c, \omega \leftarrow \text{DSKETCH}[\mathcal{H}_i(F)]$ *# get counter and timestamp*
9: $c \leftarrow c + 1$ *# increment counter*
10: $\text{DSKETCH}[\mathcal{H}_i(F)] \leftarrow c, \omega$ *# update counter*
11: $\Delta_\Omega \leftarrow \Omega - \omega$ *# timestamp difference*
12: $F.count \leftarrow \text{MIN}(c, F.count)$ *# get min count for F*
13: **if** $\Delta_\Omega \neq 0$ **then** *# not updated recently*
14: **if** $\Delta_\Omega \geq \gamma$ **then** *# counter is too old*
15: $tmp_{c_i} \leftarrow \text{GC}(c)$ *# garbage collection*
16: **else** *# counter is not fresh*
17: $tmp_{c_i} \leftarrow \text{AED}(c, \Delta_\Omega)$ *# decay counts*
18: **end if**
19: $\text{CLONE_AND_RECIRCULATE}(F, tmp)$
20: **end if**
21: **end if**
22: **end for**
23: **return** $F.count$ *# estimated count for F*
24: **end procedure**

To summarize, we successfully implemented Alg. 2, and verified that dSketch works as intended on the Intel Tofino-based commodity programmable switch, and in §IV we show that it satisfies **OBJECTIVE** 2. In §IV-B, we also show that dSketch leaves ample resources to be integrated with standard switch functionalities, i.e., with switch.p4. All technical details on the implementation are available at [29].

IV. EVALUATION

Next, we evaluate the performance of dSketch and compare it against the state-of-the-art. In line with **OBJECTIVE** 1, we focus on the true-positive rate of the algorithms as a performance indicator, i.e., we investigate how many of the true HH are indeed detected without falsely reporting too many small flows (low false-positive rate). Note, due to the resets, interval-reset algorithms inevitably reduce the chances to falsely report a small flow as heavy. Correspondingly, we expect the interval-reset algorithms to have the lowest false-positive rates. Then, we turn to **OBJECTIVE** 2.

Methodology. For evaluation purposes, we implement a Python-based trace-driven simulator (similarly to [20]–[23], [26]) which mimics the behavior of the P4 implementations of all evaluated algorithms. This allows us to tune and configure the algorithms easily for each use case. Furthermore, we also

implement dSketch in P4, and present results when running it on an Intel Tofino-based commodity programmable switch. In this regard, for the other algorithms, we rely on the available source-codes [49], [50].

Defining Heavy Hitters. The problem of heavy-hitter detection is considered well-known, however, it is easy to be lost among the different definitions. HH can mean the top-k flows by size, however, from a network engineering point of view, being in the top-k does not necessarily mean they are really heavy flows; they can simply be greater than others, but the magnitude may be negligible. In contrast to such a predefined static threshold, HH can also mean the elephant flows that are proportionally larger (in number of packets/bytes) than a fraction τ of the total traffic. Here, we focus on the latter definition, and we choose $\tau = 0.05\%$, i.e., a flow is considered to be heavy if its size (in terms of packets) exceeds 0.05% of the overall number of packets. As a reference, this percentage results in (on average) ~700, ~650, ~750 HH in CAIDA16, CAIDA18, and CAIDA19 traces, respectively.

Groundtruth. We define the groundtruth as the set of flows that exceeds threshold τ, at any point of time within the last $t = 8$ seconds, which is the closest approximate discrete timestamp we can obtain from the high-resolution clocks of today's commodity programmable switches (cf. §III-A1). We use the IP 5-tuple as the flow IDs.

Traffic Traces. We utilize three 1-hour-long anonymized ISP backbone traces from CAIDA collected in 2016 (CAIDA16 [51]), 2018 (CAIDA18 [52]), and 2019 (CAIDA19, [53]). In general, the CAIDA16, CAIDA18, and CAIDA19 consist of ~1.7 billion packets, ~1.5 billion packets, and ~2.3 billion packets, respectively. Within one second, the traces contain ~$464k$ packets (~$12k$ unique flows), ~$424k$ packets (~$20k$ unique flows), and ~$635k$ packets (~$40k$ unique flows), respectively.

Algorithms to Compare. To analyze the performance of dSketch, we compare it against the state-of-the-art in-network solutions, i.e., we only consider algorithms designed for PISA switches that can *indeed* run in the data plane and do not require any third-party (e.g., collaborative end-hosts [54]) for operation. In particular, we compare dSketch to its basis, CMS [25] and to FCM-Sketch [23], [50] the most recent sketch-based approach for in-network heavy-hitter detection that outperforms other state-of-the-art, e.g., UnivMon [26], ElasticSketch [22]. Furthermore, we also compare dSketch to PRECISION [21], [49], the state-of-the-art hash table-based proposal. Finally, we adapt the sliding window-based ConQuest [24], [49], originally proposed for microburst detection, to HH detection (HH-CQ) and compare it against dSketch.

Algorithm Configurations. We fix ~600kB[3] of memory for each of the algorithms and find the maximal data plane configuration that can best approximate the allocated memory. This is because the number of counters required have to be in the power of 2 (e.g., $2^{15} = 32K$) in order to efficiently utilize the available hardware resources, and different algorithms manage the data structures in different ways (e.g., PRECISION stores flow IDs, HH-CQ uses multiple instances of the same data structure). Accordingly, this setting results in the following configurations for each algorithm:

- **CMS:** 2 rows, $64K$ counters per stage, 32 bits per counter (512kB).
- **PRECISION:** 2-stage data structure, $16K$ entries for per stage, 136 bits per entry (544kB).
- **FCM-Sketch:** 8-ary variant[4] of the algorithm with $256K$, $32K$, $4K$ 8, 16, 32-bit counters, respectively (672kB).
- **HH-CQ:** 4 instances[5], $32K$ counters per instance, and 32 bits per counter (512kB).
- **dSketch:** 2 rows, $64K$ counters per stage, 32 bits per counter and 8-bit timestamps for each counter (640kB).

A. OBJECTIVE 1: *Detection Rate of True Heavy Hitters*

In this section, we focus **OBJECTIVE** 1, i.e., we analyse the true-positive rate (Recall) of all algorithms in all traces mentioned above.

Trace-driven simulations. The results are depicted in Fig. 4. Note, the Recall on the y axis shows the ratio of all true HH detected by each algorithm (cf. Fig. 4a). Accordingly, 1-Recall means the actual percentages of the missed HH, e.g., FCM-sketch misses 15% of the HH on the CAIDA16 trace. Observe, all state-of-the-art interval-reset-based algorithms miss around $10 - 15\%$ of the heavy hitters. On the other hand, the adapted sliding window-based HH-CQ has 0% missing HH, while dSketch provides a reasonable high recall of more than 95% irrespective of the traces. Later, in §IV-B, we show that the outstanding recall rate of HH-CQ pays a huge penalty in terms resource consumption, which violates **OBJECTIVE** 2.

In terms of falsely reported small flows, all algorithms have a negligible false-positive rate. In particular, we confirm that due to the resets, on average, the sketch-based algorithms FCM, CMS have the lowest values in the scale of 10^{-6}. In contrast, the hash table-based PRECISION and the sliding window-based HH-CQ are in the scale of 10^{-3}. Lastly, dSketch lies in the middle with an FPR rate in the scale of 10^{-4}.

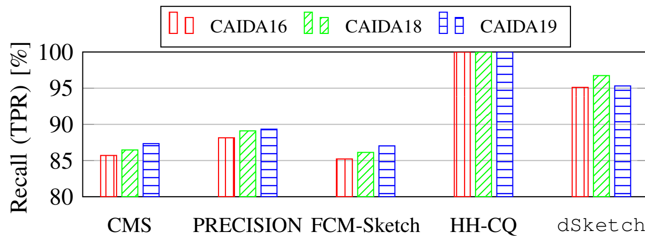
Evaluation on Hardware. Next, we evaluate how dSketch performs on the Intel Tofino-based programmable switch. The purpose of this study is to verify the results we obtained via our trace-driven simulator while the other algorithms have been verified on hardware in their respective studies.

The results depicted in Fig. 5 shows the Recall for dSketch obtained on the Tofino switch via the green bars (grid pattern). We also show the simulated results via the red bars (hrz. line pattern) for comparison. Besides, in Fig. 5b, the TNR (1-FPR) values are depicted for reference. Observe, the accuracy metrics are very close to each other with around $1 - 1.5\%$ difference on average for recall, and around 10^{-5} difference for the false-positive rates.

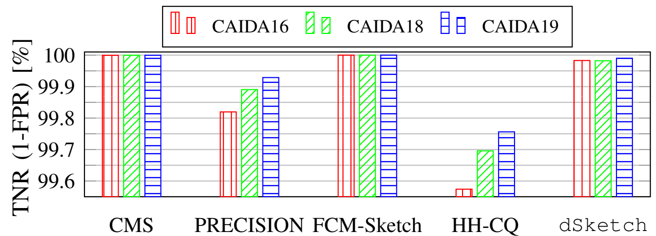
[3]Note, we also analyzed the algorithms with more allocated memory (e.g., 1MB, 2MB), however, the overall performance did not changed significantly. Hence, for brevity, we omit to discuss this in detail.

[4]The authors recommend the 8-ary variant for heavy-hitter detection [23].

[5]Four is the bare-minimum for snapshot instances as it needs to be in the form of $2^x : x \geq 2$ [24].

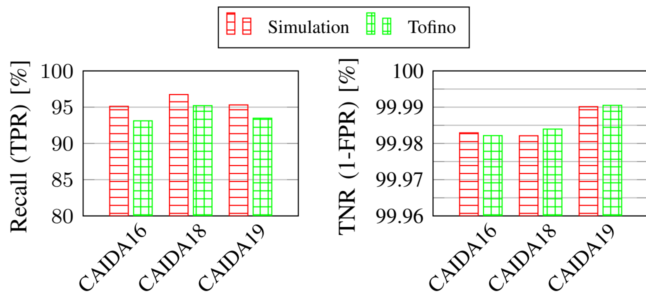


(a) True-positive rate of the algorithms



(b) True-negative rate of the algorithms

Fig. 4: Performance of the different algorithms



(a) True-positive rate of `dSketch` (b) True-negative rate of `dSketch`

Fig. 5: Verifying `dSketch` on the Intel Tofino hardware using different traffic traces.

TABLE I: Hardware resource consumption on Tofino.

Resource	switch.p4	CMS	FCM	PREC.	HH-CQ.	dSketch
SRAM	43.50%	**3.80%**	5.20%	5.60%	5.10%	4.70%
Match Xbar	21.30%	**1.10%**	2.90%	7.60%	3.20%	2.50%
TCAM	34.00%	**0.70%**	**0.70%**	3.50%	**0.70%**	**0.70%**
SALU	14.60%	**4.17%**	12.50%	20.83%	16.67%	8.33%
Hash Bits	27.30%	**1.20%**	2.60%	7.10%	4.30%	3.50%
VLIW Actions	18.00%	**1.30%**	2.60%	7.30%	6.80%	3.10%
Pipeline Stages	10	**4**	6	11	8	**4**

The reason for the slight difference is that, due to the timestamped `pcap` file, the trace-driven simulation can be very accurate compared to the ground truth. Whereas the real-world evaluations are affected by several further factors. First, the traffic trace is replayed from a server (directly connected to the Tofino switch), and the processing is done on the Tofino switch inducing a slight processing [55] and transmission delay. Second, the trace is replayed via the tool `tcpreplay`, which cannot precisely follow the timestamps in the `pcap` file[6]. Therefore, for each trace we select a certain packet-per-second rate which makes `tcpreplay` to replay the trace within 60 seconds. However, this constant rate makes the inter-packet arrival times uniform.

B. OBJECTIVE 2: *Resource Consumption*

Table I shows the hardware resource consumption of all algorithms when implemented and run on Tofino[7]. As a baseline, we also show how much resources `switch.p4`, the standard switch functionality, consumes. We conclude the following. Since `dSketch` is based on CMS, `dSketch` obviously requires more resources than CMS, however, not that much (on average, only $\sim$1.5% more). On the other hand, when compared to its other counterparts, `dSketch` on average requires much less resources in all aspects. Finally, and most importantly, `dSketch` requires the same small number of pipeline stages as CMS making it a good candidate to be integrated with other concurrent applications, such as `switch.p4` (see later). Besides `dSketch`, FCM requires the least number of pipelines stages (6), while HH-CQ and PRECISION need 8 and 11, respectively.

Next, we scrutinize the integration aspects of all algorithms, i.e., whether they can co-exists with the fundamental switching application `switch.p4` on the same Tofino device. To reach this end, we use the reference commodity switch implementation provided in the Intel P4 Studio, `switch.p4` and the source codes available for the heavy-hitter detection applications [49], [50]. After analyzing packet processing logic of both applications, we try to integrate them together. However, this is a challenging and error-prone task [56], [57] due to the monolithic nature of P4 programs. Note, while none of the individual resource groups (e.g., SALU) is fully utilized by each algorithm, the length of the chain of dependency, i.e., the number of pipeline stages plays a vital role in determining whether the integration with `switch.p4` is possible at all. Accordingly, upon integration, we found that HH-CQ and PRECISION cannot be merged with `switch.p4` as the resulting application would require much more than 12 stages. On the other hand, `dSketch`, CMS and FCM-sketch were successfully integrated; all three still leaving 1 extra stage and free resources for additional applications (e.g., packet marking, load-balancing). The detailed results are shown in Table II. Note that considering only the applications satisfying **OBJECTIVE** 2 (i.e., CMS, FCM-sketch, and `dSketch`), `dSketch` detects $5-10\%$ more HH than the other solutions.

C. *Summary*

We summarize the results obtained for all algorithms w.r.t. the **OBJECTIVES** (cf. §II) in Table III. The first column shows

[6]In our testbed having Intel Xeon Gold 6230-based servers and Mellanox ConnectX-5 100G NICs, it takes $\sim$80 seconds for `tcpreplay` to replay the 60-second trace without specifying explicit pace.

[7]All P4 programs are compiled using Intel P4 Studio (Version 9.3.0) with the default flags.

TABLE II: Hardware resource consumption after integration.

Resource	switch.p4+CMS	switch.p4+FCM	switch.p4+dSketch
SRAM	46.8%	50.1%	49%
Match Xbar	21.4%	23.3%	23.8%
TCAM	34.7%	34.7%	34.7%
SALU	18.8%	27.1%	22.9%
Hash Bits	28.5%	28.4%	30.1%
VLIW Actions	20.1%	19.5%	19.5%
Pipeline Stages	11	11	11

TABLE III: Summary

Algorithm	OBJECTIVE 1	OBJECTIVE 2
CMS	12.7%	✓ (Low)
PRECISION	10.7%	✗ (High)
FCM-Sketch	13.0%	✓ (Medium)
HH-CQ	0%	✗ (High)
dSketch	4.7%	✓ (Low)

the ratio of the missed heavy hitters, the second column shows whether the algorithm can be integrated with `switch.p4` as well as the overall resource consumption in the scale of `Low`, `Medium`, and `High`. Here, we mostly focus on the number of pipeline stages and the resource consumption for the data structure itself (SALU in Table I). If the former is within the third of the available pipeline stages, and the memory footprint for the SALU is below 10%, the algorithm's resource consumption is considered `Low`. Accordingly, if an algorithm requires more than 4 but less than 6 pipeline stages, and less than 20% utilization of SALU, it is considered `Medium`. Anything above these numbers are considered `High`.

We conclude that the current state-of-the-art interval-reset algorithms fall short in satisfying **OBJECTIVE** 1. Furthermore, the hash-table-based PRECISION cannot be integrated with further applications due to its substantial resource consumption. While the sliding window-based HH-CQ is the most accurate heavy-hitter detection algorithm, it also does not leave sufficient space for other applications. This renders our time-decaying algorithm, `dSketch` as a decent solution for in-network heavy-hitter detection that not only captures almost all HH (**OBJECTIVE** 1) but can also be integrated with `switch.p4` (**OBJECTIVE** 2).

V. RELATED WORK

Commodity Programmable Switches. The emergence of programmable commodity switches (or commonly known as PISA switches [18], [19]) introduce new possibilities for realizing fine-grained network monitoring approaches in the data plane. However, the restricted programming model [27] brings up a new set of challenges (e.g., limited hardware resources, strict memory access patterns) in materializing them.

Entirely in the data plane. Recent works [45], [46] have demonstrated the significance of performing detection entirely in the data plane to achieve low-latency reactions upon network events. Instead of relying on the control plane for periodic statistics retrieval, [45] proposes a push-based approach that leverages data plane programmability that interacts with the control plane whenever certain network event conditions are triggered. [46] has highlighted that in order to reduce the network event detection to nano-second scale, many networking tasks have to be offloaded to the data plane. In this paper, we build upon the same line of thought in designing in-network monitoring solutions, however, besides detecting almost all heavy hitters, we put additional emphasis on the resource consumption, leaving ample resources for other networking tasks, such as for the common switch funtionality (`switch.p4`) to be realized in parallel.

Interval-resets. In-network monitoring solutions, like UnivMon [26], HashParallel [20], PRECISION [21], FCM-Sketch [23] are designed to be reset periodically by using the control plane. We discussed the issues (i.e., the heavy hitters that span across intervals going undetected) with interval-resets in §I and §II. Former literature have also highlighted this concern [58]. This motivates the need for designing data structures that can keep track of time and maintain counts longer in a more sophisticated way.

Sliding windows. Sliding window-based approaches, such as WCSS [37] and its successor, the probabilistic Memento [38], can be used to detect HH quickly and accurately. However, both are designed for software implementations with dynamic memory allocations and accesses in mind, thus ruling out their realizations in commodity programmable switches. Sequential Zeroing [59] and ConQuest [24] realize sliding windows-alike data structures on hardware using multiple instances of sketches [25] that are being read, written, cleared from time to time. However, Sequential Zeroing is designed for SmartNICs [60] which more relaxed constraints than PISA switches while ConQuest was specifically designed for micro-bursts detection in data centers using PISA switches. We adapt ConQuest by scaling the memory and timescales for heavy-hitter detection in our evaluations.

Time-decay. There have been several related works on the topic for efficiently answering streaming queries using time decays (e.g. polynomial, finite and exponential), for which, exponential decay was widely used in the domain of streaming data aggregation [39]–[42]. However, due to the lack of support for floating point operations and the restricted programming model [27], such decay functions have yet to be explored. Hence, our proposed `dSketch` with approximate exponential decaying represents one of its first realization in existing commodity programmable switches.

VI. CONCLUSION

In this paper, we analyze the state-of-the-art in-network heavy-hitter detection algorithms, and identify two major traits: most of them being an interval-reset algorithm, i.e., they let a significant amount of heavy hitters spanning across interval go undetected, and do not leave sufficient resources for concurrent applications. Taking them into consideration, we design a lightweight, time-decaying algorithm, `dSketch`, likened to an approximate sliding window, for online heavy-hitter detection entirely in the data plane.

Extensive experiments using real-world traffic traces on both simulation and hardware show that `dSketch` significantly increases (on average, by 5-10%) the detection rate of true heavy hitters as compared to the state-of-the-art without a notable false-positive rate. Moreover, `dSketch` offers an attractive alternative for data plane designers due to its resource and operational efficiency, and feasibility to be integrated with existing data plane programs, such as the commodity switch implementation, `switch.p4`.

Acknowledgements

This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Corporate Laboratory@University Scheme, National University of Singapore, and Singapore Telecommunications Limited.

References

[1] T. Benson *et al.*, "MicroTE: Fine Grained Traffic Engineering for Data Centers," in *ACM CoNEXT*, 2011.
[2] M. Alizadeh *et al.*, "pFabric: Minimal near-Optimal Datacenter Transport," *ACM SIGCOMM CCR*, vol. 43, no. 4, p. 435–446, Aug. 2013.
[3] N. Katta *et al.*, "CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks," in *ACM SOSR*, 2016.
[4] M. Alizadeh *et al.*, "CONGA: Distributed Congestion-Aware Load Balancing for Datacenters," *ACM SIGCOMM CCR*, vol. 44, no. 4, p. 503–514, Aug. 2014.
[5] S. K. Singh *et al.*, "Revisiting Heavy-Hitters: Don't Count Packets, Compute Flow Inter-packet Metrics in the Data Plane," in *ACM SIGCOMM Demos and Posters*, 2020.
[6] Q. P. Nguyen *et al.*, "GEE: A Gradient-based Explainable Variational Autoencoder for Network Anomaly Detection," in *IEEE CNS*, 2019.
[7] I. Nevat *et al.*, "Anomaly Detection and Attribution in Networks With Temporally Correlated Traffic," *IEEE/ACM ToN*, vol. 26, no. 1, pp. 131–144, Feb 2018.
[8] X. Z. Khooi *et al.*, "DIDA: Distributed In-Network Defense Architecture Against Amplified Reflection DDoS Attacks," in *IEEE NetSoft*, 2020.
[9] A. Laraba *et al.*, "Defeating Protocol Abuse with P4: Application to Explicit Congestion Notification," in *IFIP Networking*, 2020.
[10] R. Sommer *et al.*, "NetFlow: Information Loss or Win?" in *ACM SIGCOMM IMW*, 2002.
[11] E. F. Castillo *et al.*, "IPro: An Approach for Intelligent SDN Monitoring," *Comput. Netw.*, vol. 170, Apr. 2020.
[12] H. Yahyaoui *et al.*, "On Providing Low-cost Flow Monitoring for SDN Networks," in *IEEE CloudNet*, 2020.
[13] L. Csikor *et al.*, "End-host Driven Troubleshooting Architecture for Software-defined Networking," in *IEEE GlobeCom*, 2017, pp. 1–7.
[14] P. Phaal, "sFlow Sampling Rates," Blogpost, https://blog.sflow.com/2009/06/sampling-rates.html, 2009.
[15] N. Duffield *et al.*, "Estimating Flow Distributions from Sampled Flow Statistics," in *ACM SIGCOMM*, 2003.
[16] P. G. Kannan *et al.*, "On Programmable Networking Evolution," *CSI Transactions on ICT*, vol. 8, no. 1, pp. 69–76, Mar 2020.
[17] P. Bosshart *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM CCR*, vol. 44, no. 3, p. 87–95, Jul. 2014.
[18] Intel, "Intel® Tofino™," https://bit.ly/3sY7beD, [Accessed: Sep 2020].
[19] Intel, "Intel® Tofino™ 2," https://bit.ly/3oer5hX, [Accessed: Sep 2020].
[20] V. Sivaraman *et al.*, "Heavy-Hitter Detection Entirely in the Data Plane," in *ACM SOSR*, 2017.
[21] R. Ben Basat *et al.*, "Designing Heavy-Hitter Detection Algorithms for Programmable Switches," *IEEE/ACM ToN*, vol. 28, no. 3, pp. 1172–1185, 2020.
[22] T. Yang *et al.*, "Elastic Sketch: Adaptive and Fast Network-Wide Measurements," in *ACM SIGCOMM*, 2018.
[23] C. H. Song *et al.*, "FCM-Sketch: Generic Network Measurements with Data Plane Support," in *ACM CoNEXT*, 2020.
[24] X. Chen *et al.*, "Fine-Grained Queue Measurement in the Data Plane," in *ACM CoNEXT*, 2019.
[25] G. Cormode *et al.*, "An Improved Data Stream Summary: The Count-min Sketch and Its Applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.
[26] Z. Liu *et al.*, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," in *ACM SIGCOMM*, 2016.
[27] P. Bosshart *et al.*, "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN," *ACM SIGCOMM CCR*, vol. 43, no. 4, p. 99–110, Aug. 2013.
[28] "A Deeper Dive Into Barefoot Networks Technology." https://bit.ly/3iKB5i3, 2017.
[29] "dSketch," https://bit.ly/3sTFWS9.
[30] Y. Zhang *et al.*, "On the Characteristics and Origins of Internet Flow Rates," in *ACM SIGCOMM*, 2002.
[31] Cisco, "Effective DDoS Mitigation in Distributed Peering Environments," White paper, https://bit.ly/2taak0P, 2018.
[32] R. Soulé *et al.*, "Merlin: A Language for Provisioning Network Resources," in *ACM CoNEXT*, 2014.
[33] D. M. Divakaran, "A Spike-Detecting AQM to Deal with Elephants," *Comput. Netw.*, vol. 56, no. 13, p. 3087–3098, Sep. 2012.
[34] X. Jin *et al.*, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *ACM SOSP*, 2017.
[35] N. Gebara *et al.*, "Challenging the Stateless Quo of Programmable Switches," in *ACM HotNets*, 2020.
[36] Intel, "Open-Tofino," https://bit.ly/3mtpzJz, [Accessed: Apr 2021].
[37] R. Ben-Basat *et al.*, "Heavy Hitters in Streams and Sliding Windows," in *IEEE INFOCOM*, 2016.
[38] R. Ben-Basat *et al.*, "Memento: Making Sliding Windows Efficient for Heavy Hitters." in *ACM CoNEXT*, 2018.
[39] S. Z. Sbz *et al.*, "The Aurora and Medusa Projects," in *IEEE Data Eng. Bull.*, ser. 26, 2003.
[40] S. Chandrasekaran *et al.*, "TelegraphCQ: Continuous Dataflow Processing," in *ACM ICMD*, 2003.
[41] C. Cranor *et al.*, "Gigascope: A Stream Database for Network Applications," in *ACM SIGMOD*, 2003.
[42] G. Cormode *et al.*, "Forward Decay: A Practical Time Decay Model for Streaming Systems," in *IEEE ICDE*, 2009.
[43] X. Z. Khooi *et al.*, "Towards In-Network Time-Decaying Aggregates for Heavy-Hitter Detection," in *ACM SIGCOMM Demos and Posters*, 2020.
[44] P4 Arch. WG, "PSA," https://bit.ly/3plPuUd, Mar 2018.
[45] J. Kučera *et al.*, "Enabling Event-Triggered Data Plane Monitoring," in *ACM SOSR*, 2020.
[46] S. Wang *et al.*, "Martini: Bridging the Gap between Network Measurement and Control Using Switching ASICs," in *IEEE ICNP*, 2020.
[47] N. K. Sharma *et al.*, "Programmable Calendar Queues for High-speed Packet Scheduling," in *USENIX NSDI*, Feb. 2020.
[48] S. Goswami *et al.*, "Parking Packet Payload with P4," in *ACM CoNEXT*, 2020.
[49] "Princeton Cabernet," https://bit.ly/2Yfs9b6.
[50] C. H. Song, "FCM P4," https://bit.ly/3iM84T2, 2020.
[51] CAIDA, "The CAIDA UCSD Anonymized Internet Traces - 2016 March," https://bit.ly/2YeeL6V, [Accessed: Sep 2020].
[52] CAIDA, "The CAIDA UCSD Anonymized Internet Traces - 2018 March," https://bit.ly/3qRuHbE, [Accessed: Sep 2020].
[53] CAIDA, "The CAIDA UCSD Anonymized Internet Traces - 2019 January," https://bit.ly/3qO6B1p, [Accessed: Sep 2020].
[54] Q. Huang *et al.*, "OmniMon: Re-Architecting Network Telemetry with Resource Efficiency and Full Accuracy," in *ACM SIGCOMM*, 2020.
[55] H. Harkous *et al.*, "P8: P4 with Predictable Packet Processing Performance," *IEEE TNSM*, 2020.
[56] H. Soni *et al.*, "Composing Dataplane Programs with μP4," in *ACM SIGCOMM*, 2020.
[57] K. Birnfeld *et al.*, "P4 Switch Code Data Flow Analysis: Towards Stronger Verification of Forwarding Plane Software," in *IEEE/IFIP NOMS*, 2020.
[58] S. Galea *et al.*, "Revealing Hidden Hierarchical Heavy Hitters in Network Traffic," in *ACM SIGCOMM Posters and Demos*, 2018.
[59] B. Turkovic *et al.*, "Sequential Zeroing: Online Heavy-Hitter Detection on Programmable Hardware," in *IFIP Networking*, 2020.
[60] Netronome, "Agilio CX SmartNICs," https://bit.ly/3pnZTyM.