Ring Sketch: A generic, Low-complexity, And Hardware-friendly Traffic Measurement Framework Over Sliding Windows

Gen Li[†], Xiong Wang*[†], Congqi Zhao[†], Zhiyu Yang[†], Jing Ren[†], Sheng Wang[†], Shizhong Xu[†]
[†]University of Electronic Science and Technology of China, Chengdu, China

Abstract—**Traffic measurement is essential for network management. Sliding window models can provide network management tasks with flow statistics within the most recent window at any moment. However, most existing solutions over sliding windows are not designed for traffic measurement scenarios, therefore they have higher complexity and cannot be implemented on programmable hardware switches. To address the issues, we designed Ring Sketch, which is a generic, low-complexity, and hardware-friendly traffic measurement framework over sliding windows. Ring Sketch can not only be easily implemented on programmable hardware switches but can also accurately answer typical flow statistics queries by using different sketches. Then we propose the estimation strategy for Ring Sketch and theoretically analyze its error bounds. At last, we implement Ring Sketch on OVS-DPDK and a programmable hardware switch with a Tofino 1 chip, and all the source codes are released on GitHub. The experimental results show that Ring Sketch has a throughput over 3x higher than the state-of-the-art Sliding Sketch, and in typical measurement tasks, Ring Sketch can achieve high measurement accuracy.**

Index Terms—**Traffic measurement, Sketch, sliding window, programmable network.**

I. Introduction

Traffic measurement is responsible for extracting the interested flow statistics from the packet stream at each network node. Traffic measurement is essential for many network management tasks (e.g., traffic engineering, congestion control, anomaly detection, etc.) since it provides necessary information for these tasks. Although there have been advancements in networking technologies, implementing fine-grained traffic measurement in high-speed networks remains challenging for the following two reasons. First, the large number of flows in each node and a high packet arrival rate make it difficult to accurately measure traffic with very limited hardware resources. Second, traffic patterns can vary significantly over time, making it challenging to measure flows using fixed mechanisms.

Sketches are a class of compact data structures that provide summarised information based on a set of counters mapped by hash functions. Sketches are known for their high memory efficiency and ability to provide bounded estimation errors, making them widely used in various data stream processing applications, such as financial data analysis, sensing data collection, and network measurement. In these applications, Sketches are commonly used for tasks such as membership query [1], flow size query [2]–[4], and heavy hitter detection [5].

On the other hand, academic and industry communities have proposed the programmable networking paradigm [6]. In this paradigm, the packet processing behavior s in the data plane can be flexibly programmed, while the network controller in the control plane can dynamically issue and collect runtime configurations and status. The programmable capability of the data and control planes can be leveraged to implement various Sketch-based traffic measurement schemes, providing fine-grained and full visibility to all flows. Therefore, the programmable networking paradigm paves the way for implementing fine-grained and accurate traffic measurements.

Network operators tend to prioritize recent traffic data due to its ability to provide a current situational snapshot and help them anticipate future trends. The sliding window model is a promising approach for measuring recent data streams. By constantly updating the window to include only the most recent data and removing outdated items, this model can provide real-time and accurate results for any queries. However, existing sliding window algorithms mainly suffer from three limitations. First, to enable accurate deletion of outdated data, many existing algorithms require storing large amounts of information, making them difficult to use in memory-constrained scenarios. Second, these algorithms need to maintain complex data structures, leading to high computational complexity. Third, most existing algorithms only support a specific type of query, so they cannot be used in scenarios requiring multiple types of queries.

To handle various query tasks under the sliding window measurement model, X. Guo et al. proposed Sliding Sketch (SS) [7]. Sliding Sketch is a generic sliding window measurement framework that can be applied to most existing sketches, enabling them to answer different types of queries for the current sliding window. Sliding Sketch utilizes an array composed of multiple segments to store measurement information and employs a scanning operation to remove outdated information. Specifically, when a packet arrives, the scanning pointer moves forward k buckets in a circular manner, and the outdated information in the k buckets will be deleted. Sliding Sketch executes the insertion and query strategies based on the strategies of the specific sketch it utilizes. Despite providing a generic framework capable of handling various queries, Sliding Sketch also suffers from the following drawbacks:

*Corresponding author, Email: wangxiong@uestc.edu.cn

1) **Hardware unfriendly**: Sliding Sketch adopts a scanning operation to eliminate outdated items, which involves updating multiple buckets when a new item arrives. However, this operation is computationally intensive and cannot be implemented on today's commodity programmable switches [**?**].

2) **High computational complexity**: Sliding Sketch uses an array as its data structure, which is partitioned into l segments of equal size, with each segment associated with a unique hash function. To perform an insert or query operation, l hash operations are necessary. Typically, to achieve the desired level of accuracy, a large l is needed (e.g., $l = 10$ in [7]). Additionally, the scanning operations are also time-consuming.

In this paper, we develop a traffic measurement framework over the sliding window model called Ring Sketch, which exhibits the following characteristics: 1) It is a generic framework that can be applied to most existing sketches, enabling them to answer different types of traffic information queries of the current window; 2) It requires only a few hash calculations and logic operations for each packet, thereby possessing low computational complexity; 3) It is hardware-friendly, i.e., it can be implemented on commodity hardware programmable switches. The main contributions of this paper are summarized as follows:

(1) To efficiently address various queries for recent traffic information in programmable networks, we propose Ring Sketch, a generic, low-complexity, and hardware-friendly traffic measurement framework over sliding windows.

(2) We propose the estimation strategy for Ring Sketch and theoretically analyze its error bounds on typical sketches.

(3) We apply the Ring Sketch framework on multiple sketches for different types of queries, and we also implement it on a hardware switch with Tofino 1 chip and OVS-DPDK. The experimental results show that Ring Sketch has a throughput over 3x higher than Sliding Sketch, and the measurement accuracy of Ring Sketch is close to or even much better than Sliding Sketch in many cases.

II. Related Work and Background

As network traffic measurement plays an important role in network management, it has received extensive attention from the research community in recent years. Sketches, which are compact data structures capable of summarizing data streams with low overhead, have proven to be efficient for addressing traffic measurement problems in networks with limited resources. In this section, we will introduce the sketches that are used for several typical traffic measurement tasks, as well as their extensions for sliding windows.

A. The sketches

The sketches are mainly used for handling three fundamental data queries: membership query, frequency query, and heavy hitter query. Generally, sketches use an l-hash model, which maps each data item to l distinct counters using hash functions. The membership query is to check whether an item belongs to a set or not. The Bloom filter (BF) [1] is a well-known sketch used for membership queries. To meet the requirements of different applications, many variants of BF are proposed, such as Counting Bloom Filter [8], Dynamic Count Filter [9], Incremental Bloom filter [10], and Shifting Bloom Filter [**?**]. The frequency query returns the frequency of an item in a data stream. In the context of traffic measurement, it is used to report the sizes of flows in a traffic stream. For the frequency queries, the widely used sketches include CM sketch [2], CU sketch [3], and Count sketch [4]. Furthermore, in recent years, there have been several advanced sketches proposed for flow size queries on traffic data. Examples of such sketches include Elastic Sketch [11], Pyramid Sketch [12], and Augmented Sketch [12]. Heavy hitter query tries to find out the items whose frequencies exceed a threshold. The Heavy Keeper [5] and Space-Saving [13] are state-of-the-art sketches for heavy hitter queries. Heavy hitters (large flows) are important for traffic optimization and network security applications. Therefore, several sophisticated sketches, such as HashPipe [**?**], ActiveCM+ [**?**], and MV-Sketch [**?**], are proposed to find large flows.

B. The sliding window sketches

The sliding window sketches only concern the data items in the current window. Therefore, every time a new item arrives, they need to update the data structure and delete the outdated information. To handle different types of queries, many sliding window sketches are proposed [13]–[15]. These sliding window sketches are designed for membership query [14], flow size query [15], and heavy hitter query [13]. However, most of these sliding window sketches are designed to answer one type of query, and they are also memory inefficient.

The authors in [7] propose a generic framework called Sliding Sketch. Sliding sketches can be applied to various sketches to answer different types of queries. However, the Sliding Sketch requires l hashing calculations for both insert and query operations. To achieve the desired level of accuracy, a large l is required. Meanwhile, the scanning operations, which are used to delete the expired data, also incur high computational complexity. In addition, the scanning operations cannot be implemented on today's commodity programmable switches due to the constraints of the PISA programming model.

As shown in Fig. 1, Sliding sketch uses an array with m buckets, which is divided into k equal-sized segments, each associated with a hash function. Every bucket B has two fields B_{new} and B_{old}, which store the information in the current

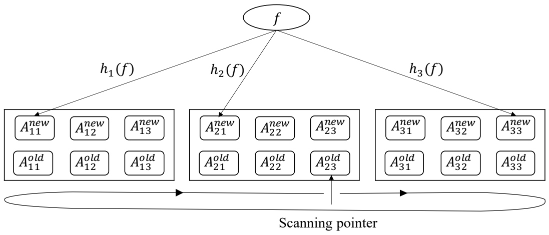


Fig. 1: The data structure of ring sketch.

window and last window, respectively. When a new item arrives, the B_{new} field in the k buckets mapped by the hash functions is updated with the strategy of the specific sketch. The Sliding sketch utilizes a scanning operation to remove outdated information stored in the B_{old} field of the buckets. To achieve this, a scanning pointer is moved across the buckets. When a new item packet arrives, the Sliding sketch moves the scanning pointer forward by $\frac{N}{m}$ buckets, where N is the window size. As the scanning pointer moves to each bucket, it sets the corresponding B_{old} field to 0. For the query operation, the Sliding sketch first get the information stored in the k buckets mapped by the hash functions, and then it returns the query result with the strategy of the specific sketch.

III. Problem Statement

The traffic measurement tasks involve extracting the interested flow statistics from the packet stream passing through each node. A packet stream is an infinite series of packets denoted as $S = \{p_1, p_2, ..., p_i, ...\}$, where each packet is identified by a unique ID (such as five-tuple in the header) and a timestamp indicating its arrival time. There may be multiple packets with the same ID in a packet stream. We define a sub-series of packets that have the same ID i as a flow f_i.

Due to the fact that many network management tasks are primarily concerned with recent traffic statistics within a current time window, this paper considers the traffic measurement problem under the sliding window models. A sliding window is a contiguous segment of packet series, and there are mainly two sliding window models: the time-based sliding window model and the count-based sliding window model.

- **Time-based sliding window model**: A time-based sliding window of size N refers to the subset of packets in S that arrive in the last N time units.
- **Count-based sliding window model**: A count-based sliding window of size N refers to the last arrived N packets in S.

We note that although the two models are different, the methods designed for them are quite similar. Therefore, for the sake of conciseness, we will only consider the count-based sliding window model in the following text. Based on the programming model and resource limitations of programmable network devices, we have aimed to develop a low-complexity and hardware-friendly sliding window traffic measurement framework. Specifically, the framework should support various traffic measurement tasks and can be efficiently implemented on programmable hardware with low resource overhead.

IV. The Ring Sketch framework

A. Data Structure

For each insertion operation, Sliding Sketch uses a pointer to scan the counter array and remove expired data. However, the scanning operation is hard to be implemented on programmable hardware switches due to the constraints of the programming model. To make the framework hardware-friendly, Ring Sketch no longer uses a pointer to actively clean up expired data. Instead, it uses a bit-array B to indicate whether the values of counters are old or new. When performing the insertion operation, the counters identified as new will be updated directly, while the counters identified as old will first be reset and then updated. Moreover, to reduce computational complexity, the Ring Sketch framework uses the same number of hash functions as the specific sketch.

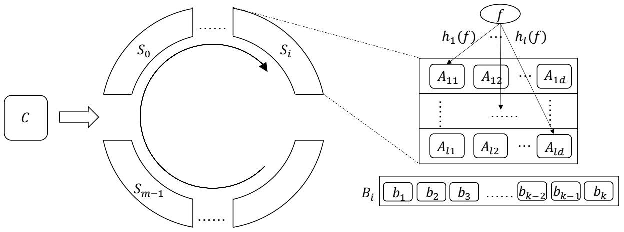


Fig. 2: The data structure of ring sketch.

We divide a sliding window of size N into m sub-windows, with each sub-window having a size of $N_s = N/m$. The data within each sub-window is stored in an independent sketch $S_i (0 \leq i < m)$, and each sketch employs the same set of l hash functions and has K counters. The packets cyclically enter each sketch, forming a ring-like structure, as shown in Fig 2. We use counter C to record the sequence number of the currently arrived packets. This sequence number determines which sketch the packet should be inserted into. Specifically, a packet with sequence number n will be inserted into the i_{th} sketch, where $i = (n/N_s)\%m$.

B. Insertion Operation

When a packet arrives, the sequence number counter C is first incremented by 1. Let the sequence number of the arriving packet be denoted as n. The sketch S_i is updated using the specific l-hashing model employed by it, where i is calculated as $i = (n/N_s)\%m$. For each of the l counters mapped by the l hash functions, it will be directly updated if the corresponding bit in the bit array indicates it as new. Otherwise, this counter will be first reset and then updated, and the corresponding indicator bit in the bit array will also be set as new (e.g., 0). It should be noted that if a counter is marked as new, it stores the information for the current sub-window, while if it is marked as old, it stores the information for a previous sub-window.

When each packet arrives, Ring Sketch not only updates the sketch for the current sub-window but also sets the flag bits of j consecutive counters in the sketch for the last sub-window to old (e.g., set to 1). The j is set to $\frac{K}{N_s}$ so that when the current sub-window ends (after N_s packet arrivals), all counters in the sketch for the previous sub-window will be marked as old, i.e., all bits in the bit-array associated with the sketch for the last sub-window are set to 1.

Example: Fig. 3 illustrates an example of an insertion operation. In this example, we set $N = 36$, $m = 4$, $N_s = 36/4 = 9$, and $K = 3 \times 3 = 9$. When a packet arrives, the value of counter C is first incremented by 1 ($n = 23$). Then the packet information should be inserted into sketch S_i,

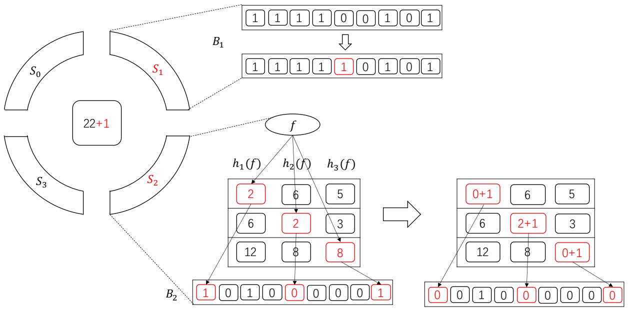
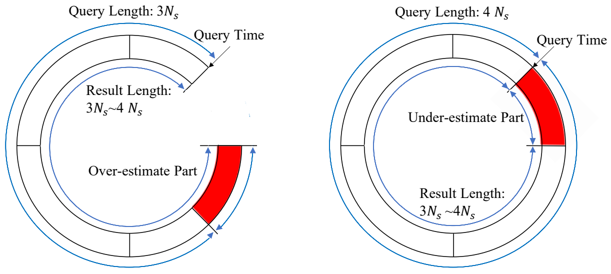


Fig. 3: Example of an insertion operation.



(a) Overestimation strategy.

(b) Underestimation strategy

Fig. 4: Query Strategy

where $i = (n/N_s)\%m = (22/9)\%4 = 2$. Here, we assume the sketches used in the framework are CM sketches for providing flow size queries. The CM Sketches share three hash functions to calculate three indexes as 1, 2, and 3, which correspond to the counters represented by the red squares. In the bit array, the corresponding bits have values 1, 0, and 1, respectively. This means that counters A_{11} and A_{33} store old data, while counter A_{22} stores new data. For counters A_{11} and A_{33}, their values are first reset to 0, then incremented by 1. Afterward, we set the corresponding bits in the bit array to 0. For counter A_{22}, its count value is directly incremented by 1. Finally, j bits in the bit array B_1 are set to 1, where $j = \frac{K}{N_s} = \frac{9}{9} = 1$.

C. Query Operation

When querying flow statistics, we sequentially query each sketch $S_i (0 \le i < m)$, and the query method depends on the specific sketch used. For example, the CM Sketch uses l hash functions to map a flow to l counters of S_i and returns the minimum value of the l counters as the query result. Then, we sum up all the query results returned by the sketches to obtain the final result. At the time of querying, the sketch for the current sub-window only contains partial data of the current sub-window. Therefore, the data range covered by the final result is $(m-1) \times N_s \sim m \times N_s$. For different tasks, we can provide two query strategies: the overestimation strategy and the underestimation strategy.

Overestimation strategy: we allow querying for data within the recent range of $(m-1) \times N_s$. In this case, as shown in Fig. 4(a), the result covers a range that is greater than or equal to the range of the query, resulting in the query results being larger than the actual results.

Underestimation strategy: we allow querying for data within the recent range of $m \times N_s$. In this case, as shown in Fig. 4(b), the result covers a range that is smaller than or equal to the range of the query, resulting in the query results being smaller than the actual results.

D. Analysis

As a generic framework, Ring Sketch can be applied to a variety of traffic measurement tasks such as flow membership query, flow size query, and heavy hitter query. The theoretical error bounds of the Ring Sketch framework on different measurement tasks are related to the specific type of used sketch and query strategy. For different measurement tasks, we provide the error-bound analysis for one type of sketch, and the analysis for other sketches is similar. For the flow size query, we provide the error-bound analysis of the CM Sketch. For the flow membership query, we provide the error-bound analysis of the Bloom Filter. For the heavy hitter query, we provide the error-bound analysis of the Heavy Keeper.

1) Bloom Filter:

Theorem 1. When using an overestimation strategy, the overall probability that our model correctly reports an element that is not in the sub-windows is:

$$Pr = \prod_{i=0}^{m-1} (1 - [1 - (1 - \frac{1}{d})^{n_i})]^l) \tag{1}$$

where n_i is the number of different elements in S_i, d is the number of counters in the Bloom Filter and l is the number of hash functions.

Proof: For a Bloom filter with n distinct elements, we can derive the probability that an element is not in the sub-windows and the model correctly reports it as:

$$Pr = 1 - [1 - (1 - \frac{1}{d})^n]^l \tag{2}$$

Where d is the length of the array and l is the number of hash functions. For a sub-data structure S_i containing $n_i (0 < n_i \le N_s)$ distinct elements, the probability that it correctly reports an element is:

$$Pr_i = 1 - [1 - (1 - \frac{1}{d})^{n_i}]^l \tag{3}$$

Therefore, the overall probability that our model correctly reports an element is:

$$Pr = \prod_{i=0}^{m-1} (1 - [1 - (1 - \frac{1}{d})^{n_i})]^l) \tag{4}$$

In our approach, data within a window is distributed among multiple independent sub-data structures. This reduces the probability of hash collisions compared to the original Bloom filter. As a result, the reporting accuracy of this approach is improved.

2) CM Sketch:

Theorem 2. When using an overestimation strategy, in a CM Sketch with $l \times d$ counters, the query results satisfy:

$$Pr[\sum_{w=0}^{m-1} \hat{a}_{i,w} \le a_i + \varepsilon N] \ge 1 - \delta \tag{5}$$

where a_i is the actual size of flow f_i, $\hat{a}_{w,i}$ is the query size of f_i in S_w, $\varepsilon = \frac{e}{d}$, and $\delta = e^{-l}$.

Proof: For flow f_i, we define the following variables:

- a_i: the actual count of flow f_i.
- $\hat{a}_i$: the queried count of flow f_i.
- $\|a\|_1$: the total count of all flows.
- $I_{i,q,z}$: indicates whether two different flows f_i and f_z yield the same value when substituted into hash function h_q. If $i \neq z$ and $h_q(i) = h_q(z)$, then $I_{i,q,z} = 1$; otherwise, it is 0.

The probability of a hash collision P is given by $P \le \frac{1}{d}$, where $d = \frac{e}{\varepsilon}$, and the full formula is as follows:

$$E(I_{i,q,z}) = Pr[h_q(i) = h_q(z)] \le \frac{1}{range(h_q)} = \frac{\varepsilon}{e} \tag{6}$$

When adopting the over-estimation strategy, it is known that the total count of all flows within the queried range $\|a\|_1$ is less than or equal to N. Based on the pairwise independence of hash functions and the linearity of expectations, the expected value of the sum of flows that collide with flow f_i under the q_{th} hash function denoted as $X_{i,q}$, can be calculated using the following formula:

$$\begin{aligned} E(X_{i,q}) &= E\left(\sum_{z=1}^{n} a_z I_{i,q,z}\right) \\ &\le \sum_{z=1}^{n} a_z E(I_{i,q,z}) \le \frac{\varepsilon}{e}\|a\|_1 \le \frac{\varepsilon}{e}N \end{aligned} \tag{7}$$

According to the Markov expression, the following formula can be derived:

$$\begin{aligned} Pr[\hat{a}_i > a_i + \varepsilon N] &= Pr[\forall_q.count[q, h_q(i)] > a_i + \varepsilon N] \\ &= Pr[\forall_q.a_i + X_{i,q} > a_i + \varepsilon N] \\ &\le Pr[\forall_q.X_{i,q} > eE(X_{i,q})] \\ &< e^{-l} \le \delta \end{aligned} \tag{8}$$

Therefore, the query results satisfy:

$$Pr[\hat{a}_i \le a_i + \varepsilon N] \ge 1 - \delta \tag{9}$$

In any w_{th} sub-window, the total count of all flows within the queried range $\|a_w\|_1$ is less than or equal to N_s, and the query results all satisfy:

$$Pr[\hat{a}_{i,w} \le a_{i,w} + \varepsilon N_s)] \ge 1 - \delta \tag{10}$$

Hence, the query results satisfy:

$$\begin{aligned} Pr[\sum_{w=0}^{m-1} \hat{a}_{i,w} \le a_i + \varepsilon N)] &\ge \prod_{w=0}^{m-1} Pr[\hat{a}_{i,w} \le a_{i,w} + \varepsilon N_s)] \\ &\ge (1-\delta)^m \end{aligned} \tag{11}$$

In fact, the query result in the w_{th} sub-data structure S_w is the minimum value of all hash functions $h_{q,w}(i)\,(1 \le q \le d)$, denoted as $\hat{a}_{i,w}$, and the sum of query values across m windows is $\sum_{w=0}^{m-1} \hat{a}_{i,w}$. For CM Sketch, the sum of values corresponding to each window is $\sum_{w=0}^{m-1} \hat{a}_{i,q,w}$, and the query result is the minimum value among $\sum_{w=0}^{m-1} \hat{a}_{i,q,w}$, denoted as $\hat{a}_i$. Therefore, it can be concluded that:

$$\sum_{w=0}^{m-1} \hat{a}_{i,w} \le \hat{a}_i \tag{12}$$

$$Pr[\sum_{w=0}^{m-1} \hat{a}_{i,w} \le a_i + \varepsilon N] \ge Pr[\hat{a}_i \le a_i + \varepsilon N] \ge 1 - \delta \tag{13}$$

According to the characteristics of CM Sketch, the query result always satisfies: $\sum_{w=0}^{m-1} \hat{a}_{i,w} \ge a_i$.

3) Heavy Keeper:

Theorem 3. When using an underestimation strategy, in a Heavy Keeper with $l \times d$ counters, the query results satisfy:

$$Pr[\hat{a}_i \ge a_i - \varepsilon N] \ge 1 - (\frac{1}{\varepsilon d a_i (b-1)})^l \tag{14}$$

where $\hat{a}_i$ is the query size of f_i, a_i is the actual size of f_i, ε is any small positive number and b is a constant for probability decay.

Proof: According to the [5], for each row of the Heavy Keeper, the query result satisfies:

$$Pr[\hat{a}_{i,q} < a_i - \varepsilon \|a\|_1] \le \frac{1}{\varepsilon d a_i (b-1)} \tag{15}$$

where a_i is the true flow size, $\hat{a}_i$ is the estimated flow size, ε is any small positive number, $\|a\|_1$ is the count of all items, d is the number of buckets per row for hashing, and b is a constant for probability decay. The query result is the maximum value among the mapped buckets of all rows. Therefore, it can be derived that:

$$\begin{aligned} Pr[\hat{a}_i < a_i - \varepsilon \|a\|_1] &= Pr[\forall_q.\hat{a}_{i,q} < a_i - \varepsilon \|a\|_1] \\ &\le (\frac{1}{\varepsilon d a_i (b-1)})^l \end{aligned} \tag{16}$$

For $\|a\|_1 \le N$, it can be derived that:

$$\begin{aligned} Pr[\hat{a}_i \ge a_i - \varepsilon N] &\ge 1 - Pr[\hat{a}_i < a_i - \varepsilon \|a\|_1] \\ &\ge 1 - (\frac{1}{\varepsilon d a_i (b-1)})^l \end{aligned} \tag{17}$$

The query result $\hat{a}_{i,w}$ in the w_{th} sub-structure S_w is the maximum value among the mapped buckets of all hash functions $h_{q,w}(i)\,(1 \le q \le l)$, and the sum of query values for each window is $\sum_{w=0}^{m-1} \hat{a}_{i,w}$. For each window of Heavy Keeper, the sum of the corresponding values after summation is $\sum_{w=0}^{m-1} \hat{a}_{i,q,w}$, and the query result $\hat{a}_i$ is the maximum value among them. Therefore, it can be concluded that:

$$\sum_{w=0}^{m-1} \hat{a}_{i,w} \ge \hat{a}_i \tag{18}$$

$$Pr[\sum_{w=0}^{m-1} \hat{a}_{i,w} \geq a_i - \varepsilon N \geq Pr[\hat{a}_i \geq a_i - \varepsilon N] \geq 1 - (\frac{1}{\varepsilon d a_i (b-1)})^l \tag{19}$$

According to the characteristics of Heavy Keeper, the query result always satisfies: $\sum_{w=0}^{m-1} \hat{a}_{i,w} \leq a_i$.

E. Improve Memory Efficiency

To support sliding window based measurements, Ring Sketch actually divides the measurement window into n sub-windows, and each sub-window uses an independent Sketch. The measurement accuracy of Ring Sketch increases with n, but the total number of counters required also increases. To implement more counters under limited memory space, small counters (i.e., counters with fewer bits) must be used. However, using small counters (e.g., 8 bits) may cause overflow in some counters recording large flows. To address this issue, several schemes such as Counter tree [16] and SA Counter [17] are proposed. Unfortunately, these solutions require many memory accesses or logic judgments, which cannot be implemented on hardware switches. Therefore, we use a lightweight scheme, which requires only one logic judgment for each update operation. Suppose a counter has b bits. The key idea of this scheme is to record small flows and sample the statistics of large flows. Specifically, if the value of a counter is less than or equal to $2^{\frac{b}{2}} - 1$, the counter is updated normally, otherwise the counter is updated with probability $2^{-\frac{b}{2}}$. Therefore, the maximum value of a counter is:

$$(2^b - 2^{\frac{b}{2}}) \times 2^{\frac{b}{2}} + 2^{\frac{b}{2}} - 1 = 2^{\frac{3b}{2}} - 2^b + 2^{\frac{b}{2}-1} \tag{20}$$

Since most flows in the network are small flows, this method can ensure high measurement accuracy for small flows. At the same time, sampling measurement for large flows has less impact on measurement accuracy.

F. Implementation

We have implemented the Ring Sketch framework on both OVS-DPDK software switch and a hardware switch with a Tofino programmable chip, and the source codes are released at GitHub [18]. As the implementation on the OVS-DPDK software switch is straightforward, we will only introduce our implementation on the hardware switch with Tofino chips.

According to the programming model of PISA switches, we deploy the data structures and operations of Ring Sketch into different stages of the switch pipeline. The framework of the hardware implementation is shown in Fig. 5. As shown in Fig. 5, the hashing and the packet sequence number counter are placed in a stage preceding the stages where the sketch data structure and operations are located. Since all the sketches for different sub-windows share the same set of hash functions, the hashing results will be stored in the PHV of each packet for later use by the sketches deployed in subsequent stages. The arrays in sketches can be implemented using registers in SRAM, and multiple sketches can be placed in the same stage because there is no dependency between sketches. To save memory resources and for ease of implementation, we use the highest bit of each counter to indicate whether the counter is new or old. However, we can also use a separate bit array to mark the counters of each sketch. When a packet arrives, the sketch for the current sub-window is updated, and the highest bits of the counters in a column of the sketch for the last sub-window are set to 1. The query operation of Ring Sketch can also be easily implemented on hardware switches. Specifically, we can write the query results in the header of a query packet, and the result collector only needs to perform some simple operations (e.g., summation or finding the minimum value) to obtain the queried result.

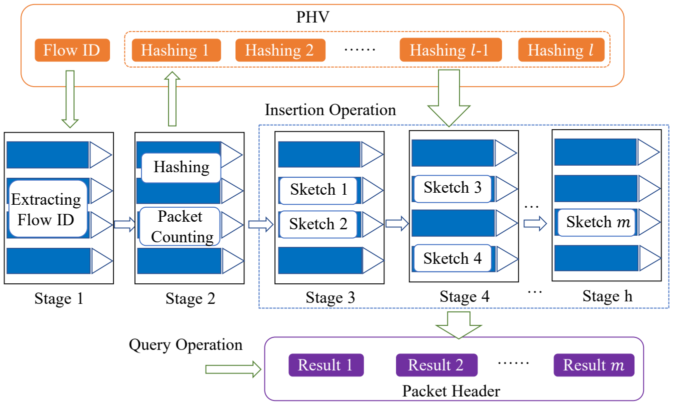


Fig. 5: Implementation of P4 platform.

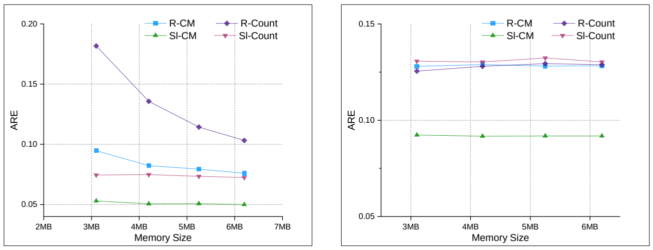
Table 1 lists the resource usage of the Ring Sketch framework when using the CM Sketch to handle flow size queries. It is notable that the number of registers allocated to each CM sketch is adjustable. In our experiments, the SRAM usage varies from 100KB to 1MB, which takes up only a small fraction of the total SRAM. The SALUs are mainly used by insert and query operations in our implementation. We can find that the Ring Sketch framework can not only be implemented on hardware switches, but also its resource utilization rate is low.

V. Performance Evaluation

In this paper, We mainly compare Ring Sketch with the state-of-the-art Sliding Sketch. We apply Sliding Sketch and Ring Sketch to five different sketches for completing three typical traffic measurement tasks: flow size query, flow membership query, and heavy hitter detection. For simplicity,

TABLE I: Resource Usage of P4

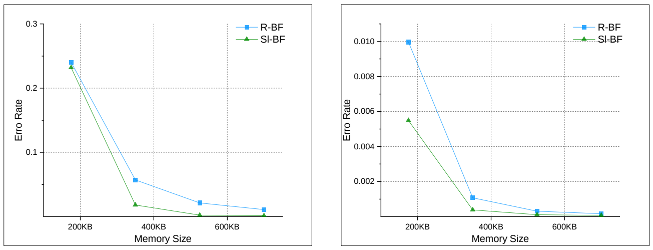
Resource	*Quantity*	*Percentage*
Match Crossbar(1536)	82	5.3%
SRAM(960)	59-143	6.1% – 14.9%
TCAM(288)	0	0%
VLIM Actions(384)	11	2.9%
Hash Bits(4992)	510	10.2%
Stateful ALUs(48)	17	35.4%
Packet Header Vector(2048)	760	37.1%



(a) MAWI dataset

(b) Synthetic dataset

Fig. 6: The ARE of flow size query.



(a) MAWI dataset

(b) Synthesized dataset

Fig. 7: The ER of flow membership query.

we use the terms 'R-sketch name' (e.g., R-CM) and 'Sl-sketch name' (e.g., Sl-CM) to refer to the specific algorithm used under the Ring Sketch and Sliding Sketch frameworks, respectively.

A. Experimental Setup

Performance Metrics: We use the following three performance metrics to evaluate the performance of Ring Sketch on the three typical traffic measurement tasks:

1) Average Relative Error (ARE): ARE is used for flow size query task, and $ARE = \frac{1}{|F|}\sum_{f_i \in F}\frac{||f_i|-|\hat{f}_i||}{|f_i|}$, where F represents the set of flows, $|f_i|$ denotes the true size of flow f_i, and $|\hat{f}_i|$ represents the reported size of flow f_i.

2) Error Rate (ER): ER is used for flow membership query task, and $ER = \frac{F_p+F_n}{|Q|}$, where F_p, F_n, and Q are false positives, false negatives, and the set of queries, respectively.

3) Precision Rate (PR): PR is used for heavy hitter detection tasks, and it is the number of large flows reported to the actual number of large flows.

4) Throughput: The throughput is used to evaluate the computation complexity of traffic measurement algorithms. It is the number of packets processed by an algorithm per second.

Dataset: We use two flow traces in the experiments:

1) MAWI dataset: The daily trace data from the WIDE Project for the year 2022 [19].

2) Synthesized dataset: Randomly generated trace using a Zipf distribution with a skew value of 1.5.

In both datasets, we identify a flow using the five-tuple (source IP address, destination IP address, source port number, destination port number, and protocol type).

Parameters: The number of hash functions l used by Ring Sketch and Sliding Sketch is 3 and 10, respectively. The number of sub-windows in Ring Sketch is 8. We set the window size N to 100k and evaluate the measurement results after every 1k packet arrives. The overestimate strategy is used in all experiments.

B. Experimental Results

(1) Evaluation of flow size query

We apply CM and Count sketches in Ring Sketch and Sliding Sketch frameworks for the flow size query. Fig. 6 shows the AREs of the flow size queries of the two sketches under the Ring Sketch and Sliding Sketch frameworks. To facilitate implementation on hardware switches, Ring Sketch does not update expired information as accurately as Sliding Sketch. This results in Ring Sketch's overestimation query strategy retaining more expired information. Consequently, Ring Sketch exhibits larger measurement errors compared to Sliding Sketch. However, for the flow size measurement task, both Ring Sketch and Sliding Sketch have small measurement errors, and their performance gap is also small. For example, when the memory size is set to 3MB, the ARE of R-CM and Sl-CM on the MAWI dataset are 0.094 and 0.053, respectively. Under the two sliding window frameworks, the CM Sketch can achieve better performance than the Count sketch in most cases. Moreover, all the algorithms share similar performance trends on the two datasets.

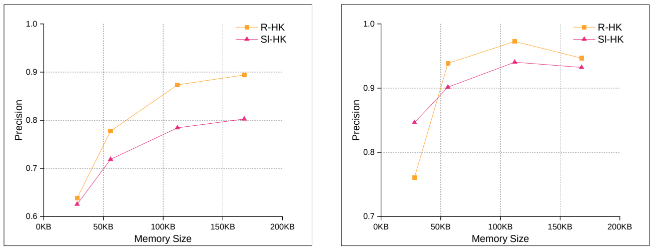
(2) Evaluation of flow membership query For the flow membership query, we use Bloom Filter in both Ring Sketch and Sliding Sketch frameworks. The ERs of R-BF and Sl-BF under MAWI and Synthesized datasets are shown in Fig. 7. The experimental results show that as the memory size increases, the ERs of R-BF and Sl-BF decrease rapidly. Similar to the flow size measurement task, R-BF has a larger ER than Sl-BF. However, for the membership query task, Ring Sketch can still achieve very small measurement errors. For example, when the memory capacity exceeds 350KB, the ER of Ring Sketch is below 5%.

(3) Evaluation of heavy hitter query

For the heavy hitter query, we use HeavyKeeper (HK) [5] in both Ring Sketch and Sliding Sketch. The flows with more than 100 packet are considered heavy hitters. Fig. 8 shows that the PRs of both R-HK and Sl-HK under MAWI and Synthesized datasets. It is interesting that in this task, R-HK performs better than Sl-HK in most cases. The reason is that in this task, small measurement errors have little impact on the accuracy of heavy hitter identification. We can observe that the PRs of R-HK are higher than 90% when the memory size exceeds 160KB.

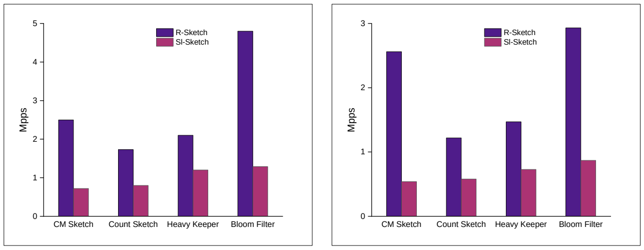
(4) Evaluation of throughput on software implementation

The Ring Sketch can process packets at a line rate on hardware switches. To evaluate its throughput on software implementation, we conduct experiments on a server with 2 CPUs (48 cores, Intel Xeon Platinum 8160H@2.1GHZ) and 256GB DRAM. From Fig. 9(a), we find that the throughput



(a) MAWI data set
(b) Synthesized dataset

Fig. 8: The PR of heavy hitter query.



(a) Throughput of algorithms
(b) Throughput on OVS-DPDK

Fig. 9: Throughput

of Ring Sketch is more than 3x higher than Sliding Sketch. We integrate our Ring Sketch into OVS 2.16.0 with DPDK 20.11.6. In this experiment, we only use one CPU core to process packets. From Fig. 9(b), we also can observe that the throughput of Ring Sketch on OVS-DPDK is at least 2x higher than Sliding Sketch. This is mainly because Ring Sketch performs much fewer hash calculations than Sliding Sketch.

VI. Conclusion

In this paper, we designed Ring Sketch, a generic, low-complexity, and hardware-friendly traffic measurement framework over the sliding windows. Ring Sketch can provide typical flow statistics widely used in network management tasks by using different sketches, and can be implemented on commodity programmable hardware switches. For the Ring Sketch framework, we proposed the estimation strategy and theoretically analyzed its error bounds for different sketches. To evaluate its feasibility and performance, we implemented Ring Sketch on OVS-DPDK and a programmable hardware switch. The evaluation results show that Ring Sketch is fast and can achieve high measurement accuracy in typical measurement tasks.

References

[1] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[2] 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.

[3] C. Estan and G. Varghese, “New directions in traffic measurement and accounting,” in *ACM SIGCOMM 2002*, pp. 323–336, 2002.

[4] M. Charikar, K. Chen, and M. Farach-Colton, “Finding frequent items in data streams,” *Theoretical Computer Science*, vol. 312, no. 1, pp. 3–15, 2004.

[5] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, “Heavykeeper: an accurate algorithm for finding top-k elephant flows,” *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[6] 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.

[7] X. Gou, L. He, Y. Zhang, K. Wang, X. Liu, T. Yang, Y. Wang, and B. Cui, “Sliding sketches: A framework using time zones for data stream processing in sliding windows,” in *ACM SIGKDD 2020*, 2020.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary cache: a scalable wide-area web cache sharing protocol,” *IEEE/ACM transactions on networking*, vol. 8, no. 3, pp. 281–293, 2000.

[9] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J. L. Larriba-Pey, “Dynamic count filters,” *Acm Sigmod Record*, vol. 35, no. 1, pp. 26–32, 2006.

[10] F. Hao, M. Kodialam, and T. Lakshman, “Incremental bloom filters,” in *IEEE INFOCOM 2008*, pp. 1067–1075, 2008.

[11] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, “Elastic sketch: Adaptive and fast network-wide measurements,” in *ACM SIGCOMM 2018*, pp. 561–575, 2018.

[12] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, “Pyramid sketch: A sketch framework for frequency estimation of data streams,” *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1442–1453, 2017.

[13] A. Metwally, D. Agrawal, and A. El Abbadi, “Efficient computation of frequent and top-k elements in data streams,” in *ICDT 2005*, pp. 398–412, Springer, 2005.

[14] A. Arasu and G. S. Manku, “Approximate counts and quantiles over sliding windows,” in *ACM SIGMOD-SIGACT-SIGART 2004*, pp. 286–296, 2004.

[15] N. Rivetti, Y. Busnel, and A. Mostefaoui, “Efficiently summarizing data streams over sliding windows,” in *IEEE NCA 2015*, pp. 151–158, 2015.

[16] M. Chen, S. Chen, and Z. Cai, “Counter tree: A scalable counter architecture for per-flow traffic measurement,” *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1249–1262, 2016.

[17] T. Yang, J. Xu, X. Liu, P. Liu, L. Wang, J. Bi, and X. Li, “A generic technique for sketches to adapt to different counting ranges,” in *IEEE INFOCOM 2019*, 2019.

[18] “Ring sketch implementation and technical report.” https://github.com/lgeon/RingSketch.

[19] “Mawi 2022 daily trace.” http://mawi.wide.ad.jp/mawi/samplepoint-F/2022/.