Universal Online Sketch for Tracking Heavy Hitters and Estimating Moments of Data Streams

Qingjun Xiao
School of Cyber Science & Engineering
Southeast University
Nanjing, China
csqjxiao@seu.edu.cn

Zhiying Tang
School of Computer Science & Engineering
Southeast University
Nanjing, China
tangzhiying@xiaomi.com

Shigang Chen
Department of CISE
Florida University
Gainesville, FL, USA
sgchen@cise.ufl.edu

***Abstract*—Traffic measurement is key to many network management tasks such as performance monitoring and cyber-security. Its aim is to inspect the packet stream passing through a network device, classify them into flows according to the header fields, and obtain statistics about the flows. For processing big streaming data in size-limited SRAM of line cards, many space-sublinear algorithms have been proposed, such as CountMin and CountSketch. However, most of them are designed for specific measurement tasks. Implementing multiple independent sketches places burden for online operations of a network device. It is highly desired to design a universal sketch that not only tracks individual large flows (called heavy hitters) but also reports overall traffic distribution statistics (called moments). The prior work UnivMon successfully tackled this ambitious quest. However, it incurs large and variable per-packet processing overhead, which may result in a significant throughput bottleneck in high-rate packet streaming, given that each packet requires 65 hashes and 64 memory accesses on average and many times of that in the worst case. To address this performance issue, we need to fundamentally redesign the solution architecture from hierarchical sampling to new progressive sampling and from CountSketch to new ActiveCM+, which ensure that per-packet overhead is a small constant (4 hash and 4 memory accesses) in the worst case, making it much more suitable for online operations, especially for pipeline implementation. The new design also makes effort to reduce memory footprint or equivalently improve measurement accuracy under the same memory. Our experiments show that our solution incurs just one sixteenth per-packet overhead of UnivMon, while improving measurement accuracy by three times under the same memory.**

I. Introduction

The line rate in modern high-speed networks has reached hundreds of Gbps or multiple Tbps. Meanwhile it becomes a common practice to let switches/routers inspect their packet streams for network performance monitoring, event detection, or threat identification against worm activities, DDOS attacks, scanning, etc. Network operators often need to collect a variety of different statistics and measurements, including per-flow sizes [1], [2], heavy hitters [3]–[8], and aggregated information about flow distributions (called *moments* such as entropy and variance) [9], [10], which may be used to detect anomalies in overall traffic patterns. Most existing algorithms are designed for specific measurement tasks. Moreover, the aforementioned measurements may need to be conducted under various different flow definitions [6], [7], which will cause the number of measurement tasks to multiply.

Implementing a large number of tasks separately occupies significant on-chip memory/computing resources that are shared by other network functions. The prior research addresses the problem in two directions. One is taken by OpenSketch [7], which allows different tasks to share a common set of primitive implementation components. This paper is interested in the other direction taken by UnivMon [6], which is to design a universal sketch that can estimate per-flow sizes, identify heavy hitters and measure various moments at once.

UnivMon opens the door for a promising but challenging research direction. But its hierarchical sampling design incurs large and extraordinarily-varying overhead per packet, requiring 65 hashes and 64 memory accesses on average, with the worst-case numbers being 257 hashes and 256 memory accesses. The benefit of a universal sketch over multiple sketches — which each requires a few hashes and memory accesses per packet — is significantly weakened with such a high per-packet processing overhead. Therefore, it is practically important to investigate new, efficient universal sketches that not only perform multiple tasks but do so at a cost similar to some of the single-task sketches.

In this paper, we present a Light-weight Universal Sketch (LUS), which is light-weight in terms of processing overhead and space requirement. It has two major technical innovations. One is called progressive sampling that records each packet exactly once in a single sketch, whereas hierarchical sampling in UnivMon records each packet in a variable number of sketches (four on average), where both LUS and UnivMon employ multiple sketches for moment measurements. We give the algorithms that identify heavy hitters and compute moments based on the data recorded from progressive sampling. Single sketch update per packet makes our design more suitable for pipeline implementation on the data plane of a high-speed switch. The other innovation is a new sketch called ActiveCM+ which takes less memory, fewer hashes and fewer memory accesses than the CountSketch [4] used by UnivMon. Combining the above two techniques, our LUS incurs 5 hashes (which can be reduced to two) and 3.18 memory accesses per packet on average. Our trace-based experiments also reveal that the measurement accuracy of LUS is about 3 times better than that of UnivMon under the same amount of memory for two reasons. First, under the same memory, ActiveCM+

has more counters than CountSketch, which helps improve accuracy. Second, progressive sampling records each packet just once, which helps reduce noise in the sketches.

II. Problem Definition

In this section, we formulate our network flow measurement problem, and describe the key performance metrics.

A. *Definition of Flow Statistics*

From a stream of IP packets, we can extract a sequence of tuples $\langle f_1, c_1\rangle$, ..., $\langle f_t, c_t\rangle$, ..., where $\langle f_t, c_t\rangle$ is the pair of flow ID and packet size extracted from the t-th IP packet, f_t is the flow ID with $1 \le f_t \le \mathcal{F}$, and c_t is the size of the packet. Flow ID may be defined as source IP/Port, destination IP/Port, or the tuple $\langle$srcIP, srcPort, dstIP, dstPort, Protocol$\rangle$, depending on different monitoring applications. In this paper, we treat "stream" and "flow" as two different concepts: A stream is an arbitrary interleaving of IP packets belonging to a number of flows that are concurrently transmitted.

Per-Flow Size. Let n_f be the size of a flow f. When a packet $\langle f_t, c_t\rangle$ arrives, we increase the size of flow f_t by one (or by the size of the packet c_t). Hence, n_f is the number of packets (or bytes) that belongs to the flow f. Our algorithm to present later cab support the counting of the number of packets. It can also be extended easily to count the number of bytes. Let n be the combined sizes of all flows. Clearly, $n = \sum_{1\le i\le \mathcal{F}} n_f$.

Flow Moment. The g-moment of the stream is the functional sum of the flow size n_f for all flows $f \in [1, \mathcal{F}]$:

$$L_g = \sum_{1\le f\le \mathcal{F}} g(n_f), \tag{1}$$

where $g(x)$ is a monotonic function bounded by x^2. Typical definitions of the function g are given as follows.

- If $g(x) = x^0 = 1$, then L_g is called the zeroth-order moment, and it is equal to the number of flows $\mathcal{F}$.
- If $g(x) = x$, then L_g is called the first-order moment, and it is equal to the combined size of all flows: $L_g = m$.
- If $g(x) = x \log x$, then L_g is the entropy of the sizes of all flows, which can be used to measure the diversity of the size distribution of all flows.
- If $g(x) = x^2$, then L_g is the second-order moment of the sizes of all flows, which can be used to calculate the variance of the size distribution of all flows.

Flow moment can quantify the overall condition of the network traffic. We can measure the flow moment L_g at regular time intervals and obtain a time series about this metric. Then, by testing whether its short-term change exceeds a threshold, we may detect the anomalous events in flow size distribution.

Heavy Hitters. Intuitively, a heavy hitter is a flow whose size contribute a lot to the flow moment L_g. More formally, a heavy hitter is any flow f whose size is larger than a threshold $\alpha\, L_g$:

$$H_g = \{f \mid g(n_f) \ge \alpha\, L_g\}. \tag{2}$$

where α is a pre-defined small ratio between zero and one, and H_g is the set of all heavy hitters. When $g(x) = x$, we call H_g the first-order heavy hitters. When $g(x) = x^2$, we call H_g the second-order heavy hitters or $L2$ heavy hitters.

B. *Performance Metrics*

For many applications, it is unnecessary to determine the precise values of heavy hitters and flow moments. It suffices to provide only their approximated values with bounded error.

Heavy Hitter Estimation. Let $\hat{H}_g$ be the estimation for the set of heavy hitters H_g in (2). The probability for the identified heavy hitters $\hat{H}_g$ to include all the actual heavy hitters H_g must be greater than $1-\delta$, where δ is called failure probability.

$$Pr\{H_g \subseteq \hat{H}_g\} \ge 1 - \delta \tag{3}$$

For each identified heavy hitter $f \in \hat{H}_g$, we must generate an estimation $\hat{n_f}$ for its flow size n_f, and guarantee the relative estimation error $\frac{g(\hat{n_f})-g(n_f)}{g(n_f)}$ is bounded by a threshold $\pm\epsilon$ at a probability of at least $1-\delta$. More formally, we must ensure

$$\forall f \in \hat{H}_g, \quad Pr\big\{|g(\hat{n_f}) - g(n_f)| \le \epsilon\, g(n_f)\big\} \ge 1 - \delta. \tag{4}$$

Moment Estimation. For the moment L_g defined in (1), let $\hat{L_g}$ be its estimated value. Its relative estimation error $\frac{\hat{L_g}-L_g}{L_g}$ is bounded by the threshold $\pm\gamma$ at a probability at least $1-\eta$.

$$Pr\big\{|\hat{L_g} - L_g| \le \gamma L_g\big\} \ge 1 - \eta \tag{5}$$

For an efficient solution of flow-level measurement, two other performance metrics exist besides estimation accuracy.

Memory Overhead. A sketch, as deployed in a router/switch, depends on the on-chip memory of line card to keep up with line speed. However, on-chip memory is size-limited and has to be shared with other network functions. For a measurement function, there is a tradeoff between the allocated memory and the accuracy: The more memory is given, the better accuracy it will provide. Hence, the memory cost to satisfy pre-defined measurement error bound is an important performance metric.

Packet Processing Cost. The time cost of processing a packet is composed of two parts: hash computations and memory accesses. The results of hash computations are typically used for two purposes: perform packet sampling or locate a random memory unit to access. Since sampling is often performed only once for a packet, updating multiple memory units in a sketch dominates the packet processing cost, which involves multiple hash computations and memory reads/writes. Clearly, updating too many memory units can easily consume up the tight time budget per packet, especially when the line speed of a network switch/router evolves to hundreds of Gbps or multiple Tbps.

C. *Applications to Network Monitoring*

As discussed earlier, a g-moment of the packets passing through a switch can be estimated for an arbitrary g function upper bounded by $g(x) = x^2$. Next, we show a range of measurement tasks that can be undertaken by the universal sketch.

- Heavy hitter detection is to identify the heavy flows that occupy more than a fraction α of the total network traffic as in (2). Configuring a function $g(x) = x$, the universal sketch will output a list of heavy flows whose packet frequencies are larger than α fraction of the total number of packets.

- DDoS detection is to detect whether a host is under distributed denial of service attack [7]. We can do so by checking if more than k unique flows from different sources are sending packets to the host. To implement this function, we can deploy a universal sketch to monitor the packets towards this host and set the g function to $g(x) = x^0$.
- Entropy estimation: Suppose we want to monitor the traffic towards a sensitive destination IP. We classify the packets into flows by the source IPs, and we define the flow entropy as $E = -\sum_{1\leq f\leq \mathcal{F}} \frac{n_f}{n} \log \frac{n_f}{n}$. A heavy change of this metric may indicate the sensitive destination IP is under attack. Since n can be easily recorded by a single counter, the core task is to estimate the moment $\sum_{1\leq f\leq \mathcal{F}} n_f \log n_f$.
- Global iceberg detection: Consider a network consist of N distributed nodes (e.g., switches). Suppose a flow is a per-destination flow whose packets have a common destination IP. The packets of a flow f may span multiple monitoring nodes. We want to detect the presence of the flows whose total frequency exceed a certain threhold. Suppose each node independently monitors its network traffic using a universal sketch. Later the sketches at different nodes can be fetched to a central server to perform the offline analysis. Our universal sketch can easily support this global merging operation.

III. Prior Art and Motivation

We first introduce the most related work, explain a serious performance issue, and then provide motivation for our work.

A. Per-flow Size Sketches

Many compact data structures have been proposed to estimate per-flow sizes, such as CountMin (CM) [3], Conservative Update (CU) [11], Count Sketch (CS) [4] and Virtual Active Counters (VAC) [2]. They do not keep the flow IDs. Given a flow ID, they can provide an estimate of the flow's size.

CM [3] maintains a two-dimensional array of counters with d rows, where d is typically set to 4. For each packet, it hashes the flow ID (from the packet header) to a counter in each of the d rows, and increases the counter by one. Its per-packet overhead is thus d hashes and $2d$ memory accesses, one read and one write for each of the d hashed counters. CU [11] reads all d hashed counters but only writes back the smallest one(s). To query the size of a flow, CM/CU reports the smallest value of the d hashed counters as the estimate.

Instead of increasing the d hashed counters by one, CS [4] differs from CM/CU by adding a +1/-1 hash value to each hashed counter, where a +1/-1 hash function will pseudo-randomly map a flow ID to +1 or -1. The per-packet overhead of CS is $2d$ hashes and $2d$ memory accesses. To query the size of a flow, CS reports the medium or mean of the d hashed counters, where d is set to 8 for good accuracy. Each counter in CM, CU or CS is often 32 bits long for a range up to $2^{32}-1$.

VAC [2] uses virtual active counters to improve memory efficiency. However, its query needs to operate on hundreds of counters (instead of 4 or 8), which makes it less suitable for online queries.

For all the above sketches, measurement accuracy depends on two factors. One is the total number of packets from all flows that are recorded. Each flow shares its hashed counters with other flows. As the number of packets from other flows increases, the error in the shared counters will increase. The second factor is the memory allocated to the sketch. When the memory increases, the number of counters increases and the error deceases due to less sharing.

B. Universal Sketch

It will be too expensive to implement separate modules for different measurement tasks, such as per-flow sizes, heavy hitters, and various moments. OpenSketch [7] provides a framework to share common components among different measurement tasks, which are still treated individually.

Universal sketch that can handle different measurement tasks at once is an under-investigated subject. UnivMon [6] is the first and arguably only true universal sketch. It uses a series of $\ell + 1$ Counter Sketches (CS) [4], denoted as C_i, $0 \leq i \leq \ell$, where ℓ is set to 15 in the original paper. It performs hierarchical sampling on the flows, with probability $\frac{1}{2^i}$ for C_i, $0 \leq i \leq \ell$. That is, all flows are sampled for C_0 and their packets are recorded in C_0. Half of the flows in C_0 are sampled to be also recorded in C_1. Similarly, half of the flows in C_1 are sampled to be again recorded in C_2, and so on. Without going to details, such recording in the $\ell + 1$ sketches will allow us to estimate all moments defined in Section II.

C. Performance Issue

UnivMon has a serious performance issue due to its high per-packet processing overhead. Recall that the overhead of CS is $2d$ hashes and $2d$ memory accesses. Each packet is expected to be recorded in multiple CS sketches. That number is $1 \times 1 + \frac{1}{2} \times 2 + \frac{1}{4} \times 3 + \frac{1}{8} \times 4 \ldots = 4$ on average. But in the worst case, a packet will be sampled for recording in all $\ell + 1$ sketches. Therefore, the average per-packet overhead is $8d + 1$ hashes and $8d$ memory accesses, while the worst-case overhead is $2d(\ell + 1) + 1$ hashes and $2d(\ell + 1)$ memory accesses, where we need one hash for sampling.

If we use the typical values of $d = 8$ and $\ell = 15$ for CS and UnivMon respectively, the overhead of UnivMon is 65 hashes and 64 memory accesses per packet on average, while the worst-case numbers are 257 hashes and 256 memory accesses! Although the worst-case numbers rarely occur, they do suggest extraordinarily varying per-packet overhead, which is not welcome in processing a high-rate packet stream, particularly for pipeline implementation.

D. Our Goals

We have three goals in this paper. The first goal is to address the performance issue in designing a universal sketch. We will decrease the average per-packet overhead to 5 hashes and 3.18 memory accesses and the worst-case overhead to 5 hashes and 8 memory accesses. In order to achieve this goal, we have to abandon hierarchical sampling and introduce new progressive sampling (with its algorithms for heavy hitters and moments), which records each packet in exactly one of $\ell + 1$ sketches. We also abandon CS and introduce a new ActiveCM+ sketch, which is more compact and requires much fewer writes. Furthermore, we discuss how to reduce 5 hashes of our solution to just two per packet, one for sampling and one for ActiveCM+.

Our second goal is to improve measurement accuracy. Instead of recording each packet in four CS sketches on average by UnivMon, we record each packet in one ActiveCM+ sketch, which means each of our $\ell + 1$ sketches records one fourth of the packets that a sketch in UnivMon records on average. As we explained earlier, with fewer packets recorded, we reduce error in the counters and thus improve measurement accuracy.

Our third goal is to reduce memory use. We reduce each counter in ActiveCM+ to 16 bits, yet with a larger range than a 32-bit counter in CS. Therefore, we can reduce the memory of each sketch by half. Or if we use the same amount of memory, we can double the number of counters, which again help improve measurement accuracy.

IV. ActiveCM+ Sketch

We present a new ActiveCM+ sketch, which will be used by our progressive universal sketch in the next section. We first introduce the design of a compact data structure called ActiveCM. Next, we describe its probabilistic recording, query operation and hash acceleration. By adding a heavy-hitter filter, our final sketch is called ActiveCM+.

A. ActiveCM Data Structures

ActiveCM (Active Count-Min) adopts a compact data structure with two techniques: counter sharing and counter compression, which allow an arbitrary number of flows to share limited on-chip memory pre-allocated for traffic measurement.

Counter sharing: As shown in Figure 1, a single physical counter array PC_* is constructed from the allocated memory. Let m be the number of counters in PC_*. These counters are shared by all flows no matter how many they are.

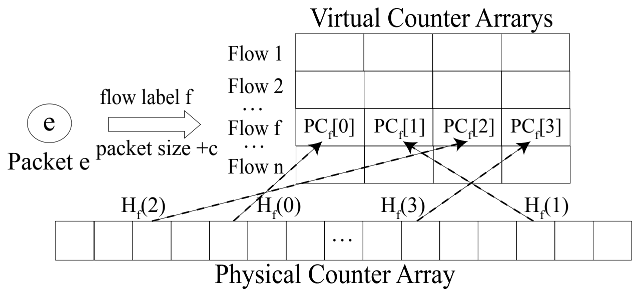


Fig. 1. Counter Sharing in ActiveCM

Each flow is assigned a virtual counter array, shown as a small row of d counters in Figure 1, which is typically set to 4 (as our experimental results suggest). Let PC_f be the virtual counter array for flow f. The ith counter in PC_f is denoted as $PC_f[i]$, $0 \le i < d$. We construct this virtual counter by randomly choosing a physical counter from the array PC_*:

$$PC_f[i] = PC_*[H_i(f)], \qquad 0 \le i < d, \tag{6}$$

where H_i is a pseudorandom hash function, which can be implemented from a master hash function H:

$$H_f(i) = H(f \oplus R[i]) \mod m, \qquad 0 \le i < d \tag{7}$$

where $\oplus$ is the XOR operator and R is an array of d randomly selected constants.

Counter compression: We use a variant design of active counters [12] in PC_*. Each counter $PC_*[i]$, $0 \le i < m$, is 16 bits long, which are split into two parts: (1) $PC_*[i].\alpha$ contains the first 5 bits, which is an exponent, and (2) $PC_*[i].\beta$ contains the remaining 11 bits, which serves as a counter in units of $2^{PC_*[i].\alpha}$ — namely, $PC_*[i].\beta$ will be increased by one after an expected number of $2^{PC_*[i].\alpha}$ packets are received. Moreover, the counter $PC_*[i].\beta$ is in fact 12 bits long, with an implicit leftmost bit of 1. The value of $PC_*[i]$, written in the form of a function V, is

$$V(PC_*[i]) = PC_*[i].\beta \times 2^{PC_*[i].\alpha} + 2^{11+PC_*[i].\alpha} - 2^{11}, \tag{8}$$

where multiplying $2^{PC_*[i].\alpha}$ can be implemented by shifting $PC_*[i].\beta$ to the left for $PC_*[i].\alpha$ bits. The term $2^{11+PC_*[i].\alpha}$ accounts for the leftmost implicit bit of 1. At the very beginning before any packet is counted, both $PC_*[i].\alpha$ and $PC_*[i].\beta$ are zeros, the implicit leftmost bit would cause an initial value of 2^{11}, which needs to be corrected, as shown by the last term in the formula. The purpose of the above design will become clear when we describe its operations.

Consider a simplified example where where $PC_*[i].\alpha = 10001_b = 17$ and $PC_*[i].\beta = 0...011_b = 3$. Then, the value of $PC_*[i]$ is $3 \times 2^{17} + 2^{11+17} - 2^{11} = 134608896$.

The maximum value of $PC_*[i].\alpha$ is $2^5 - 1 = 31$. The maximum value of the counter $PC_*[i].\beta$ with an leftmost implicit bit is $2^{12} - 1 = 4095$. Hence, the maximum counting range is $2^{31} \times 4095 + 2^{41} - 2^{11}$, far greater than a 32-bit counter. Because the counter $PC_*[i].\beta$ increases in units of $2^{PC_*[i].\alpha}$, the error may be as large as $2^{PC_*[i].\alpha}$. But the relative error, $\frac{2^{PC_*[i].\alpha}}{V(PC_*[i])}$, is related to the length of $PC_*[i].\beta$ and bounded by $\frac{1}{2^{11}}$, which is very small.

B. ActiveCM Operations

ActiveCM has two main operations: recording arrival packets and querying a flow's size. For simplicity, we assume that ActiveCM increments each virtual counter $PC_f[i]$ by one, when a packet of a flow f arrives, with the purpose to count the number of packets. In fact, this algorithm can be easily extended to count the number of bytes for each flow.

Probabilistic Recording: At the beginning of each measurement period, all active counters in PC_* are reset to zeros. When a packet of flow f arrives, we record it in all d counters in PC_f. Consider an arbitrary counter $PC_f[i]$, $0 \le i < d$, which is in fact $PC_*[H_i(f)]$. We increase it by one with a probability of $\frac{1}{2^{PC_f[i].\alpha}}$; the increment will happen after an expected number of $2^{PC_f[i].\alpha}$ packets arrive. Clearly, we need to read $PC_f[i]$ in order to compute the probability, whereas the chance of writing back after increment is $\frac{1}{2^{PC_f[i].\alpha}}$, which decreases rapidly as $PC_f[i].\alpha$ increases. As the number of packets increases, the average number of writes per packet approaches toward zero.

When we increase $PC_f[i].\beta$ by one, if it overflows, it will become zero and we need to increase the exponent $PC_f[i].\alpha$ by one. Our active-counter design makes these operations extremely simple: we simply treat $PC_f[i]$ as a two-byte counter and increase it by one, without having to separately consider the first 5 bits for $PC_f[i].\alpha$ and the remaining bits for $PC_f[i].\beta$. More specifically, if the lower 11 bits of $PC_f[i]$ are ones, after increasing by one, they will automatically become zeros and the upper 5 bits will automatically increase by one.

An example of increasing $PC_f[i]$ is shown in Figure 2. Suppose the exponent $PC_f[i].\alpha$ is zero. The recording probability is 1. The counter $PC_f[i].\beta$ will always be increased whenever a packet of flow f arrives. Suppose now $PC_f[i].\beta = 1...1$ as shown in the figure. With an implicit leftmost bit of 1, it contains 12 bits of 1. One additional packet will cause it overflow to become 10...0, which is an implicit leftmost 1 and 12 bits of 0. There is only 11-bit room in $PC_f[i].\beta$. We scale it to 11 bits of 0 by increasing the exponent $PC_f[i].\alpha$. The final result is also shown in the figure, which is simply the previous value of $PC_f[i]$ plus one.

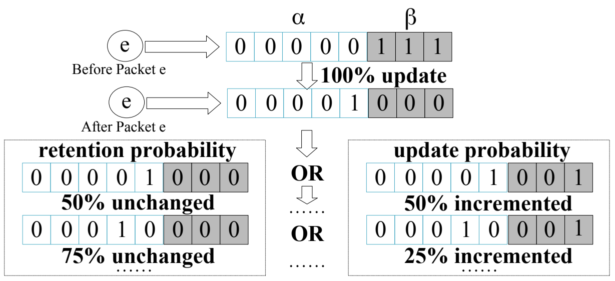


Fig. 2. Three examples for recording a packet in an active counter

Querying: When querying the size of a flow f, we locate the d active counters in PC_f, and return their minimum value as the estimated flow size $\hat{n_f}$.

$$\hat{n_f} = \min_{0 \le i < d}\{V(PC_f[i])\}. \tag{9}$$

Hash Acceleration: ActiveCM requires d hashes per packet. For theoretical analysis, it is desired that the d hashes are independent. But when efficiency outweights, this requirement may be relaxed in practice. We propose a hash-acceleration method that uses one hash to replace the d ones. It performs well in our experiments.

After computing one 64-bit hash value on flow f, we split it into d segments and use each segment in place of $H_i(f)$ to select a counter from PC_* for the virtual counter array of f. For example, if $d = 4$, each segment of the hash value is 16 bits long and can be used to select a counter from a physical array with $m \le 2^{16}$. If m is larger, we will need to perform addition hash(es) for more hash bits.

C. ActiveCM+

ActiveCM records a packet stream and provides size estimate of any given flow, just as CM and CS do, but using smaller memory. Next we extend it for tracking the top-k heavy hitters. To do so, we augment ActiveCM with a min-heap to keep k flow IDs and an associated counter for each ID.

The prior method of combining a CM/CU/CS sketch with a min-heap is as follows [3]: When a packet of flow f arrives, we record the packet in the sketch and also in the min-heap if f is found there. While recording the packet in the sketch, we also make a query on the size of f. If the estimated size is smaller than that of the min-heap root, we will remove the flow in the current root node while adding f and its estimated size to the min-heap. The per-packet overhead of this method include both the sketch operation and the min-heap operation.

Adopting a design choice similar to [13], we use the min-heap as a frontal filter to the sketch such that the packets of top-k heavy hitters only incur overhead for the min-heap, but not for the sketch. More specifically, if the flow ID is in the min-heap, we increase its counter and bypass the sketch. Only if the flow is not in the min-heap, we update the sketch. When we remove a flow f (with its counter c) from the min-heap, we put it back to the sketch by replacing each counter smaller than c in the virtual counter array of f with the value of c. The impact of min-heap filtering is significant because a small number of large flows often accounts for a large proportion of the traffic. By our experiments, when $d = 4$, ActiveCM+ sketch performs 3.18 memory accesses per packet on average, even less than d, thanks to the min-heap filter. Such a design is particularly beneficial when we can implement the min-heap in hardware such as FPGA due to its small size or within a high-speed cache and using SIMD on the processor chip [13].

V. Analysis of ActiveCM

In this section, we will prove that the absolute error of the flow size estimation in (9) by our ActiveCM is upper bounded: $Pr\{\hat{n_f} \ge n_f + (n - n_f)\frac{2d}{m}\} \le \left(\frac{1}{2}\right)^d$. As a result, when the sketch sizes m and d are configured large enough, the heavy hitters defined in (2) can be identified with accuracy guarantees of (3) and (4). Due to page limit, we leave this detailed proof to a technical report for the extended version of this paper.

Let n be the total number of packets for all flows. For an arbitrary flow f, let n_f be its actual flow size. Let n_d be the number of packets mapped to the virtual active counter array PC_f of flow f. Note that, due to counter sharing, other flows may also have their packets mapped to PC_f. Hence, n_d is the flow f's size plus the noise introduced by other flows. Let Y be the number of "noise" packets (from other flows) that are recorded by the d counters in PC_f. We have

$$Y = n_d - n_f. \tag{10}$$

Let m be the number of active counters in PC_*. Then, the probability that a virtual counter of the flow f introduces noise is $\frac{1}{m}$. Since each "noise" packet is mapped d times to update d different counters, the noise Y follows a Binomial distribution.

$$Y \sim Binom\left(d(n - n_f), \frac{1}{m}\right) \tag{11}$$

The flow f has d active counters in PC_f to give d independent estimations about its size, which are later combined by (9). The estimated value given by $PC_f[i]$ using (8) is denoted by $\hat{n_d}$. Here, we omit the symbol i, since clearly each counter $PC_f[i]$ follows the same probability distribution. According to [2], under the $(a+b)$-bit counter scheme, the estimated value $\hat{n_d}$ have the following expected value and variance.

$$E(\hat{n_d}) = n_d, \qquad Var(\hat{n_d}) = \frac{0.6742\, n_d^2}{2^a} \tag{12}$$

This equation states that $\hat{n_d}$ is an unbiased estimation of n_d. Its standard deviation, which depends on the number of bits a (by default, $a = 11$) given to the coefficient α, is very small.

According to (10), considering the noise problem of the virtual counter array PC_f, let $Y = l$ with $l \in [0, d(n-n_f)]$,

$$E(\hat{n_d} \,|\, Y = l) = n_d = n_f + l. \tag{13}$$

Combining it with equations (10), (11) and (12), we can obtain the relationship between $\hat{n_d}$ and n_f.

$$\begin{aligned} E(\hat{n_d}) &= \textstyle\sum_{l=0}^{d(n-n_f)} E(\hat{n_d} \,|\, Y = l) \cdot Pr(Y = l) \\ &= \textstyle\sum_{l=0}^{d(n-n_f)} (n_f + l) \cdot \binom{d(n-n_f)}{l} (\frac{1}{m})^l (1 - \frac{1}{m})^{d(n-n_f)-l} \\ &= n_f + (n - n_f)\frac{d}{m} \end{aligned} \tag{14}$$

Combining formula (14) with Markov's inequality, the error bound of a single virtual active counter is as follows.

$$\begin{aligned} Pr(\hat{n_d} \geq n_f + (n-n_f)\tfrac{2d}{m}) &= Pr(\hat{n_d} - n_f \geq (n-n_f)\tfrac{2d}{m}) \\ &\leq \tfrac{m}{2d(n-n_f)}(E(\hat{n_d}) - n_f) \\ &= \tfrac{m}{2d(n-n_f)}(n-n_f)\tfrac{d}{m} = \tfrac{1}{2} \end{aligned} \tag{15}$$

Since the estimated size $\hat{n_f}$ of the flow f is the minimum value of the d virtual active counters, combining equations (9) and (15), the estimation $\hat{n_f}$ is upper bounded:

$$\begin{aligned} &Pr(\hat{n_f} \geq n_f + (n-n_f)\tfrac{2d}{m}) \\ &= \textstyle\prod_{0 \leq i < d} Pr(\hat{n_d}^{(i)} \geq n_f + (n-n_f)\tfrac{2d}{m}) \leq \left(\tfrac{1}{2}\right)^d. \end{aligned} \tag{16}$$

VI. Progressive Universal Sketch

We present a new universal sketch named LUS. We firstly introduce the motivation and basic idea behind our design. Next, we describe its detailed algorithm procedure, which is divided into packet insertion phase and moment query phase.

A. Basic Idea

We propose a *progressive sampling* techique. We create an array of subsketches $M_0, M_1, M_2, \ldots, M_\ell$ in memory. Their flow sampling probability reduces exponentially: The flows in the 0th subsketch M_0 have 100% sampling probability; The flows in M_{j+1} is a 50% pseudorandom sample of the flows in M_j. Our idea is that, when a packet arrives carrying a flow ID f, we update the last subsketch where f is sampled. The benefit is that only one subsketch, instead of variably multiple subsketches, needs to update per arrival packet. We call this algorithm *light-weight universal sketch* (LUS).

When a packet arrives in Figure 3, we use its flow ID f to generate a pseudorandom bit array, where each bit equals to one with 50% probability. Clearly, the probability for this array's leading j bits are all ones is $1/2^j$. If it happens, the flow f is sampled in the jth subsketch. Assume the leading j^* bits are all ones and the (j^*+1)th bit is zero. We call j^* the longest run of leading ones. In this case, f is sampled in the $0, 1, \ldots, j^*$th subsketches but not in (j^*+1)th subsketch. We update only the j^*th subsketch to record the packet information, whose number of memory accesses is less than 4, since the subsketch is implemented by ActiveCM+ with $d = 4$.

The progressive sampling technique brings another benefit: The moment estimation error can be reduced by more than half in experiment than UnivMon. The reason is as follows. UnivMon updates all the subsketches $0, 1, \ldots, j^*$, where j^* is the last subsketch having f sampled. UnivMon also uses all these subsketches to query the flow size of f when it estimates the flow moments. However, different subsketches in this list have different estimation accuracy of flow f. Since the j^*th subsketch M_{j^*} has the smallest sampling probability, f experiences the least noise from other flows to affect its size estimation and has the best accuracy in M_{j^*}. So we use only M_{j^*} for the size estimation of f. Better estimation accuracy of heavy hitters will bring higher accuracy in moment estimation.

The third advantage of LUS sketch is that we significantly reduce the time cost of computing moment estimations. We do not use a time-expensive recursive formula like [14], which estimates a moment by reading the entire array of subsketches. Instead, as the arrival packet changes the j^*th subsketch, we use a simplified formula to incrementally update our moment estimation. As a result, our time cost of querying the flow moment is negligbly small, even when it is queried per packet.

B. Detailed Algorithm Procedure

Our algorithm can be divided into two phases, as shown in Figure 3. The insertion phase processes a stream of packets and squeeze them into an array of subsketches[1]. In the query phase, each subsketch reports the heavy hitters among its sampled flows. We use such information to estimate the moment of the 0th subsketch whose sampling probability is 100%.

Insertion Phase. Recall that the flows appearing in the jth subsketch is sampled with 50% probability in the $(j+1)$th subsketch. Let S_0 be the set of all flows IDs. Let S_j be the set of sampled flow IDs in the jth subsketch. Then, we have

$$S_0 = \{1, 2, \ldots, \mathcal{F}\}, \qquad S_j = \{f \mid f \in S_{j-1} \,\wedge\, h_j(f) = 1\} \tag{17}$$

where h is the hash function applied to the flow ID f for sampling. We interpret the hash value $h(f)$ as a bit array. Let $h_j(f)$ be the jth bit of this array. Clearly, when the j leading consecutive bits of $h(f)$ are all ones, we have $f \in S_j$, and we say the flow f is sampled in the jth subsketch M_j.

The pseudocode of the insertion phase is given in Algorithm 1. When an IP packet arrives carrying a flow ID f, instead of updating each subsketch M_j with $0 \leq j \leq j^*$ and satisfying $f \in S_j$, the line 4 updates only the j^*th subsketch with

$$j^* = \min\left(\ell,\ \arg\max_j \left\{ \textstyle\bigwedge_{1 \leq i \leq j} h_i(f) = 1 \right\}\right). \tag{18}$$

[1]A subsketch is normally implemented by ActiveCM+. But to make the last subsketch M_ℓ more accurate, it is implemented by a hash table to record the per-flow sizes of sampled flows. Since the number of subsketches is above 12, e.g., 14, it is sufficient to let the hash table to hold about 100 flows. All flows recorded in this hash table are regarded as heavy hitters of the last subsketch.

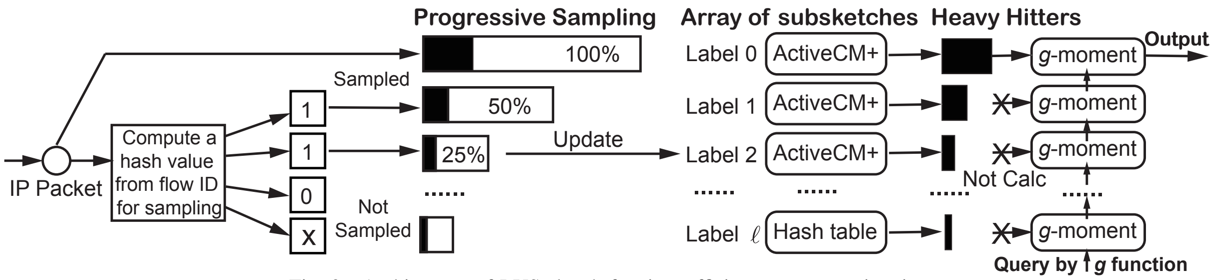


Fig. 3. Architecture of LUS sketch for time-efficient moment estimation.

Here, $\bigwedge_{1\leq i\leq j} h_i(f)=1$ implies the leading j bits of $h(f)$ are all ones, $\arg\max_j$ finds the longest run of leading consecutive ones in the bit array of $h(f)$, $\min(\dots)$ returns the minimum value of its parameters, and ℓ is the largest subsketch label.

The last subsketch M_ℓ records the per-flow size information of all flows in S_ℓ. This is because any flow $f \in S_\ell$ has at least ℓ consecutive leading ones in $h(f)$, making its j^* equal to ℓ by (18). For another subsketch M_j with $j<\ell$, it records only the flows in the relative complement set $S_j \setminus S_{j+1}$. This is because any flow $f \in S_{j+1}$ has at least $j+1$ consecutive leading ones, making its j^* larger than j by (18).

Algorithm 1: LUS Insertion Phase

```
Input: Packet stream $\langle f_1,c_1\rangle,\dots\langle f_t,c_t\rangle,\dots\langle f_n,c_n\rangle$
Output: Heavy hitters sets $\{\hat{H_0},\dots,\hat{H_\ell}\}$ among the
        sampled flows $\{S_0,\dots,S_\ell\}$, respectively
1 From an arrival packet, get flow ID $f$ and packet size $c$;
2 Compute a hash value $h(f)$ from the flow ID $f$;
3 Calc $j^* = \min\left(\ell,\ \arg\max_j \left\{\bigwedge_{1\leq i\leq j} h_i(f)=1\right\}\right)$;
4 Insert the tuple $\langle f,c\rangle$ to the subsketch $M_{j^*}$ with $\hat{H_{j^*}}$
    filter, and query $M_{j^*}$ for a flow size estimate $\hat{n_f}$;
5 if flow $f$ is a heavy hitter in the filter $\hat{H_{j^*}}$ then
6     Use the tuple $\langle f,\hat{n_f}\rangle$ to update the sets of heavy
      hitters $\{\hat{H}_{j^*-1},\dots,\hat{H_0}\}$ among $\{S_{j^*-1},\dots,S_0\}$;
7 end
```

Query Phase. This phase can estimate the flow moment of the sampled flows in each subsketch. Let L_j be the moment of the sampled flows S_j in the jth subsketch, $0\leq j\leq \ell$. Then,

$$L_j = \textstyle\sum_{f\in S_j} g(n_f), \tag{19}$$

where S_j is the set of sampled flows as in (17), n_f is the size of flow f, and $g(x)$ is a monotonic function bounded by x^2. Since the last subsketch is simply a hash table to hold all flows in S_ℓ, the moment L_ℓ can be estimated as $\hat{L_\ell} = \sum_{f\in S_\ell} g(\hat{n_f})$.

The moment estimation needs each subsketch to report a set of heavy hitters. Let H_k be the set of heavy hitters among the sampled flows S_k. Similar to the heavy hitter definition in (2),

$$H_j = \{f \mid f\in S_j \ \wedge\ g(n_f)\geq \alpha L_j\}. \tag{20}$$

The identifcation of heavy hitters H_j among the sampled flow set S_j has been implemented by Algorithm 1 at lines 4-7.

We may use the same method proposed by the original theoretical work [14] to estimate the moment of the flow set S_j.

$$\hat{L_j} = 2\hat{L}_{j+1} + \textstyle\sum_{f\in\hat{H_j}} (1-2h_{j+1}(f))g(\hat{n_f}) \tag{21}$$

This recursive formula computes the moment of each subsketch, from the last to the 0th subsketch, and outputs the moment estimation $\hat{L_0}$ whose flow sampling probability is 100%. Its high computation cost prevents to perform evaluation per packet for supporting online estimation of moments. We will dramatically reduce the time cost by only estimating the moment L_0, and we update $\hat{L_0}$ incrementally when each packet arrives. The pseudocode of our solution is in Algorithm 2.

Algorithm 2: LUS Query Phase

```
Input: Heavy hitters $\{\hat{H_0},\dots,\hat{H_\ell}\}$, Moment function $g$
Output: Moment estimation $\hat{L_0}$ of the flow ID set $S_0$
1 if the flow ID $f$ of the arrival packet is a heavy hitter
    in $\hat{H_{j^*}}$ then $j'=j^*$;  else return $\hat{L_0}$;
2 Calc the increment of $\hat{L_{j^*}}$ as $\Delta_f = g(\hat{n_f}^{\text{new}}) - g(\hat{n_f}^{\text{old}})$
3 for $j=j^*-1,\dots,0$ do
4     if $f\in\hat{H_j}$ then $j'=j$;  else break;
5 end
6 return $\hat{L_0} = \hat{L_0} + 2^{j'}\,\Delta_f$;
```

When a packet with flow ID f comes, we locate the j^*th subsketch, where j^* is the largest index of the subsketch where f is sampled as in (18). To save computational time cost, our Algorithm 2 avoids to update all moment estimations $\hat{L_{j^*}},\dots,\hat{L_0}$. It updates only the moment estimation $\hat{L_0}$ of the complete flow set S_0, which network operators are interestd in. Here, we suppose $\hat{L_0}$ is held in on-chip high-speed cache, closer to processor than off-chip memory, so that the time cost of accessing the memory unit of $\hat{L_0}$ is negligbly small.

At line 1, we check whether f is a heavy hitter among the sampled flow set S_{j^*}. If not, we ignore the packet and leave the moment estimation $\hat{L_0}$ unchanged. If f is not a heavy hitter in S_{j^*}, f will not be in other sampled sets S_j with $j<j^*$. So we can ignore the packet. This is because, when the subsketch index j reduces, the sampling probability increases, and the flow set S_j expands. This will make it more difficult for f to become a heavy hitter among S_j than among S_{j^*}.

Now, we know the flow f is a heavy hitter among S_{j^*}. At line 2, we compute Δ_f, which is the increment of the g-moment of S_{j^*} caused by the arrival packet. Here we need both the old and the new size estimations of flow f, which can be obtained when we update the subsketch M_{j^*} at line 4 of Algorithm 1. Note that line 2 is in accordance with (21), since $f \in \hat{H_{j^*}}$ and $h_{j^*+1}(f) = 0$. At lines 3-5, we search for the flow set $S_{j'}$ with the smallest index (or the largest sampling probability), where f remains a heavy hitter (i.e., $f \in \hat{H_{j'}}$). Therefore, the moment $\hat{L_{j'}}$ increases by Δ_f, which is consistent with (21). Now, we know f is not a heavy hitter but a mouse flow in $S_{j'-1}, \ldots, S_0$. To estimate the increases of moments $L_{j'-1}, \ldots, L_0$, we need to multiply Δ_f by $2^1, \ldots, 2^{j'}$, respectively. Line 6 only updates the moment estimate $\hat{L_0}$ by adding $2^{j'}\Delta_f$. It is our estimate of moment L_0.

The heavy hitters H_j in (20) have been estimated as $\hat{H_j}$, by Algorithm 1. It can be proved that, if $\hat{H_j}$ can satisfy the accuracy constraints in (3) and (4), then the g-moment L_g of all flows (in this section, L_0 for short) can be estimated with accuracy constraint in (5). The proof is somewhat similar to [14]. We leave the proof to the extended version of this paper.

VII. Experimental Evaluation

In this section, we implement several existing heavy hitter detection sketches, including CountMin Sketch (CM) [3], Conservative Update Sketch (CU) [11], Count Sketch (CS) [4], Virtual Active Counter (VAC) [2], and Self-Adaptive Counters for CountMin (SA-CM) [15]. We compare their performance with our ActiveCM+ when given the same memory, to show the advantage of ActiveCM+ for online tracking heavy hitters. Next, we compare our LUS sketch with UnivMon [6] to show its performance advantage for online estimating flow moments. Our IP traffic traces for evaluation are from CAIDA [16].

Performance Metrics. We consider the application scenario of online tracking heavy hitters and online estimating moments for the prompt detection of network anomalies. Hence, for the high-speed packet stream, we suppose the insertion of packet information into the sketch and the querying of flow states are performed per packet. This places stringent demand on packet processing throughput. Therefore, we will evaluate the packet throughput (quantified by items per second), which is primarily determined by the average number of hashes and memory accesses per packet for insertion and querying. We will also evaluate the average estimation accuracy of heavy hitters and flow moments, which have been formalized in Section II-B.

A. Performance of Heavy Hitter Detection

Packet Throughput. We compare the packet processing throughput of several heavy hitter detection algorithms in Figure 4. It shows that the packet throughput of our ActiveCM+ is about 1.14, 1.16, 1.89 higher than those of the CM, CU and CS sketches, respectively. The main reason is that our ActiveCM sketch with $d = 4$ needs slightly more than four memory accesses for packet insertion (four reads plus occasionally a few write backs), while CM, CU, CS sketches need at least eight memory accesses per packet. The insertion throughput of VAC is higher than ours, since VAC needs slightly more than one memory accesses per packet for packet insertion. But VAC needs a few hundreds memory reads to query the size of a flow. Hence, the overall throughput of our ActiveCM+ is 18.6 times higher than VAC. SA-CM is perhaps the latest work for heavy hitter detection [15]. Compared with it, our ActiveCM+ is still 1.05 times faster, because besides the active counter technique, we have adopted also the frontal filtering strategy for heavy hitters, which is proposed in Section IV-C.

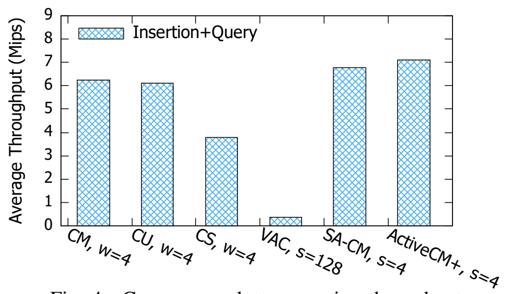
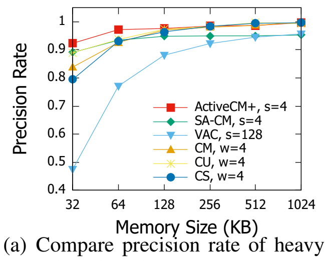


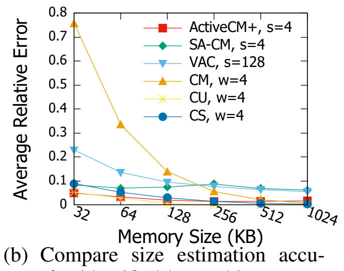
Fig. 4. Compare packet processing throughput.

Estimation Accuracy. Firstly, in Figure 5(a), we compare the heavy hitter identification accuracy of different algorithms including CM, CU, CS, VAC, SA-CM and ActiveCM+, when they are given the same amount of memory. We use precision ($\frac{\text{TPs}}{\text{TPs+FPs}}$) to quantify the identification accuracy. It shows that our ActiveCM+ is the highest, about 1.10, 1.04, 1.16, 1.94,1.04 times higher than CM, CU, CS, VAC, SA-CM, respectively.

Secondly, we evaluate the flow size estimation accuracy based on Average Relative Error (ARE). ARE is the relative difference between the estimated flow size and the true flow size: $ARE = \frac{1}{|N|}\sum_{f \in N} \frac{|n_f - \hat{n_f}|}{n_f}$, Where N is the set of heavy flows to be queried, and $\hat{n_f}$ is the estimated value of the size n_f of the flow f. Figure 5(b) shows that the AREs of the CM, CU, CS sketches are higher than the ARE of our ActiveCM+. This is because ActiveCM+ uses the active counter technique [12] to compress the counter size by half. As a result, more counters can be allocated from the same amount of memory to undertake the counting job, which improves the accuracy.



(a) Compare precision rate of heavy hitter identification



(b) Compare size estimation accuracy for identified heavy hitters

Fig. 5. Compare heavy hitters estimation accuracy.

Impact of Virtual Counter Array Size. We evaluate the impact of d, the size of virtual counter array, on the estimation accuracy of our ActiveCM+. We plot the evaluation result in Table I. It shows that we can minimize the estimation error when we configure $d = 4$ or $d = 5$. But smaller d value means smaller insertion and querying time cost of our sketch. Thus, $d = 4$ is the recommended parameter setting for ActiveCM+.

B. Performance of Universal Sketch

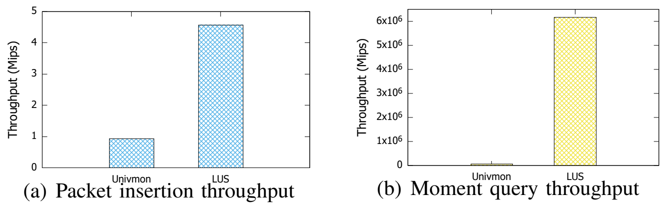
In this subsection, we compare the performance of universal sketches, i.e., UnivMon and our LUS. UnivMon is configured

TABLE I
IMPACT OF THE VIRTUAL COUNTER ARRAY SIZE d

Value of d	ARE in different Memory size		
	1 MB	2 MB	3 MB
2	4.042715	1.783720	0.735967
3	3.281323	1.226965	0.367778
4	3.210663	1.008244	0.277993
5	3.416525	0.992986	0.255877
6	3.841261	1.038691	0.265438
7	4.376923	1.117811	0.285739

with 14 hierarchical layers, while LUS is given 10 or 14 subsketches, and has $d = 4$ for its underlying ActiveCM+.

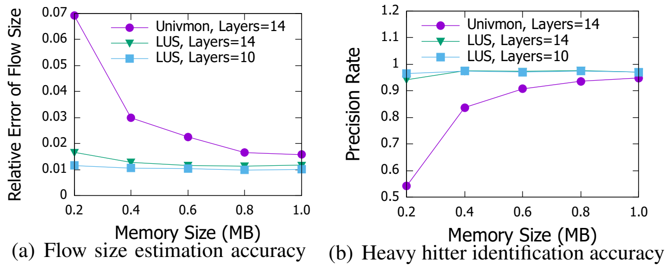
Firstly, we compare the packet processing throughput in Figure 6. The plot (a) shows that the insertion throughput of our LUS is about 4.85 times higher than Univmon. This is because our LUS uses the progressive sampling technique which only needs to update one subsketch per packet, and its subsketch is ActiveCM+, which is more time efficient than CountSketch as in Fig. 4. The plot (b) shows that, when performing online flow moment query, our LUS is dramatically higher than Univmon. This is because we adopt an incremental updating strategy for moment estimation, which is implemented in Algorithm 2.



(a) Packet insertion throughput (b) Moment query throughput

Fig. 6. Compare packet processing throughput

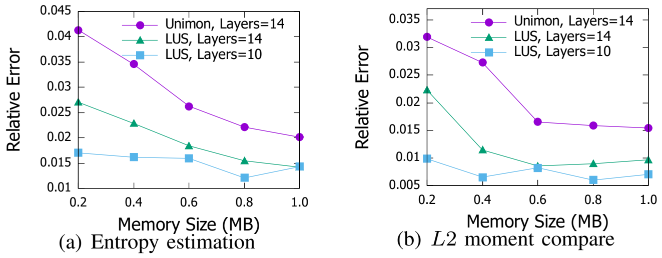
Secondly, we compare the heavy hitter estimation accuracy in Fig. 7. The plot (a) shows LUS is much more accurate than UnivMon especially when the memory is limited. This is because LUS inserts a flow's packets into a single subsketch, where the flow is sampled and the sampling rate is the smallest as in (18). In that subsketch, the noise from other flows is minimized. The plot (b) shows the LUS always has higher precision rate for heavy hitter identification than UnivMon. This is because LUS has better estimation accuracy for flow sizes, which helps to better differentiate which flows are heavy.



(a) Flow size estimation accuracy (b) Heavy hitter identification accuracy

Fig. 7. Compare heavy hitter estimation accuracy

Thirdly, we compare the moment estimation accuracy of LUS and UnivMon, including entropy in Fig. 8(a) and 2nd-order moments in Fig. 8(b). They show that LUS offers better moment estimation accuracy for most memory size settings, due to our progressive sampling technique and ActiveCM+, which attains better heavy hitter estimation accuracy in Fig. 7.



(a) Entropy estimation (b) $L2$ moment compare

Fig. 8. Compare flow moment estimation accuracy.

Finally, we evaluate the impact of the number of subsketches on the partial flow moments, when the allocated memory is equal to 0.6MB. We plot the evaluation result in Figure 9. It shows that UnivMon achieves better moment estimation accuracy as the number of layers grows, and the accuracy stablizes when greater than 12. It also shows that the flow moment accuracy of our LUS is not obviously affected by the number of subsketches, because our last subsketch is a hash table that accommodates all sampled flows.

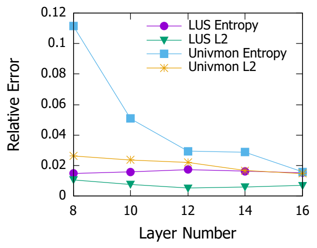


Fig. 9. Impact of the number of subsketches $\ell + 1$.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China under grant 61872080, by Jiangsu Provincial Key Lab of Computer Networking Technology, by National Science Foundation of USA under grants STC-1562485 and CNS-1719222, and a grant from Florida Cybersecurity Center.

VIII. CONCLUSION

In this paper, we firstly present a highly efficient counter sharing architecture for online tracking heavy hitters in a data stream, called CountMin with Active counters plus (ActiveCM+). It reduces the number of per-packet memory accesses by half and reduces the memory footprint by half than CountMin sketch, thanks to the active counter compression and heavy hitter filtering technique. Next, based on the ActiveCM+, we present an algorithm called light-weight universal sketch (LUS), designed for online estimating moments of flow size distribution. Compared with UnivMon, our sketch needs 16 times smaller number of per-packet memory accesses, and provides 3 times better accuracy when given the same memory. Therefore, our LUS sketch becomes more time-efficient for online estimating moments, including cardinality, entropy and second-order moment. We have verified the superior performance of our algorithm by both theoretical analysis and experimental results based on CAIDA traces.

References

[1] L. Tao, S. Chen, and Y. Ling, “Per-flow traffic measurement through randomized counter sharing,” *IEEE/ACM TON*, vol. 20, no. 5, 2012.

[2] Y. Zhou, Y. Zhou, S. Chen, and Y. Zhang, “Highly compact virtual active counters for per-flow traffic measurement,” *IEEE INFOCOM*, 2018.

[3] G. Cormode and S. Muthukrishnan, “An improved data stream summary: The count-min sketch and its applications,” *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[4] M. Charikar, K. C. Chen, and M. Farach-Colton, “Finding frequent items in data streams,” *Theor. Comput. Sci.*, vol. 312, pp. 3–15, 2004.

[5] R. Ben-Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, “Constant time updates in hierarchical heavy hitters,” *Proc. of ACM SIGCOMM*, 2017.

[6] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, “One sketch to rule them all: Rethinking network flow monitoring with univmon,” *Proc. of ACM SIGCOMM*, 2016.

[7] M. Yu, L. Jose, and R. Miao, “Software defined traffic measurement with OpenSketch,” *Proc. of NSDI*, 2013.

[8] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi, “Mergeable summaries,” *ACM TODS*, vol. 38, no. 4, 2013.

[9] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, “Data streaming algorithms for estimating entropy of network traffic,” *Proc. of ACM SIGMETRICS*, pp. 145–156, 2006.

[10] E. Assaf, R. B. Basat, G. Einziger, and R. Friedman, “Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free,” *Proc. of IEEE INFOCOM*, 2017.

[11] C. Estan and G. Varghese, “New directions in traffic measurement and accounting,” *Computer Communication Review*, vol. 32, no. 4, 2002.

[12] R. Stanojevic, “Small active counters,” *Proc. of IEEE INFOCOM*, 2007.

[13] P. Roy, A. Khan, and G. Alonso, “Augmented sketch: Faster and more accurate stream processing,” *Proc. of ACM SIGMOD*, 2016.

[14] V. Braverman and R. Ostrovsky, “Generalizing the layering method of Indyk and Woodruff: Recursive sketches for frequency-based vectors on streams,” *APPROX Workshop*, pp. 58–70, 2013.

[15] 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,” *Proc. of IEEE INFOCOM*, 2019.

[16] CAIDA, “The caida anonymized internet traces,” 2016.

[17] Y. Liu, W. Chen, and Y. Guan, “Identifying high-cardinality hosts from network-wide traffic measurements,” *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 5, pp. 547–558, 2016.

[18] R. T. Schweller, A. Gupta, E. Parsons, and C. Yan, “Reversible sketches for efficient and accurate change detection over network data streams,” *Proc. of ACM SIGCOMM*, 2004.

[19] A. Kumar, M. Sung, J. Xu, and W. Jia, “Data streaming algorithms for efficient and accurate estimation of flow size distribution.” *Proc. of ACM SIGMETRICS*, 2004.

[20] A. Metwally, D. Agrawal, and A. E. Abbadi, “Efficient computation of frequent and top-k elements in data streams,” *Proc. of ICDT*, 2005.

[21] J. Jiang and S. Uhlig, “Elastic sketch : Adaptive and fast network-wide measurements,” *Proc. of ACM SIGCOMM*, pp. 561–575, 2018.

[22] M. Chen, S. Chen, and Z. Cai, “Counter tree: A scalable counter architecture for per-flow traffic measurement,” *IEEE/ACM TON (Trans. on Networking)*, vol. 25, no. 2, pp. 1249–1262, 2017.

[23] Y. Qiao, T. Li, and S. Chen, “One memory access bloom filters and their generalization,” *Proc of IEEE INFOCOM*, pp. 1745–1753, 2011.

[24] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, “Counter braids: a novel counter architecture for per-flow measurement,” *Proc. of ACM SIGMETRICS*, 2008.

[25] N. Bandi, A. Metwally, D. Agrawal, and A. El Abbadi, “Fast data stream algorithms using associative memories,” *Proc. of ACM SIGMOD*, 2007.