Adaptive Measurements Using One Elastic Sketch

Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou,
Rui Miao, Xiaoming Li, and Steve Uhlig

***Abstract*— When network is undergoing problems such as congestion, scan attack, DDoS attack, *etc*, measurements are much more important than usual. In this case, traffic characteristics including available bandwidth, packet rate, and flow size distribution vary drastically, significantly degrading the performance of measurements. To address this issue, we propose the Elastic sketch. It is adaptive to currently traffic characteristics. Besides, it is generic to measurement tasks and platforms. We implement the Elastic sketch on six platforms: P4, FPGA, GPU, CPU, multi-core CPU, and OVS, to process six typical measurement tasks. Experimental results and theoretical analysis show that the Elastic sketch can adapt well to traffic characteristics. Compared to the state-of-the-art, the Elastic sketch achieves $44.6 \sim 45.2$ times faster speed and $2.0 \sim 273.7$ smaller error rate.**

***Index Terms*— Sketches, network measurements, elastic, compression, generic.**

I. Introduction

NETWORK measurements provide indispensable information for network operations, quality of service, capacity planning, network accounting and billing, congestion control, anomaly detection in data centers and backbone networks [2]–[16]. Recently, sketch-based solutions[1] [7], [17] have been widely accepted in network measurements [3], [4], [18], [19], thanks to their higher accuracy compared to sampling methods [3], [19], [20] and their speed.

Existing measurement solutions [7], [17], [19]–[25] mainly focus on a good trade-off among accuracy, speed and memory usage. The state-of-the-art UnivMon [3] pays attention to an additional aspect, generality, namely using one sketch to process many tasks, and makes a good trade-off among these four dimensions. Although existing work has made great contributions, they do not focus on one fundamental need: achieving accurate network measurements no matter how traffic characteristics (including available bandwidth, flow size distribution, and packet rate) vary. Measurements are especially important when network is undergoing problems, such as network congestion, scans and DDoS attacks. In such cases, traffic characteristics vary drastically, significantly degrading the measurement performance. Therefore, it is desirable to achieve accurate network measurements when traffic characteristics vary a lot.

The first traffic characteristic is the available bandwidth. In data centers, administrators care more about the state of the whole network than a single link or node, known as network-wide measurements [3], [19], [26]. In data centers, administrators can deploy many measurement nodes, which periodically report sketches to a collector [3], [19], [26]. It requires available bandwidth for measurements, which share the same data plane as the user traffic. However, in data centers, network congestion is common [27]. It can happen frequently within a single second [28] and be as large as more than half of the network bandwidth [8]. In this case, on the one hand, measurements are especially critical for congestion control and troubleshooting. One cannot wait for the available bandwidth to be sufficient to report the sketches, because network problems should be handled immediately. On the other hand, network measurements should not be a burden for the network, as pointed out in [29]–[31]. A good solution is to actively compress the sketch with little accuracy loss, thereby reducing bandwidth usage. Therefore, it is desirable to compress the sketch. This has not been done before in the literature. Besides passive compression during congestion, network operators need to proactively control the measurement tasks as well. For example, to keep service-level agreements (SLA) during maintenance or failures [32], operators tend to reduce measurements and leave the bandwidth for critical user traffic.

The second characteristic is the packet arrival rate (packet rate for short) [33], [34], which could vary drastically. For example, some routing protocols or mechanisms are proposed to adjust the packet sending rate to optimize network performance [35]–[37]. Also, when the network is under attack (*e.g.*, a network scan or a DDoS attack), most packets tend to be small. In this case, the packet rate is very high, even though the available bandwidth might still be significant. The processing speed of existing sketches on software platforms is fixed in terms of packet rate. Therefore, it does not work well when the packet rate suddenly becomes much higher, likely failing to record important information, such as the IP addresses of attackers. Therefore, in this case, it is desirable to accelerate

Manuscript received July 24, 2018; revised May 18, 2019; accepted August 30, 2019; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Argyraki. This work was supported in part by the Primary Research & Development Plan of China under Grant 2016YFB1000304 and Grant 2018YFB1004403, in part by the NSFC under Grant 61672061, and in part by the project PCL Future Regional Network Facilities for Large-scale Experiments and Applications under Grant PCL2018KP001. The preliminary version of this article will appear in ACM SIGCOMM 2018 [1].

T. Yang is with the Department of Computer and Science, Peking University, Beijing 100871, China, and also with the Peng Cheng Laboratory, Shenzhen 518052, China (e-mail: yangtongemail@gmail.com).

J. Jiang, P. Liu, and X. Li are with the Department of Computer and Science, Peking University, Beijing 100871, China (e-mail: jie.jiang@pku.edu.cn).

Q. Huang is with the Institute of Computing Technology, CAS, Beijing 100190, China (e-mail: huangqun@ict.ac.cn).

J. Gong and Y. Zhou are with Harvard University, Cambridge, MA 02138 USA (e-mail: yangzhou@g.harvard.edu).

R. Miao is with the Alibaba Group, Hangzhou 311121, China (e-mail: miao.rui@alibaba-inc.com).

S. Uhlig is with the Networks School of EECS, Queen Mary University of London, London E1 4NS, U.K. (e-mail: steve.uhlig@quml.ac.uk).

Digital Object Identifier 10.1109/TNET.2019.2943939

[1] In this paper, sketches refers to data streaming algorithms that can be used for network measurements.

the processing speed by actively discarding the unimportant information.

The third characteristic is flow size distribution. It is known that most flows are small [38], referred to as mouse flows, while a very few flows are large, referred to as elephant flows [20], [31], [39]–[41]. An elegant solution is to accurately separate elephant flows from mouse flows, and use different data structures to store them. However, the flow size distribution varies. One might think we can predict traffic and allocate appropriate size of memory for sketches in advance. It may be easy to predict the number of elephant flows in one hour, but hard at timescales of seconds or milliseconds. Therefore, it is desirable to design an elastic data structure which can dynamically allocate appropriate memory size for elephant flows.

In summary, this leads us to require our sketch to be **elastic**: adaptive to bandwidth, packet rate, and flow size distribution. Besides them, there are three other requirements in measurements: 1) generic, 2) fast, and 3) accurate. First, each measurement node often has to perform several tasks. If we build one data structure for each task, processing each incoming packet requires updating all data structures, which is time- and space-consuming. Therefore, one generic data structure for all tasks is desirable. Second, to be fast, the processing time of each packet should be small and constant. Third, being accurate implies that the error rate should be small enough when using a given amount of memory. Among all existing solutions, no solution is elastic, and only two well known solutions claim to be generic: UnivMon [3] and FlowRadar [26]. However, our experimental results in Section VII show that UnivMon is practically not accurate, while FlowRadar is not memory efficient.

In this paper, we propose a novel sketch, namely the Elastic sketch. It is composed of two parts: a *heavy part* and a *light part*. We propose a separation technique named **Ostracism** to keep elephant flows in the heavy part, and mouse flows in the light part.

To make it "elastic", we do the following. 1) To be adaptive to bandwidth, we propose algorithms to compress and merge sketches. First, we can compress our sketch into an appropriate size to fit the current available bandwidth. Second, we can use servers to merge sketches, and reduce the bandwidth usage. 2) When the packet rate becomes high, we change the processing method: each packet only accesses the heavy part to record the information of elephant flows exclusively, discarding the information of mouse flows. In this way, we can achieve much faster processing speed at the cost of reasonable accuracy drop. 3) As the number of elephant flows varies and is unknown in advance, we propose an algorithm to dynamically increase the memory size of the heavy part.

To make our solution "generic", we do the following. 1) To be generic in terms of measurement tasks, we keep all necessary information for each packet, but discard the IDs of mouse flows, which is based on our observation that the IDs of mouse flows are memory consuming but practically useless. 2) To be generic in terms of platforms, we propose a software and a hardware version of the Elastic sketch, to make our sketch easy to be implemented on both software and hardware platforms. Further, we tailor a P4 version of the Elastic sketch, given the popularity of this platform [42].

Owing to the separation and discarding of unnecessary information, our sketch is accurate and fast: experimental results show that our sketch achieves $44.6 \sim 45.2$ times faster speed and $2.0 \sim 273.7$ smaller error rate than the state-of-the-art: UnivMon [3].

In this paper, we make the following contributions:

- We propose a novel sketch for network measurements, namely the Elastic sketch. Different from previous work, we mainly focus on the ability of the sketch to adapt to bandwidth, packet rate and flow size distribution. The Elastic sketch is also generic, fast and accurate. We propose two key techniques, one to separate elephant flows from mouse flows, and another for sketch compression.
- We implement our sketch on six platforms: P4, FPGA, GPU, CPU, multi-core CPU, and OVS, to process six typical measurement tasks.
- Experimental results show that our sketch works well on all platforms, and significantly outperforms the state-of-the-art for each of the six tasks.

II. Background and Related Work

A. Challenges of Adaptive Measurements

As mentioned above, when network does not work well, the network measurement is especially important. In this case, traffic characteristics vary drastically, posing great challenges for measurement.

First, it is challenging to send measurement data (*e.g.*, sketch) in appropriate size according to the available bandwidth. When the available bandwidth is small, sending a large sketch will cause long latency and affect user traffics. Furthermore, all existing solutions fix the memory size before starting measurement. The problem is how to make the sketch size smaller than the available bandwidth, especially when network does not work well. A naive solution is to build sketches in different sizes for the same network traffic. For example, one can build two sketches $\mathcal{S}_1$, $\mathcal{S}_2$ with the memory size of M and M/2, and then we can send $\mathcal{S}_2$ to the collector when the available bandwidth is small. A better solution is to build only $\mathcal{S}_1$, and quickly compress it into a half. It is not hard for the compressed $\mathcal{S}_1$ to achieve the same accuracy with $\mathcal{S}_2$. However, it is challenging for the compressed $\mathcal{S}_1$ to achieve much higher accuracy than $\mathcal{S}_2$, which is one design goal of this paper.

Second, it is challenging to make the processing speed adaptive to the packet rate, which could vary drastically during congestion or attack. Existing sketches often have constant processing speed, but require several or even more than 10 memory access for processing one packet. The design goal is 2 memory accesses for processing each packet when packet rate is low, and 1 memory access when packet rate is high. However, it is challenging to keep high accuracy when using only one memory access.

Third, in real network traffic, the flow size distribution is skewed and variable. "Skewed" means most flows are mouse flows [38], while a few flows are elephant flows [20], [31], [39]. To achieve memory efficiency, one can manage to separate elephant flows from mouse flows. As elephant flows are often more important than mouse flows, it is desirable to assign appropriate memory size for the elephant flows. Unfortunately, the number of elephant flows is not known in advance and hard to predict [43]. Therefore, it is challenging to dynamically allocate more memory for the elephant flows.

B. Generic Method for Measurements

We focus on the following network measurement tasks, and more tasks can be found in [7], [44]–[47].

Flow Size Estimation: estimating the flow size for any flow ID. A flow ID can be any combinations of the 5-tuple, such as source IP address and source port, or only protocol. In this paper, we consider the number of packets of a flow as the flow size. This can be also used for estimating the number of bytes for each flow: assuming the minimal packet is 64 bytes, given an incoming packet with 120 bytes, we consider it as $\lceil \frac{120}{64} \rceil = 2$ packets.

Heavy Hitter Detection: reporting flows whose sizes are larger than a predefined threshold.

Heavy Change Detection: reporting flows whose sizes in two adjacent time windows increase or decrease beyond a predefined threshold, to detect anomalous traffic.

Flow size Distribution Estimation: estimating the distribution of flow sizes.

Entropy Estimation: estimating the entropy of flow sizes.

Cardinality Estimation: estimating the number of flows.

Generic solutions can use one data structure to support all these measurement tasks. If the IDs and sizes of all the flows are recorded, then we can process these tasks, but recording all flow IDs is difficult and needs high memory usage [19], [20]. We observe that *flow IDs of mouse flows are not necessary* for these tasks. As most flows are mouse flows, discarding IDs of mouse flows can significantly save memory and bandwidth. For this, we need to separate elephant flows from mouse flows. To address this problem, we leverage the spirit of Ostracism, and propose a fast and accurate separation algorithm. Finally, our sketch is both generic and memory efficient.

Another meaning of generic is that the algorithm can be implemented on various platforms. For small companies, the traffic speed may be not high, and measurement on CPU is a good choice. For large companies, the traffic speed could be very high, and then hardware platforms should be used for measurements to catch up with the high speed. Therefore, the measurement solution should be generic, and can make good performance trade-off on different platforms.

CounterBraids [48] is an excellent work when knowing all flow IDs. However, we didn't compare with it because CounterBraids and the Elastic sketch focus on different goals when measuring network traffic. There are three typical differences between CounterBraids and Elastic. 1) For accuracy, Elastic allows small error, while CounterBraids can achieve no error when the allocated memory is large enough. 2) Elastic stores only flow IDs of elephant flows, while CounterBraids does not store any flow ID. CounterBraids works only when flow IDs of both elephant and mouse flows are known. 3) When knowing all flow IDs, CounterBraids can potentially handle most existing tasks. But the paper of CounterBraids focuses on only frequency estimation. Differently, Elastic handles six typical frequency related tasks. From the above comparison in three aspects, we can see that CounterBraids tries to achieve high accuracy by using elegant mechanisms, while Elastic focuses on handling measurement tasks under different network conditions and achieve as high accuracy as possible. Therefore, it is hard to compare Elastic sketch with CounterBraids, because recording all flow IDs in real high-speed streams is challenging. Besides, the light part *i.e.*, the CM sketch of Elastic can be replaced by CounterBraids. In this way, we can achieve the same function as CounterBraids: decoding all frequencies with no error.

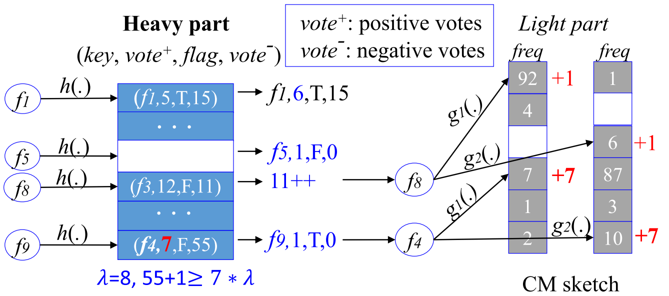


Fig. 1. Basic version of Elastic. To insert f_9, after incrementing $votes^-$, $\frac{vote^-}{vote^+} \geqslant \lambda = 8$, hence f_4 is evicted from the heavy part and inserted into the light part.

UnivMon [3] is the another work to be generic, which can be applied to several tasks. UnivMon is based on a key method named universal streaming [49]. Accuracy is guaranteed thanks to the theory of universal streaming. UnivMon can achieve good performance, however, it does not handle the problem of variable traffic characteristics. To the best of our knowledge, our sketch is the first work that relies on a single data structure which is adaptive to bandwidth, packet rate, and flow size distribution.

III. Elastic Sketches

A. Basic Version

Rationale: As mentioned above, we need to separate elephant flows from mouse flows. We simplify the separation to the following problem: given a high-speed network stream, how to use only one bucket to select the largest flow? As the memory size is too small, it is impossible to achieve the exactly correct result, thus our goal is to achieve high accuracy. Our technique is similar in spirit to Ostracism (Greek: ostrakismos, where any citizen could be voted to be evicted from Athens for ten years). Specifically, each bucket stores three fields: flow ID, positive votes, and negative votes. Given an incoming packet with flow ID f_1, if it is the same as the flow in the bucket, we increment the positive votes. Otherwise, we increment the negative votes, and if $\frac{\#negative\ votes}{\#positive\ votes} \geqslant \lambda$, where λ is a predefined threshold, we expel the flow from the bucket, and insert f_1 into it.

Data structure: As shown in Figure 1, the data structure consists of two parts: a "heavy" part recording elephant flows and a "light" part recording mouse flows. In this paper, we use "elephant flows" to represent flows whose sizes are larger than a threshold T, and "mouse flows" to represent flows whose sizes are no larger than T. The value of T is different in different scenarios, and can be determined by network operators. The heavy part $\mathcal{H}$ is a hash table associated with a hash function $h(.)$. Each bucket of the heavy part records the information of a flow: flow ID (key), positive votes (vote$^+$), negative votes (vote$^-$), and flag. Vote$^+$ records the number of packets belonging to this flow (flow size). Vote$^-$ records the number of other packets. The flag indicates whether the light part may contain positive votes for this flow. The light part is a CM sketch. A CM sketch [17] consists of d arrays ($\mathcal{L}_1$, $\mathcal{L}_2$, …, $\mathcal{L}_d$). Each array is associated with one hash function, and is composed of w counters. Given an incoming packet, the CM sketch extracts the flow ID, computes d hash functions to locate one counter per array, and increments the d counters (we call them d **hashed counters**) by 1. The query is similar to the insertion: after obtaining the d hashed counters, it reports the minimum one.

Insertion:[2] Given an incoming packet with flow ID f, we hash it to the bucket $\mathcal{H}[h(f)\%B]$, where B is the number of buckets in the heavy part. Suppose the bucket stores $(f_1, vote^+, flag_1, vote^-)$. Similar to Ostracism, if f matches f_1, we increment $vote^+$. Otherwise, we increment $vote^-$ and decide whether to evict f_1 according to the two votes. Specifically, there are four cases:

Case 1: The bucket is empty. We insert $(f, 1, F, 0)$ into it, where F means no eviction has happened in the bucket. The insertion ends.

Case 2: $f = f_1$. We just increment $vote^+$ by 1.

Case 3: $f \neq f_1$, and $\frac{vote^-}{vote^+} < \lambda$ after incrementing $vote^-$ by 1 (λ is a predefined threshold, *e.g.*, $\lambda = 8$) We insert $(f, 1)$ into the CM sketch.

Case 4: $f \neq f_1$, and $\frac{vote^-}{vote^+} \geqslant \lambda$ after incrementing $vote^-$ by 1. We "elect" flow f by setting the bucket to $(f, 1, T, 1)$, and evict flow f_1 to the CM sketch: increment the mapped counters by $vote^+$. Note that in this case the flag is set to T (true), because some votes of flow f may be inserted into the light part before f is elected.

Query: For any flow not in the heavy part, the light part (the CM sketch) reports its size. For any flow f in the heavy part, there are two cases: 1) The flag of f is false. Its size is the corresponding vote$^+$ with no error; 2) The flag of f is true. We need to add the corresponding vote$^+$ and the query result of the CM sketch.

The accuracy of Elastic is high in most cases, owing to the separation of elephant flows and mice flows. 1) There is no error in the heavy part: for the flows with flag of false, the recorded vote$^+$ is the flow size with no error; for flows with flag of true, the recorded vote$^+$ is one part of the flow size still with no error, while the other part is recorded in the light with error. 2) In the light part, we do not record the flow ID, and only record the sizes of mice flows, and thus can use many small counters (*e.g.*, 8-bit counters), while traditional sketch needs to use a few large counters (*e.g.*, 32-bit counters) to accommodate the elephant flows. Therefore, our light part can be very accurate. In summary, the accuracy of both elephant and mice flows is high, and we give the formal analysis of Elastic in Section IV.

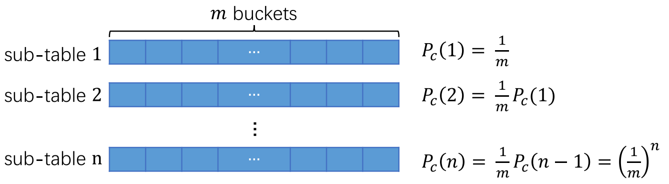
The accuracy of Elastic drops in the worst case – elephant collisions: when two or more elephant flows are mapped into the same bucket, some elephant flows are evicted to the light part and could make some mouse flows significantly overestimated.

Elephant collision rate: defined as the number of buckets mapped by more than one elephant flows divided by the total number of buckets. It is proved that the number of elephant flows that mapped to each bucket follows a Binomial distribution in the literature [50]. We show the following formula of the elephant collision rate P_{hc}.

Theorem 1: Within any bucket in the heavy part of the Elastic sketch, the probability of elephant collisions is

$$P_{hc} = 1 - \left(\frac{H}{w} + 1\right) e^{-\frac{H}{w}} \tag{1}$$

where H is the number of elephant flows, and w is the number of buckets in the heavy part.



m buckets
sub-table 1
sub-table 2
⋮
sub-table n
···
$P_c(1) = \frac{1}{m}$
$P_c(2) = \frac{1}{m} P_c(1)$
$P_c(n) = \frac{1}{m} P_c(n-1) = \left(\frac{1}{m}\right)^n$

Fig. 2. This is the hardware version of Elastic. The elephant collision rate $P_c(.)$ drops exponentially owing to the use of different hash functions for each sub-table.

Proof: There are totally H elephant flows, and each flow is randomly mapped to a certain bucket by the hash function. Given an arbitrary bucket and an arbitrary flow, the probability that the flow is mapped to the bucket is $\frac{1}{w}$. Therefore, for any bucket, the number of elephant flows that mapped to the bucket Z follows a Binomial distribution $B(H, \frac{1}{w})$. When H is large (*e.g.*, $H > 100$), and $\frac{1}{w}$ is a small probability, then Z approximately follows a Poisson distribution $\pi(\frac{H}{w})$, *i.e.*,

$$Pr\{Z = i\} = e^{-\frac{H}{w}} \frac{\left(\frac{H}{w}\right)^i}{i!} \tag{2}$$

There are elephant collisions within one bucket *iff* $Z \geqslant 2$ for this bucket. Therefore, we have

$$\begin{aligned} P_{hc} &= 1 - Pr\{Z = 0\} - Pr\{Z = 1\} \\ &= 1 - \left(\frac{H}{w} + 1\right) e^{-\frac{H}{w}} \end{aligned} \tag{3}$$

□

For example, when $H/w = 0.1$ or 0.01, the elephant collision rate is 0.0046 and 0.00005, respectively.

Solutions for elephant collisions: Obviously, reducing the hash collision rate can reduce the elephant collision rate. Thus, we can use leverage the following two classic methods. 1) **Multiple sub-tables (hardware version):** We can use several sub-tables in the heavy part, and each sub-table is exactly the same as the heavy part of the basic version, but is associated with different hash functions. As shown in Figure 2, the elephant collision rate decreases exponentially as the number of sub-tables increases linearly. As each sub-table has the same operations, this version is suitable for hardware platforms. 2) **Multiple key-value pairs in one bucket (software version):** This allows several elephant flows be recorded in one bucket, and thus the elephant collision rate drops significantly. The differences from the basic version are: 1) All the flows in each bucket share one $vote^-$ field; 2) We always try to evict the smallest flow in the mapped bucket. In this way, the bucket size could be larger than a machine word, thus the accessing of the heavy packet could be the bottleneck. Fortunately, this process can be accelerated by using SIMD on CPU platforms, and thus this version is suitable for software platforms.

B. Adaptivity to Available Bandwidth

To adapt to the available bandwidth, we propose to compress the sketches before sending them. Most flows are mouse flows, thus the memory size of the light part is often much larger than that of the heavy part. In this section, we will show how to compress and merge the light part - CM sketch. In prior work, Michael and *et al.* [51] proposed a method to compress bloom filters. Our work contributes to compressing sketches.

[2] During insertions, we follow one principle: the insertion operations must be one-directional, because it is hard to perform back-tracking operations on hardware platforms.

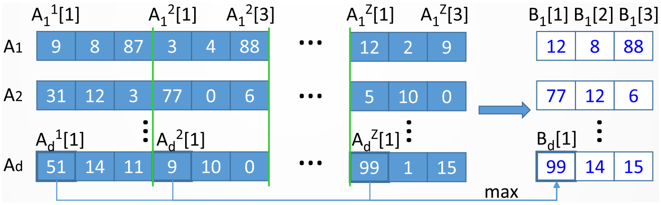


Fig. 3. The Equal Division Compression algorithm.

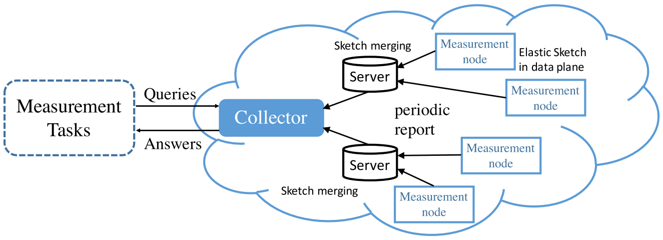


Fig. 4. Network-wide measurements. Servers can be used to merge sketches when the network is large.

1) Compression of Sketches: To compress a sketch, our key idea is first to group the counters, and then merge counters in the same group into one counter.

Grouping: As shown in Figure 3, given a sketch $\mathbb{A}$ of size $zw' \times d$ (width $w = zw'$, depth d, z is an integer representing the compression rate). Our grouping method proceeds as follows: **1)** We split $\mathbb{A}$ into z equal divisions. The size of each division is $w' \times d$. **2)** We build a sketch $\mathbb{B}$ of size $w' \times d$. **3)** Counters with the same index in its division ($\{A_i^k[j]\}_{k=1,\dots,z}$) are in the same group, so we can set $B_i[j] = OP_{k=1}^{z}\{A_i^k[j]\}$ $(1 \leqslant i \leqslant d,\ 1 \leqslant j \leqslant z)$, where OP is the merging operator (*e.g.*, Max or Sum). To query sketch $\mathbb{B}$, we only need to change the hash function $h_i(.)\%w$ to $h_i(.)\%w\%w'$, owing to the following lemma.

Lemma 2: Given an arbitrary integer i, two integers w and w', if w is divisible by w', then $(i\%w)\ \%w' = i\%w'$.

For example, $(10\%6)\%3 = 10\%3$. This lemma will be repeated leveraged in this paper.

Merging: we propose two merging methods. The first method is to sum up the counters in each group, *i.e.*, $B_i[j] = \sum_{k=1}^{z}\{A_i^k[j]\}$. We name this method **Sum Compression (SC)**. As mentioned in Section II, to adapt to available bandwidth, one can build two CM sketches $\mathcal{S}_1$ and $\mathcal{S}_2$ with memory size of M and M/2. A better solution is to compress $\mathcal{S}_1$ to a half. Using SC, the compressed $\mathcal{S}_1$ has the same accuracy as $\mathcal{S}_2$, while SC does not take advantage of the information recorded by $\mathcal{S}_1$. The second method is **Maximum Compression (MC)**. Instead of "sum", we can use "maximum", *i.e.*, $B_i[j] = max\{A_i^1[j], A_i^2[j], \dots, A_i^z[j]\}$. Compared with SC, the sum operation in MC uses more information in $\mathcal{S}_1$, and thus has better accuracy.

About SC and MC, we have the following conclusions: 1) We prove that after SC, the error bound of the compressed CM sketch does not change, while after MC, the error bound is tighter. 2) We prove that using MC, the compressed CM sketch has over-estimation error but no under-estimation error. 3) Our Compression is fast, and our experimental results show that the compressing speed is accelerated by $5 \sim 8$ times after using SIMD (Single Instruction and Multiple Data). 4) There is no need for decompression. 5) Compression does not require any additional data structure. More analysis are provided in Section IV.

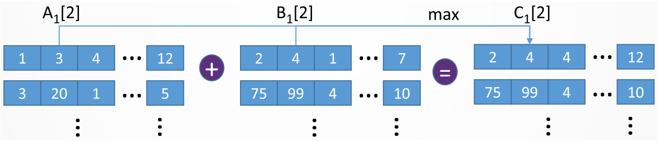


Fig. 5. Maximum merging algorithm.

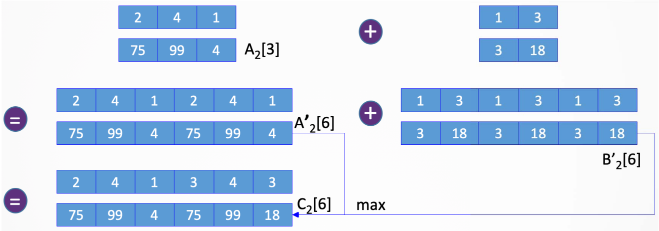


Fig. 6. Maximum Merging algorithm for sketches with different sizes.

2) Merging of Sketches: As shown in Figure 4, one can use servers to save bandwidth. Each server receives many sketches from measurement nodes, merges them, and then sends them to the collector. For the sake of merging, we need to use the same hash functions for all sketches. If they have common flow IDs, we propose to use a naive method – Sum Merging. Otherwise, we propose a novel method, namely Maximum Merging.

Sum Merging: Given two CM sketches of the same size $d \times w$, the Sum merging algorithm just adds the two CM sketches, by adding every two corresponding counters. This algorithm is simple and fast, but not accurate.

Maximum Merging for same-size sketches: Our algorithm is named Maximum Merging (**MM**). As shown in Figure 5, given two sketches $\mathbb{A}$ and $\mathbb{B}$ of size $w \times d$, we build a new sketch $\mathbb{C}$ also of size $w \times d$. We simply set $C_i[j] = max\{A_i[j], B_i[j]\}$ $(1 \leqslant i \leqslant d,\ 1 \leqslant j \leqslant w)$. For example in Figure 5, $C_1[2] = max\{A_1[2], B_1[2]\} = max\{3, 4\} = 4$. This merging method can be easily extended to multiple sketches. Obviously, after MM merging, the sketch still has no under-estimation error.

Maximum Merging for different-size sketches: In real applications, one cannot be sure that two sketches have the same size. When two sketches have different sizes, we propose an algorithm called *least common multiple expansion* (**LCME**). Given a sketch $\mathbb{A}$ with size $w_1 \times d$ and a sketch $\mathbb{B}$ of size $w_2 \times d$. The LCME algorithm proceeds in the following steps (see Figure 6). First, we find the `least common multiple` of w_1 and w_2, suppose it is w_3. Second, we perform a **copy operation**. We copy $\mathbb{A}$ $w_3/w_1 - 1$ times and connect all copies into one. We copy $\mathbb{B}$ $w_3/w_2 - 1$ times and append these copies one by one. Then sketch $\mathbb{A}$ and $\mathbb{B}$ have the same size of $w_3 * d$. The new hash function is $h_i(.)\%w_3$. Third, we merge $\mathbb{A}$ and $\mathbb{B}$ using the above mentioned CMA algorithm.

Example of LCME: As shown in Figure 6, given a sketch $\mathbb{A}$ with size 2×3 and a sketch $\mathbb{B}$ with size 2×2, as the `least common multiple` of 3 and 2 is 6, we copy sketch $\mathbb{A}$ and get a new sketch $\mathbb{A}'$ with size 2×3, and do the same operations on $\mathbb{B}$. Without loss of generality, we discuss $\mathbb{A}$ only. For the counter $A_2[3] = 4$, after expansion, it corresponds to $A'_2[3] = 4$ and $A'_2[6] = 4$. The hash functions before and after expansion are $h(.)\%3$ and $h(.)\%6$, respectively. Then we perform our Maximum Merging algorithm. For example, after merging, $C_2[6] = max\{A'_2[6], B'_2[6]\} = max\{4, 18\} = 18$.

C. Adaptivity to Packet Rate

In measurement nodes, there is often an input queue to buffer incoming packets. The packet rate (*i.e.*, the number of incoming packets per second) is variable: in most cases, it is low, but in the worst case, it is extremely high [35]–[37], [52]. When packet rate is high, the input queue will be filled quickly, and it is difficult to record the information of all packets. To handle this, a recent work, the SketchVisor [19], leverages a dedicated component, namely fast path, to absorb excessive traffic at high packet rate. However, it needs to travel the entire data structure in the worst case, albeit with an amortized $O(1)$ update complexity. This incurs substantial memory accesses and hinders performance. In contrast, our proposed method always needs exactly one memory access.

We propose a new strategy to enhance the insertion speed when needed. When the number of packets in the input queue is larger than a predefined threshold, we let the incoming packets only access the heavy part, so as to record the information of elephant flows only and discard mouse flows. The insertion process of the heavy part is almost unchanged except in the following case: if a flow f in a bucket is replaced by another flow f', the flow size of f' is set to the flow size of f. The reason why setting $f' = f$ is that, we can guarantee that the number of packets recorded in Elastic equals to the real number of inserted packets. Specifically, when a flow f is evicted from the heavy part, in the basic version of Elastic, f is supported to be inserted in the light part. However, when recording flows only in the heavy part, f will not be inserted in the light part. Therefore, setting the frequency of new coming flow to f keeps the number of packets recorded in Elastic unchanged. Therefore, each insertion needs one probe of a bucket in the heavy part. When packet rate goes down, we use our previous algorithms.

D. Adaptivity to Flow Size Distribution

A key metric of the flow size distribution is the number of elephant flows. As it can vary a lot, it is hard to determine the size of the heavy part. To address this issue, we need to make the heavy part adaptive to changes in the traffic distribution. We propose a technique to dynamically double the heavy part. It works as follows. Initially, we assign a small memory size to the heavy part. As more and more elephant flows are inserted, the heavy part will become `full`. We define a threshold T_1. If an incoming packet is mapped into a bucket in which all flows are larger than T_1, we regard the bucket is full. If the number of full buckets exceeds a threshold T_2, we regard the heavy part is full. When the heavy part becomes full, we propose the following **copy operation**: *just copy the heavy part and combine the heavy part with the copy into one*. The hash function is changed from $h(.)\%w$ to $h(.)\%(2w)$. Again, this copy operation works thanks to Lemma 2. After the copy operation, half of the flows in the buckets should be removed. The remove operation can be performed incrementally. For each insertion, we can check all flows in the mapped bucket, and on average half of the flows are not mapped to that bucket and can be removed. Even though some buckets may end up not being cleaned, this does not negatively impact the algorithm.

When using such a strategy to determine when to copy, there could be the following cases. For example, when Γ elephant flows, whose sizes are larger than T_1, are mapped into different buckets, we regard the Elastic sketch is almost full and expand it, but the sketch has plenty of buckets indeed. However, such a case rarely happens. For example, assuming the heavy part has 1000 buckets and $\Gamma = 600$ of them has exactly one flow over T_1, the probability is: $P = \sum_{i=1}^{600} \frac{1000-i+1}{1000} \approx 10^{-102}$. Therefore, the effect of this case can be negligible. The reason for using the size rather than the number of flows per bucket to judge whether a bucket is full is that, in the beginning of the insertion, many mouse flows will be inserted in the heavy part. In this case, we should not execute the copy operation but evict the mouse flows into the light part.

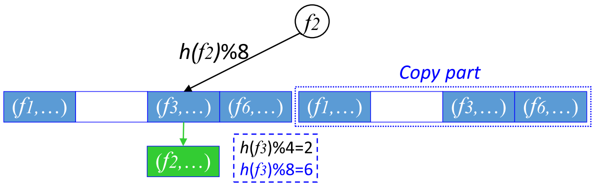


Fig. 7. Duplication of the heavy part of Elastic. The original number of buckets in the heavy part is 4, and becomes 8 after duplication.

Example: As shown in Figure 7, we show how to insert the incoming packet with flow f_2 after duplication. We compute $h(f_2)\%8$ and get the mapped bucket, in which flow f_3 is. We compute $h(f_3)\%8 = 6$ and find that it should be mapped to the bucket in the copy part. Therefore, we replace f_2 by f_3.

Overhead: As the heavy part is often very small (*e.g.*, 150KB), the time overhead of copying an array of 150KB is often small enough to be negligible.

IV. Formal Analysis

A. Performance of Elastic Sketch

Now we derive the error bound of the basic version of Elastic for the flow size estimation task. The adaptivity of Elastic is not considered in this analysis.

Lemma 3: Suppose we use an Elastic sketch (Elastic) and a Count-min sketch (CM) to record a stream at the same time, and the CM sketch and the light part of Elastic have the same width w (the number of counters in each array) and height d (the number of arrays). At any time during recording, if we insert all the flows and the corresponding sizes stored in the heavy part of Elastic into its light part, the light part is exactly the same as CM.

Proof: We give a brief explanation of this Lemma. At any time, the heavy part in Elastic captures some flows in the stream. If we insert the flows from the heavy part into the light part, it is equivalent to reordering the flows in the stream and then inserting them into a CM sketch. According to the insertion of CM sketches, it is obvious that the order of items does not affect the final state of the CM sketch. As a result, Lemma 3 holds. □

Based on Lemma 3, if we use an Elastic Sketch to record a stream, the stream can be seen as two sub-streams recorded by the two parts separately. We then have the following definition.

Definition 1: Let vector $\boldsymbol{f} = (f_1, f_2, \ldots, f_n)$ denote the size vector for a stream, where f_i denotes the size of the i-th flow. Then $\boldsymbol{f}_h$ and $\boldsymbol{f}_l$ denote the size vector of sub-streams recorded by the heavy part and the light part, respectively.

Now we give the error bound of the Elastic on flow size estimation task.

Theorem 4: Given two parameters ϵ and δ, let $w = \lceil \frac{e}{\epsilon} \rceil$ (e is Euler number) and $d = \lceil \ln \frac{1}{\delta} \rceil$. Let an Elastic sketch with *height d and width w record the stream with $\boldsymbol{f}$. The reported flow size $\hat{f}_i$ by the Elastic for flow i is bounded by*

$$\hat{f}_i \leqslant f_i + \epsilon\|\boldsymbol{f}_l\|_1^{3} < f_i + \epsilon\|\boldsymbol{f}\|_1 \tag{4}$$

with probability at least $1 - \delta$.

Proof: According to the query procedure of the Elastic sketch, the reported value $\hat{f}_i$ can be expressed as $\hat{f}_i \leqslant \widehat{f_{ih}} + \widehat{f_{il}}$, where $\widehat{f_{ih}}$ and $\widehat{f_{il}}$ denote the estimated value of the heavy part and the light part, respectively. Notice that we use the "$\leqslant$" operator instead of "$=$", because if the flag in the heavy part is not set, the light part will not be queried. According to Definition 1, $\widehat{f_{ih}}$ and $\widehat{f_{il}}$ are the estimated values of f_{ih} and f_{il}, where f_{ih} and f_{il} is the i-th value in $\mathbf{f}_h$ and $\mathbf{f}_l$, respectively. Since the heavy part counts each flow exactly, we have $\widehat{f_{ih}} = f_{ih}$. Since the light part of the Elastic sketch is a Count-Min sketch for $\mathbf{f}_l$, according to [17], we have $\widehat{f_{il}} \leqslant \epsilon\|\mathbf{f}_l\|_1$ with probability at least $1 - \delta$. Then, we have

$$\hat{f}_i \leqslant \widehat{f_{ih}} + \widehat{f_{il}} \leqslant f_i + \epsilon\|\mathbf{f}_l\|_1 \tag{5}$$

with probability at least $1 - \delta$. Since $\mathbf{f}_l$ represents the stream stored in the light part, $\|\mathbf{f}_l\|_1 < \|\mathbf{f}\|_1$. Therefore, Theorem 4 holds. □

According to Theorem 4, the estimation error of Elastic is bounded by $\|\mathbf{f}_l\|_1$, instead of $\|\mathbf{f}\|_1$ in Count-Min. Notice that $\|\mathbf{f}_l\|_1$ represents the number of packets recored in the light part, and $\|\mathbf{f}\|_1$ represents the total number of packets in the stream. In practice, often, most packets of a stream are recorded in the heavy part, and thus $\|\mathbf{f}_l\|_1$ is usually significantly smaller than $\|\mathbf{f}\|_1$. Thus Elastic has *a much tighter error bound* than Count-Min when the parameters (d and w) are the same. Although the Elastic sketch needs an additional heavy part compared to CM, it has a smaller light part because we do not need large counters to accommodate the largest flow.

Another observation from Theorem 4 is that, the error bound of Elastic sketch is determined by the light part, so its adaptivity to flow size distribution does not affect this error bound. In the next part, we will discuss how the adaptivity to available bandwidth affects the error bound.

B. Performance of Compression

1) Error Bound of the Compressed CM Sketch Using Sum Compression: Here we prove that the error bound does not change after using the Sum Compression algorithm.

Theorem 5: Assume that a CM sketch with d arrays and zw counters per array, and z is the SC compression rate of the sketch. Given an arbitrary small positive number ϵ, the error of any flow after compression is bounded by

$$Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} \leqslant \left(\frac{1}{\epsilon w}\right)^d \tag{6}$$

Clearly that the error bound is identical to that of the CM sketch not using the Sum Compression algorithm.

Proof: We first focus on one array of the CM sketch. Let flow f_j mapped to counter C_1, and after compression, C_1 is compressed into a new counter with other $z - 1$ counters $C_2, C_3 \cdots C_z$. Let X_i be the number of packets that mapped to counter C_i before compression (except for packets of flow f_j). Therefore, after compression, the value in the new counter is $X_1 + X_2 + \cdots + X_z$. Then the estimated flow size of flow f_j in this array is

$$\hat{n}_j^1 = n_j + \sum_{i=1}^{z} X_i \tag{7}$$

According to the Markov inequality, given a positive number ϵ, we have

$$\begin{aligned} Pr\{\hat{n}_j^1 \geqslant n_j + \epsilon N\} &= Pr\{\hat{n}_j^1 - n_j \geqslant \epsilon N\} \\ &\leqslant \frac{E(\hat{n}_j^1 - n_j)}{\epsilon N} = \frac{E(\sum_{i=1}^{z} X_i)}{\epsilon N} \\ &= \frac{\sum_{i=1}^{z} E(X_i)}{\epsilon N} = \frac{z \cdot \frac{N}{zw}}{\epsilon N} = \frac{1}{\epsilon w} \end{aligned} \tag{8}$$

Because the estimated flow size of f_j is the minimum of the estimated flow size in each array, *i.e.*, $\hat{n}_j = min\{\hat{n}_j^1\, \hat{n}_j^2 \cdots \hat{n}_j^d\}$, we have

$$Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} = Pr\{\hat{n}_j^1 \geqslant n_j + \epsilon N\}^d \leqslant \left(\frac{1}{\epsilon w}\right)^d \tag{9}$$

□

2) Error Bound of the Compressed CM Sketch Using Maximum Compression: Consider a Count-min sketch with d arrays and w counters per array. According to the literature [17], we can easily get the error bound of the CM sketch

$$Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} \leqslant \left(\frac{1}{\epsilon w}\right)^d \tag{10}$$

where ϵ is a given positive number, n_j is the real size of flow f_j and $\hat{n}_j$ is the estimated size of f_j.

Next, we consider the CM sketch using our compression technique.

Theorem 6: Assume that a CM sketch with d arrays and zw counters per array, and z is the compression rate of the sketch. Given an arbitrary small positive number ϵ and an arbitrary flow f_j, the error of the sketch after our compression is bounded by

$$\begin{aligned} &Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} \\ &\quad \leqslant \left\{1 - \left(1 - \frac{1}{\epsilon z w}\right)\left[1 - \frac{N}{zw(n_j + \epsilon N)}\right]^{z-1}\right\}^d \end{aligned} \tag{11}$$

Proof: After compression, each counter in the new sketch will be the maximum value of z counters in the original sketch. Because each array is independent of each other, we first only focus on the first array. For a certain flow f_j, it is mapped to one counter, and the counter is in the same *compression group* with other $z - 1$ counters. For convenience, we use $V_1, V_2, \cdots, V_z$ to denote the number of packets mapped to the z counters, excluding packets from flow f_j. Without loss of generality, we assume that flow f_j is mapped to the first counter in the compression group [53], [54]. In this way, the estimated size of f_j in the first array ($\hat{n}_j^1$) is

$$\hat{n}_j^1 = max(V_1 + n_j, V_2, V_3, \cdots, V_z) \tag{12}$$

And we have

$$\begin{aligned} &Pr\{\hat{n}_j^1 \geqslant n_j + \epsilon N\} \\ &\quad = Pr\{max(V_1 + n_j, V_2, V_3, \cdots, V_z) \geqslant n_j + \epsilon N\} \\ &\quad = 1 - Pr\{max(V_1 + n_j, V_2, V_3, \cdots, V_z) < n_j + \epsilon N\} \\ &\quad = 1 - Pr\{V_1 + n_j < n_j + \epsilon N\} \prod_{i=2}^{z} Pr\{V_i < n_j + \epsilon N\} \end{aligned} \tag{13}$$

[3] $\|\mathbf{x}\|_1$ is the first moment of vector $\mathbf{x}$, i.e., $\|\mathbf{x}\|_1 = \sum x_i$.

According to Markov inequality, it is easy to derive that

$$\begin{aligned} Pr\{V_1 + n_j < n_j + \epsilon N\} &= Pr\{V_1 < \epsilon N\} \\ &\geqslant 1 - \frac{E(V_1)}{\epsilon N} = 1 - \frac{1}{\epsilon z w} \end{aligned} \tag{14}$$

Note that $E(V_1) = \frac{N}{zw}$. For $i = 2, 3, \cdots, z$, we have

$$\begin{aligned} Pr\{V_i < n_j + \epsilon N\} &\geqslant 1 - \frac{E(V_i)}{n_j + \epsilon N} \\ &= 1 - \frac{N}{zw(n_j + \epsilon N)} \end{aligned} \tag{15}$$

Therefore, we have

$$\begin{aligned} &Pr\{\hat{n}_j^1 \geqslant n_j + \epsilon N\} \\ &= 1 - Pr\{V_1 + n_j < n_j + \epsilon N\} \prod_{i=2}^{z} Pr\{V_i < n_j + \epsilon N\} \\ &\leqslant 1 - \left(1 - \frac{1}{\epsilon z w}\right) \prod_{i=2}^{z} \left[1 - \frac{N}{zw(n_j + \epsilon N)}\right] \\ &= 1 - \left(1 - \frac{1}{\epsilon z w}\right) \left[1 - \frac{N}{zw(n_j + \epsilon N)}\right]^{z-1} \end{aligned} \tag{16}$$

Now we focus on the error bound for d arrays. Note that the estimated size of flow f_j is minimum value of d mapped counters, we have

$$\hat{n}_j = min(\hat{n}_j^1, \hat{n}_j^2, \cdots, \hat{n}_j^d) \tag{17}$$

where $\hat{n}_j^i$ denotes the estimated size of f_j in the i^{th} array. Therefore, we have

$$\begin{aligned} &Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} \\ &= Pr\{min(\hat{n}_j^1, \hat{n}_j^2 \cdots \hat{n}_j^d) \geqslant n_j + \epsilon N\} \\ &= \prod_{i=1}^{d} Pr\{\hat{n}_j^i \geqslant n_j + \epsilon N\} \\ &\leqslant \prod_{i=1}^{d} \left\{1 - \left(1 - \frac{1}{\epsilon z w}\right) \left[1 - \frac{N}{zw(n_j + \epsilon N)}\right]^{x-1}\right\} \\ &= \left\{1 - \left(1 - \frac{1}{\epsilon z w}\right) \left[1 - \frac{N}{zw(n_j + \epsilon N)}\right]^{z-1}\right\}^d \end{aligned} \tag{18}$$

□

3) Error Bound Comparison: In this part, We compare the error bound of the CM sketch and the compressed CM sketch by our maximum compression.

Theorem 7: Given a compressed CM sketch using our maximum compression, assume it has the same d and w with another standard CM sketch. The compressed CM sketch has a smaller error bound than the standard CM sketch.

Proof: For convenience, we use P_{CM} to denote the error bound of standard CM sketches, and use P_{MC} to denote the error bound of the compressed CM sketch using our maximum compression algorithm. First, we have

$$\begin{aligned} P_{MC} &= \left\{1 - \left(1 - \frac{1}{\epsilon z w}\right) \left[1 - \frac{N}{zw(n_j + \epsilon N)}\right]^{z-1}\right\}^d \\ &< \left[1 - \left(1 - \frac{1}{\epsilon z w}\right) \left(1 - \frac{N}{zw\epsilon N}\right)^{z-1}\right]^d \\ &= \left[1 - \left(1 - \frac{1}{\epsilon z w}\right)^z\right]^d \end{aligned} \tag{19}$$

For convenience, we let $x = \frac{1}{\epsilon w}$. Then we define a function $F(z)$ as

$$F(z) = \left(1 - \frac{x}{z}\right)^z \tag{20}$$

where z is a positive integer, and x is a number in $[0, 1]$. According to the inequality of arithmetic and geometric means, we have

$$\begin{aligned} F(z) &= \left(1 - \frac{x}{z}\right)^z = 1 \cdot \left(1 - \frac{x}{z}\right)^z \\ &\leqslant \left[\frac{1 + z(1 - \frac{x}{z})}{z+1}\right]^{z+1} = \left(1 - \frac{x}{z+1}\right)^{z+1} = F(z+1) \end{aligned} \tag{21}$$

Therefore, $F(z)$ is a monotonic increasing function, and we have

$$\begin{aligned} P_{MC} < \left[1 - \left(1 - \frac{1}{\epsilon z w}\right)^z\right]^d &= [1 - F(z)]^d \leqslant [1 - F(1)]^d \\ &= P_{CM} \end{aligned} \tag{22}$$

Therefore, the compressed sketch has a smaller error bound than the standard CM sketch. □

According to Theorem 7, we can conclude that if we build a large sketch S_1 for recording traffic and compress it to a smaller sketch S_2 for adaptive to the bandwidth, the error bound of S_2 will always no worse than if we use S_2 to record the traffic directly.

4) Error Bound of the Merged Sketch: Given z sketches with different size $d \times w_i (1 \leqslant d \leqslant z)$, consider merging them into one sketch using our LCME merging algorithm. Then the size of the merged sketch should be $d \times w$, where w is the least common multiple of $w_1, w_2, \ldots, w_z$. In this part, we give the error bound of the merged sketch after using Sum Merging or Maximum Merging.

Suppose the i-th sketch records N_i packets, and let $N = \sum_{i=1}^{z} N_i$. Then we have the following results.

Theorem 8: Suppose the Sum Merging is used to get the merged sketch. Given an arbitrary small positive number ϵ, the error of any flow after merging is bounded by

$$Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} \leqslant \left(\frac{\sum_{i=1}^{z} \frac{N_i}{w_i}}{\epsilon N}\right)^d \tag{23}$$

Theorem 9: Suppose the Maximum Merging is used to get the merged sketch. Without loss of generality, suppose the queried flow f_j is recorded by the first sketch. Given an arbitrary small positive number ϵ and the queried flow f_j, the error of the sketch after merging is bounded by

$$\begin{aligned} &Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} \\ &\leqslant \left\{1 - \left(1 - \frac{N_1}{\epsilon N w_1}\right) \Pi_{i=2}^{z} \left[1 - \frac{N_i}{w_i(n_j + \epsilon N)}\right]\right\} \end{aligned} \tag{24}$$

The proofs of Theorem 8 and 9 are similar to those of Theorem 5 and 6, so we omit the detailed proofs here.

5) Proof of No Under-Estimation Error:

Theorem 10: After using the MC algorithm, the CM sketch still has only over-estimation error but no under-estimation error.

Proof: Without loss of generality, we only focus on one array (denoted as $A[]$) in the CM sketch. For any flow f_i and its mapped counter in the array, the value in the counter $A[g(f_i)]$ should be not smaller than n_i, *i.e.*, $A[g(f_i)\%w] \geqslant n_i$. Let the array after using compression algorithm be denoted as $A'[]$. After using compression algorithm with compression ratio z, then flow f_i is mapped to counter $A'[g(f_i)\%(w/z)]$, and the value in the counter is the maximum value of the original z values. Therefore, we have

$$A[g(f_i)\%w] \leqslant A'[g(f_i)\%(w/z)] \tag{25}$$

In this way, we have $n_i \leqslant A'[g(f_i)\%(w/z)]$. Therefore, for any flow, its estimated value in one array is not smaller than its real flow size. □

V. Applications

Flow Size Estimation: Our Elastic can be directly used to estimate flow size in packets. Our sketch has a unique characteristic: for those flows that have a flag of false, our estimation has no error. According to our experimental results, we find that more than 56.6% flows in the heavy part have no error when using 600KB memory for 2.5M packets.

Heavy Hitter Detection: For this task, we query the size of each flow in the heavy part. If one's size is larger than the predefined threshold, then we report this flow as a heavy hitter. We can achieve very high accuracy of detecting heavy hitter, because we record flow IDs in the heavy part, only a very small part of flows those are exchanged from the light part could have error.

Heavy Change Detection: For two adjacent time windows, we build two Elastic sketches, respectively. To find heavy changes with threshold T, one common used method is to check all flows in each time window with size no less than T. Therefore, we only check flows in the heavy parts of the two sketches. There are two types of flows that we check: 1) flows stored in the heavy parts of both sketches; 2) flows stored in the heavy part of one sketch, but not stored in the heavy part of the other sketch. If the size difference of a flow in the two windows is larger than T, we report it as a heavy change.

Cardinality Estimation: We first count the number of distinct flows in the heavy part. Then we calculate the number of distinct flows in the light part using the method of linear counting [55]. The cardinality is the sum of the two numbers.

Estimation of Flow Size Distribution and Entropy: These three tasks care about both the elephant flows and mouse flows. For flows in the heavy part, we can get their information directly. For flows in the light part, we can get the needed information from the counter distribution. So at the end of each time window, we collect the counter distribution array $(n_0, n_1, \ldots, n_{255})$ of the light part, where n_i is the number of counters whose value is i. Then we send this array together with the heavy part and the compressed light part to the collector. We estimate the distribution of the light part using the basic version of the MRAC algorithm [24] which is an Expectation-Maximization (EM) algorithm. Given a counter, there are several ways to forming this counter by flows. For example, a counter with 3 can be formed by one flow (of size 3) or two flows (of size 1 and 2). The key idea of MRAC is that, given a distribution, the probability of different ways to form a counter can be derived, and given the probability, we can get an expectation of the distribution. MRAC iterates this process until the convergence condition is satisfied. Since MRAC needs to enumerate different ways to make up a counter, its time complexity is related to the maximum size of a counter. Since the maximum size of a counter of the light part is small, it is fast to run MRAC in our Elastic sketch.

After estimating the distribution of the light part, we sum it with the distribution of the heavy part as the estimated the distribution. Then we compute the entropy based on the flow size distribution as $-\sum(i * \frac{n_i}{m} \log \frac{n_i}{m})$, where m is the sum of n_i, and n_i is the number of flows with size of i.

VI. Implementations

In this section, we briefly describe the implementation of hardware and software versions of the Elastic sketch on P4, FPGA, GPU platforms, and CPU, multi-core CPU, OVS platforms, respectively.

A. P4 Implementation

We have fully built a P4 prototype of the Elastic sketch on top of a baseline switch.p4 [42] and compiled on a programmable switch ASIC [56]. We add 500 lines of P4 code that implements all the registers and meta-data needed for managing the Elastic sketch in the data plane.

We implement both heavy part and light part of the hardware version in registers instead of match-action tables because those parts require updating the entries directly from the data plane. We leverage the Stateful Algorithm and Logical Unit (Stateful ALU) in each stage to lookup and update the entries in register array. However, Stateful ALU has its resource limitation: each Stateful ALU can only update a pair of up to 32-bit registers while our hardware version of Elastic needs to access four fields in a bucket for an insertion. To address this issue, we tailor our Elastic sketch implementation for running in P4 switch at line-rate but with a small accuracy drop.

The P4 version of the Elastic sketch: It is based on the hardware version of the Elastic sketch, and we only show the differences below. 1) We only store three fields in two physical stages: vote^{all}, and (key, vote^{+}), where vote^{all} refers to the sum of positive votes and negative votes. 2) When $\frac{vote^{all}}{vote^{+}} \geqslant \lambda'$, we perform an eviction operation. 3) When a flow $(f, vote^{+})$ is evicted by another flow $(f_1, vote_1^{+})$, we set the bucket to $(f_1, vote^{+} + vote_1^{+})$. We recommend using 4 subtables in the P4 version. In this way, we only need $4 \times 2 = 8$ stages for the heavy part, and 1 stage for the light part, and thus in total 9 stages. Note, we are not using additional stages for Elastic. Instead, incoming packets go through the Elastic sketch and other data plane forwarding tables in parallel in the multi-stage pipeline. Table I shows the additional resources that the Elastic sketch needs on top of the baseline switch.p4 mentioned before. We can see that additional resource use is less than 6% across all resources, except for SRAM and stateful ALUs. We need to use SRAM to store the Elastic sketch and stateful ALUs to perform transactional read-test-write operations on the Elastic sketch. Note, adding additional logics into ASIC pipeline does

TABLE I

Additional H/W Resources Used by Elastic Sketch, Normalized by the Usage of the Baseline Switch.P4

Resource	Baseline	Additional usage
Match Crossbar	474	5.9%
SRAM	288	12.5%
TCAM	102	0%
VLIW Actions	145	5.5%
Hash Bits	1605	2.3%
Stateful ALUs	4	75%
Packet Header Vector	277	0.36%

not really affect the ASIC processing throughput as long as it can fit into the ASIC resource constraint. As a result, we can fit the Elastic sketch into switch ASIC for packet processing at line-rate.

Note that in this version, we do not use flags, and during an eviction, there will be errors because the size of the incoming flow is set to the sum of the two flow sizes, so there is a small accuracy drop compared with the software and hardware versions. Larger value of λ' leads to fewer number of evictions, and higher accuracy. According to our experimental results on different datasets, we find the accuracy increases when varying λ' from 4 to 32, but when varying λ' from 32 to 128, the accuracy increases little. Therefore, we choose $\lambda' = 32$, while $*32$ can be achieved by $\ll 5$.

We have not implemented the aforementioned three adaptivities in the data plane for the following reasons. To be adaptive to available bandwidth, it is hard to implement the compression step in the data plane. Besides, since sketches should be read into the control plane before sending, the control plane is a better place for the compression. To be adaptive to packet rate, we don't need to implement it, because the throughput capacity of P4 switches is always constant, i.e., irrelevant to the number of memory accesses for processing each packet. To be adaptive to flow size distribution, it cannot be implemented in the data plane because it needs to change the length of *register arrays*, which is not supported by the current P4 switches (Tofino), and is probably supported in the near future.

B. FPGA Implementation

In the FPGA implementation, two operations affect the clock frequency: "%" and "×". To improve efficiency, we make two minor modifications. First, we let the two hash functions for the heavy and light part avoid "%" operations. We set the size of the heavy part and the light part to be 2^{12} and 2^{19} respectively, and the total memory usage is 0.69MB. In this way, the "%" can be replaced by "&". For example, $100\%16 = 100\&15$. Second, we find that accuracy and speed barely change when λ varies from $4 \sim 64$. Therefore, we choose 8 and the operation $*8$ is replaced by $\ll 3$.

We implement the Elastic sketch on an FPGA platform. We use the Stratix V family of Altera FPGA (model 5SEEBF45I2). The capacity of the on-chip RAMs (Block RAM) of this FPGA is 54,067,200 bits. The resource usage information is as follows: 1) We use 1,978,368 bits of Block RAM, 4% of the total on-chip RAM. 2) We use 36/840 pins, 4% of the total 840 pins. 3) We use 2939 logics, less than %1 of the 359,200 total available. After using the pipeline, our implementation can process each packet in one clock cycle. The clock frequency of our implemented FPGA is 162.6 MHz, which means that *it can achieve a processing speed of 162.6 Mpps*. We also release the Verilog code of the FPGA implementation [57].

C. GPU Implementation

We use the CUDA toolkit to write programs on GPU to accelerate the insertion of the Elastic sketch. Two techniques, batch processing and multi-streaming, are applied to achieve the acceleration. GPU has a large number of threads that can perform tasks concurrently. Therefore, instead of inserting keys one by one, we first copy a batch of keys to be inserted from the CPU to the GPU, and then utilize many threads to insert those keys concurrently into the data structure stored on GPU. This process is called batch processing. Furthermore, although a batch of keys must be copied from the CPU to the GPU before it can be inserted, it is possible that a previous batch of keys is being inserted while a new batch of keys is being copied. This technique is called multi-streaming. The CUDA toolkit provides convenient functions to distinguish different data streams.

Specifically, we let each thread process one packet. Leveraging the atomic add operation in CUDA, the insertion of the light part can be implemented directly. Implementing the insertion of the heavy part is more challenging, because different threads more need modify the same bucket, and this may incur some problems. For example, two threads may find that the negative votes is enough and try to evict a flow at the same time. The atomic operations supported by CUDA cannot solve this problem directly, so we implement the lock mechanism by leveraging `atomicCAS` and `atomicExch`. We create an array of locks, and each lock is corresponding to one bucket. When a thread try to insert a packet into a bucket, it will try to acquire the lock of this bucket first. To avoid starvation, we set the maximum number of attempts to acquire the lock of a bucket, *e.g.*10. This means if a thread have tried 10 times for inserting the packet into a bucket, it will give up trying and insert this packet into the light part.

D. CPU Implementation Using SIMD

We use SIMD (Single Instruction Multiple Data) instructions to accelerate the processing speed. With the AVX2 instruction set, we can compare 8 32-bit integers with another set of 8 32-bit integers in a single instruction. We use this to accelerate the key comparisons. Also, we can use SIMD instructions to find the minimum counter among 8 counters, as well as its corresponding index in a single comparison instruction. To leverage the power of SIMD instructions, we have to make both the keys and counters stored sequentially in a bucket. If one bucket contains 8 cells, we put the 8 keys together, and then store the 8 counters. AVX2 instructions ask for the addresses of items to be aligned on 32 bytes. In our algorithm, each bucket with 8 cells occupies exactly 64 bytes, so we align the bucket on 64 bytes. Since the cache line size is 64 bytes, this alignment is also friendly for the cache, which means many insertions only need to access one cache line.

For longer flow IDs (*e.g.*, when the length of the 5-tuple is 13 bytes), it is hard to use SIMD instructions to match the key. To address this, we calculate 32-bit fingerprints for long flow IDs, use the fingerprints as keys, and allocate another memory space to store the flow IDs. During the insertion, we can match the fingerprint only to reduce memory accesses. This will incur false positives because of the collisions. Since there are way

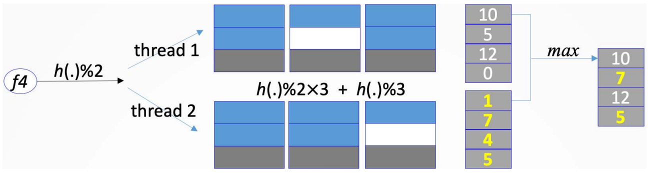


Fig. 8. Merging of small Elastic sketches on multi-core CPU.

fewer flows in a time interval than different items a 32-bit fingerprint can represent, the collision rate is small, hence the false positives can be ignored.

E. Multi-Core Implementation With Merging

Multi-core CPUs are popular nowadays, so we tailor the elastic sketch implementation to multi-core. If we dispatch packets to different cores, each core will record only a subset of packets, and thus the overall performance will be significantly improved. The problem is how to merge the results recorded by different cores. Fortunately, for Elastic sketches, we can leverage the compression techniques again. For example, as shown in Figure 8, we use two threads. In each thread, we deploy an Elastic with 3 buckets, and each of which has 3 cells in the heavy part, and 4 counters in the light part. To insert flow f_4, we first use the value $h(.)\%2$ to determine a thread, *e.g.*, thread 1, to use. Then, in this thread, we use the value $h(.)\%3$ to determine a bucket in the heavy part to insert. After merging of small Elastic on multi-core, we concatenate all heavy parts into one bucket array (the merged heavy part). Therefore, we can determine the bucket in the merged heavy part by the value $(h(.)\%2)\times 3$. At the end of each time window, we combine all small elastic sketches into one. The heavy parts are easy to combine: we combine all the heavy parts one by one, and only need to change the hash function. For example, in Figure 8, after merging, the hash function becomes $h(.)\%2 \times 3 + h(.)\%3$. For all light parts, we merge them into one using the merging algorithms, by choosing the maximum of the corresponding counters. As shown in Figure 8, the first counters of the two light parts are 10 and 1, and the first counter after merging is $max\{10, 1\} = 10$.

F. OVS Implementation

We implement a prototype software switch. Software switches have been important building blocks of virtualization software in modern public and private clouds. Our implementation is based on Open vSwitch (OVS) [58], one of the most widely deployed software switches. Particularly, we target the DPDK version of OVS. The DPDK version realizes its data plane entirely in user space. The user-space data plane directly accesses NIC buffers, hence it completely eliminates the overhead due to memory copies and context switching between kernel and user space.

Integrating sketches into OVS will introduce extra overhead for packet processing, because each packet should be inserted to the sketch. To reduce this overhead, we run the Elastic instance as an individual process, and let it communicate with OVS via a shared-memory based ring buffer. The data plane of OVS is responsible to intercept packets, parse their headers, and write headers into the ring buffer. The Elastic process works as a consumer which reads the ring buffer continuously. Since OVS with DPDK supports multi-thread processing to improve the throughput, we can create multiple ring buffers, and let the Elastic process read these ring buffers by round-robin.

TABLE II
CAIDA TRACES USED IN THE EVALUATION

Trace	Date	# packets	# flows (SrcIP)
CAIDA1	2015/02/19	1164.9M	2.6M
CAIDA2	2015/05/21	1081.0M	3.9M
CAIDA3	2016/01/21	1835.1M	8.9M
CAIDA4	2016/02/18	1799.7M	8.4M

VII. EXPERIMENTAL RESULTS

A. Experimental Setup

Traces: We use four one-hour public traffic traces collected in Equinix-Chicago monitor from CAIDA [59]. The details of these traces are shown in Table II. We divide each trace into different time intervals (1s, 5s, 10s, 30s, and 60s). For example, each one-hour trace contains 720 5s-long sub-traces, and we plot 10^{th} and 90^{th} percentile error bars across these 720 sub-traces. We use the CAIDA4 trace with a monitoring time interval of 5s as default trace, which contains 1.1M to 2.8M packets with 60K to 110K flows (SrcIP). Due to space limitations, we only show the results with the source IP as the flow ID; the results are qualitatively similar for other flow IDs (*e.g.*, destination IP, 5-tuple).

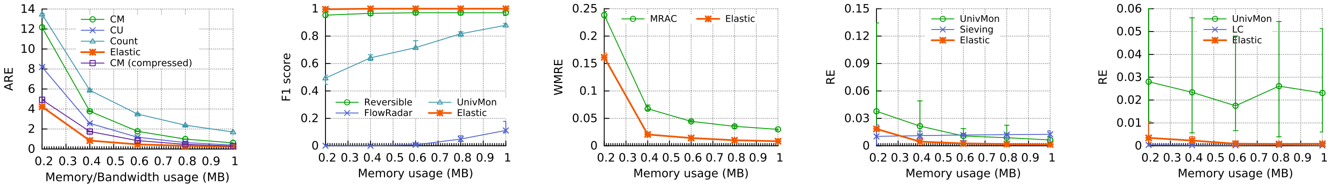
Evaluation metrics:

- *ARE (Average Relative Error):* $\frac{1}{n}\sum_{i=1}^{n}\frac{|f_i-\widehat{f_i}|}{f_i}$, where n is the number of flows, and f_i and $\widehat{f_i}$ are the actual and estimated flow sizes respectively.
- F_1 *score:* $\frac{2\times PR\times RR}{PR+RR}$, where PR (Precision Rate) refers to the ratio of true instances reported and RR (Recall Rate) refers to the ratio of reported true instances.
- *WMRE (Weighted Mean Relative Error) [19], [24]:* $\frac{\sum_{i=1}^{z}|n_i-\widehat{n}_i|}{\sum_{i=1}^{z}(\frac{n_i+\widehat{n}_i}{2})}$, where z is the maximum flow size, and n_i and $\widehat{n}_i$ are the true and estimated numbers of flows of size i respectively.
- *RE (Relative Error):* $\frac{|True-Estimated|}{True}$, where $True$ and $Estimate$ are the true and estimated values, respectively.
- *Throughput:* million packets per second (Mpps).

Setup: When comparing with other algorithms, we use the **software version** of Elastic. Specifically, we store 7 flows and a vote$^-$ for each bucket in the heavy part, and use one hash function and 8-bit counters in the light part. For each algorithm in each task, the default memory size is 600KB. The heavy part does not dynamically resize except for the experiments of adaptivity to traffic distribution (Section VII-C). Detailed configurations for each task are as follows:

- *Flow size estimation:* We compare four approaches: CM [17],[4] CU [20], Count [21], and Elastic. For CM, CU, and Count, we use 3 hash functions as recommended in [62].
- *Heavy hitter detection:* We compare six approaches: Space-Saving (SS) [22], Count/CM sketch [17], [21] with a min-heap (CountHeap/CMHeap), UnivMon [3], HashPipe [23] and Elastic. For CountHeap/CMHeap, we use 3 hash functions and set the heap capacity to 4096. For UninMon, we use 14 levels and each level records 1000 heavy hitters. We set the HH threshold to 0.02% of the number of packets in one measurement epoch.

[4] A CM sketch can be seen as a variant of Bloom filters [44], [60], [61]



(a) Flow size. (b) Heavy change. (c) Flow size distribution. (d) Entropy. (e) Cardinality.

Fig. 9. Accuracy comparison for five tasks. The heavy part in Elastic is 150KB.

- *Heavy change detection:* We compare Reversible sketch [63], FlowRadar [26], UnivMon, and Elastic. For Reversible, we use 4 hash functions as recommended in [63]. For FlowRadar, we use 3 hash functions in both the Bloom filter [64] and the IBLT part [65]; we allocate $1/10$ of the memory for the Bloom filter and the rest for IBLT. UnivMon uses the same setting as before. We set the HC threshold as 0.05% of total changes over two adjacent measurement epochs.
- *Flow size distribution:* We compare MRAC [24] and Elastic.
- *Entropy estimation:* We compare UnivMon, Sieving [66], and Elastic. UnivMon uses the same setting as before. We use 8 sampling groups in Sieving.
- *Cardinality estimation:* We compare UnivMon, linear counting (LC) [55], and Elastic. UnivMon uses the same setting as before.

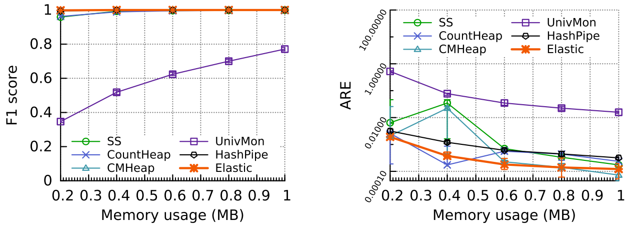
B. Accuracy

Figure 9(a)-(e) and 10(a)-(b) provide a comparison of the accuracy of different algorithms for six tasks. Note that Elastic only uses one data structure with memory of 600KB to handle all six tasks.

Flow size estimation (Figure 9(a)): We find that Elastic offers a better accuracy vs. memory usage trade-off than CM, CU, and Count sketch. When using 600KB of memory, the ARE of Elastic is about 3.8, 2.5, and 7.5 times lower than the one of CM, CU, and Count. We also run the maximum compression algorithm (§III-B.1) on a CM sketch with initial 16MB memory, and measure its ARE when its memory after compression (*i.e.*, bandwidth) reaches $0.2, 0.4, \ldots, 1$ MB, respectively. We find that our compression algorithm significantly improves the accuracy of CM sketch. The compressed CM sketch has better accuracy than a standard CM sketch with the same initial size.

We use an example to illustrate the improvement on the accuracy. Assuming we have two CM sketches, C_0 of size w and C_1 of size $2w$. After recording the same traffic into C_0 and C_1, we use MC to compress C_1 into the compressed CM sketch C_2 of size w. According to the mechanism of MC, every counter in C_2 is smaller than the corresponding counter in C_0, *i.e.*, for any $i \in [1, w]$, $C_2[i] \leqslant C_0[i]$. Further, it is easy to prove that the compressed CM sketch has only overestimate error but no underestimate error. Therefore, C_2 is more accurate than C_0.

Heavy hitter detection (Figure 10(a)-(b)): We find that Elastic is much more accurate than the other five algorithms for most memory sizes. Even with less than 200KB of memory, Elastic is able to achieve 100% precision and recall with only 0.002 ARE, an ARE much lower than the other five algorithms.



(a) F1 score. (b) ARE.

Fig. 10. Accuracy comparison for heavy hitter detection. The heavy part in Elastic is 150KB.

Heavy change detection (Figure 9(b)): We find that Elastic always achieves above 99.5% F1 score while the best F1 score from the other algorithms is 97%. When using more than 200KB of memory, the precision and recall rates of Elastic both reach 100%. When using little memory, FlowRadar can only partially decode the recorded flow IDs and frequencies, causing a low F1 score.

Flow size distribution (Figure 9(c)): We find that Elastic always achieves better accuracy than the state-of-the-art algorithm (MRAC). When using 600KB of memory, the WMRE of Elastic is about 3.4 times lower than the one of MRAC. The reason is that the accuracy of MRAC increases as the number of counters increases. Since we use small (8-bit) counters in the light part of Elastic, compared with the large counter (32-bit) in MRAC, given the same amount of memory, our light part can have 3X more counters than MRAC. In our experiments, we allocate 150KB memory for the heavy part and 450KB memory for the light part, so Elastic has twice more counters than MRAC. As a result, Elastic achieves higher accuracy than MRAC.

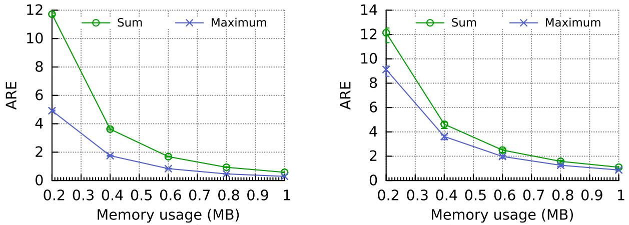
Entropy estimation (Figure 9(d)): We find that Elastic offers a better estimation than the other two algorithms for most memory sizes. When using a memory larger than or equal to 400KB, Elastic achieves higher accuracy than both state-of-the-art algorithms.

Cardinality estimation (Figure 9(e)): We find that Elastic achieves comparable accuracy with the state-of-the-art algorithm (LC).

C. Elasticity

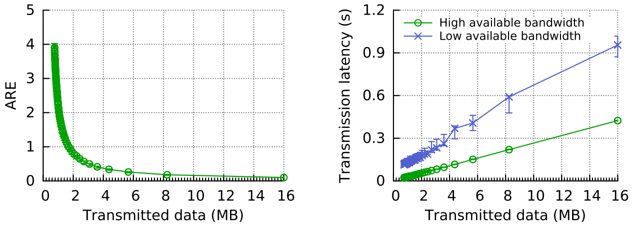
Adaptivity to Bandwidth: We first evaluate the accuracy of different compression and merging algorithms. From Figure 11(a)-(b), we find that the maximum algorithms always achieve better accuracy than the sum algorithms for both aggregation and merging. Specifically, maximum compression is between 1.24 and 2.38 times more accurate than sum compression, while maximum merging is between 1.26 and 1.33 times more accurate than sum merging.

Next, we constrain our NIC bandwidth to 0.5Gbps, and use this 0.5G NIC to evaluate the impact of available



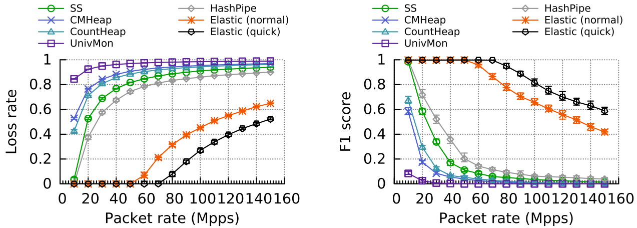
(a) Compression. (b) Merge.

Fig. 11. Accuracy comparison of different compression and merging algorithms for CM sketch in flow size estimation.



(a) ARE vs. bandwidth trade-off. (b) Latency vs. bandwidth trade-off.

Fig. 12. ARE and transmission delay comparisons for different sketch sizes in flow size estimation. We use TCP to transmit data. Transmitted data refers to the data that needs to be transmitted after compression (original memory is 16MB with 500KB heavy part).



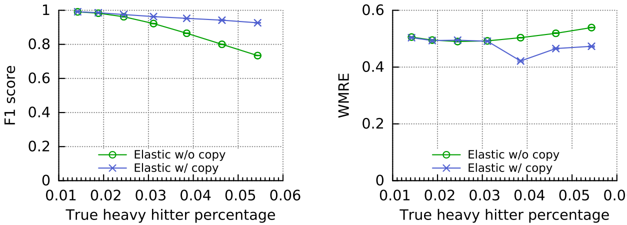
(a) Packet loss. (b) Accuracy of HH detection.

Fig. 13. Loss rate and accuracy comparisons for heavy hitter detection under different packet rates. "Elastic (quick)" means Elastic without light part. Due to the constraint of our NIC speed (*i.e.*, 40Gbps), we simulate the packet arriving process purely in memory and use ring buffer with multiple threads to do the measurement. The average number of heavy hitters in each traces is about 397. For more details, please refer to § VI.

bandwidth. Figure 12(a)-(b) show the results, where low available bandwidth means that we transmit sketch data on this 0.5G NIC with a consistently 0.5Gbps interfered traffic on it, and high available bandwidth means that we transmit sketch data without any interference of other traffic. We observe that transmitting data under low available bandwidth has a much longer latency than under high available bandwidth, and the transmission latency increases almost linearly as the transmitted data increases. Our Elastic provides a good trade-off between the accuracy and transmission delay: under low available bandwidth, we can send high-compression sketch data with decent accuracy to avoid long transmission delay.

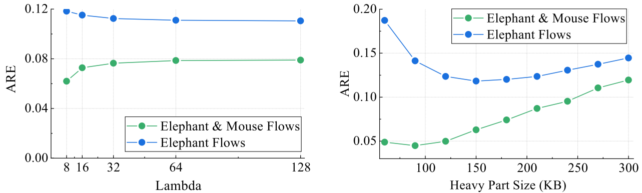
Adaptivity to Packet Rate: From Figure 13(a)-(b), we find that Elastic can sustain around 50Mpps packet rate without packet loss and with perfect accuracy, while Elastic without light part can even sustain around 70Mpps packet rate. For the other tested algorithms, only Space-Saving (SS) and HashPipe could achieve zero packet loss and perfect accuracy, but in that case, they can only sustain 10Mpps packet rate.

Adaptivity to Traffic Distribution We change the traffic distribution by changing the percentage of true heavy hitters. Specifically, we change the skewness of zipf distribution [67] and get multiple traces with different percentages of true heavy hitters. From Figure 14(a)-(b), we find that the copy operation (§ III-D) successfully avoids the accuracy degrading when traffic distribution changes.



(a) Heavy hitter detection. (b) Flow size distribution.

Fig. 14. Benefits of copy operation (§ III-D) for heavy hitter detection and flow size distribution under different traffic distributions.



(a) Effect of λ. (b) Effect of heavy part size.

Fig. 15. The effect of different λ and heavy part size on flow size estimation.

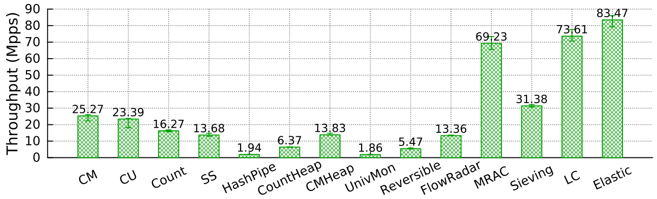


Fig. 16. Processing speed comparison for six tasks on CPU platform.

Effect of λ and Heavy Part Size We vary the parameter λ and test the ARE of flow size estimation. From Figure 15(a), we find that when λ increases, the ARE of the whole flows increases and the ARE of elephant flows decreases, which is because of that larger λ makes more middle-sized flows kept in the heavy part. However, the changes of these two AREs are small. As the ARE of elephant flows changes a little when varing λ, we set $\lambda = 8$ in our experiments for small ARE of mouse flows.

We fix the total memory to 600KB and vary the heavy part size from 60KB to 300KB. From Figure 15(b), we find that it reaches the lowest ARE when the heavy part size is 150KB. Therefore, we use 150KB as the heavy part size in our experiments.

D. Processing Speed

1) CPU Platform (single Core): We conduct this experiment on a server with two CPUs (Intel Xeon E5-2620V3@2.4GHZ) and 378GB DRAM. From Figure 16, we find that Elastic achieves much higher throughput than all other algorithms. Only three conventional algorithms (*i.e.*, MRAC, Sieving, LC) can reach a throughput of 30Mpps, while Elastic can reach more than 80Mpps. In particular, Elastic is 44.9 and 6.2 times faster than UnivMon and FlowRadar, respectively.

2) OVS Integration: We integrate our Elastic into OVS 2.5.1 with DPDK 2.2. We conduct this experiment on two servers, one for sending packets and one for OVS. Each server is equipped with two CPUs (Intel Xeon E5-2620@2.0GHz), 64 GB DRAM, and one Mellanox ConnectX-3 40 Gbit/s NIC.

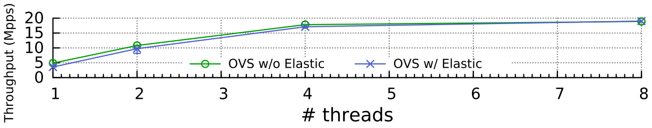


Fig. 17. Processing speed evaluation for Elastic in OVS.

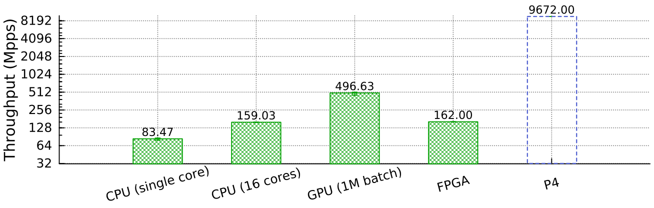


Fig. 18. Processing speed comparison for Elastic on different platforms. For the implementation of CPU with 16 cores, the master core sends flow IDs to 16 slave cores in a polling manner. We equally (for both heavy and light parts) divide the 600KB of memory to the 16 slave cores. We deploy the Elastic sketch in P4 switch running at line-rate of 6.5 Tbps, which translates into 9672Mpps when each packet has the minimum size of 64 bytes.

The two servers are connected directly through the NICs. From Figure 17, we find that in OVS, the throughput of Elastic gradually increases as the number of threads increases, while the overhead of using Elastic gradually decreases. When using a single thread, Elastic degrades the throughput of OVS by 26.8%; when using 4 threads, by 4.0% only; when using 8 threads, Elastic does not influence the throughput.

3) Other Platforms: From Figure 18, we find that Elastic achieves the highest processing speed on the P4 switch and the second highest speed on the GPU. Elastic achieves a comparable processing speed on the CPU with 16 cores and the FPGA. The processing speed of Elastic on CPU (16 cores), GPU (1M batch), FPGA, and P4 switch is $1.9, 5.9, 1.9, 115.9$ times higher than on the CPU (single core).

VIII. Conclusion

Fast and accurate network measurements are important and challenging in today's networks. So far, no work had focused on the issue of enabling measurements that are adaptive to changing traffic conditions. We propose the Elastic sketch, which is adaptive in terms of the three traffic characteristics. The two key techniques in our sketch are (1) Ostracism to separate elephant flows from mouse flows and (2) sketch compression to improve scalability. Our sketch is generic to measurement tasks and works across different platforms. To demonstrate this, we implement our sketch on six platforms: P4, FPGA, GPU, CPU, multi-core CPU, and OVS, to process six typical measurement tasks. Experimental results show that Elastic works well when the traffic characteristics vary, and outperforms the state-of-the-art in terms of both speed and accuracy for each of the six typical tasks. The source code from all platforms is available at Github [57].

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful suggestions.

References

[1] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 561–575.

[2] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. 28th Int. Conf. Very Large Data Bases*, Aug. 2002, pp. 346–357.

[3] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 101–114.

[4] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," Jul. 2017, *arXiv:1707.06778*. [Online]. Available: https://arxiv.org/abs/1707.06778

[5] E. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: Methods, evaluation, and applications," in *Proc. 3rd ACM SIGCOMM Conf. Internet Meas.*, Oct. 2003, pp. 234–247.

[6] X. Li *et al.*, "Detection and identification of network anomalies using sketch subspaces," in *Proc. 6th ACM SIGCOMM Conf. Internet Meas.*, Oct. 2006, pp. 147–152.

[7] G. Cormode, "Sketch techniques for approximate query processing," in *Foundations and Trends in Databases*. Boston, MA, USA: Now, 2011.

[8] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas.*, Nov. 2009, pp. 202–208.

[9] H. Xu, Z. Yu, C. Qian, X.-Y. Li, and L. Huang, "Minimizing flow statistics collection cost using wildcard-based requests in SDNs," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3587–3601, Dec. 2017.

[10] W. Cui and C. Qian, "DiFS: Distributed flow scheduling for adaptive routing in hierarchical data center networks," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Oct. 2014, pp. 53–64.

[11] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proc. Internet Meas. Conf.*, Nov. 2017, pp. 78–85.

[12] Y. Yu and C. Qian, "Space shuffle: A scalable, flexible, and high-performance data center network," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3351–3365, Nov. 2016.

[13] N. K. Sharma *et al.*, "Evaluating the power of flexible packet processing for network resource allocation," in *Proc. NSDI*, Mar. 2017, pp. 67–82.

[14] C. Hu *et al.*, "ANLS: Adaptive non-linear sampling method for accurate flow size measurement," *IEEE Trans. Commun.*, vol. 60, no. 3, pp. 789–798, Mar. 2012.

[15] K. Li and G. Li, "Approximate query processing: What is new and where to go?" *Data Sci. Eng.*, vol. 3, no. 4, pp. 379–397, 2018.

[16] N. B. Seghouani, F. Bugiotti, M. Hewasinghage, S. Isaj, and G. Quercini, "A frequent named entities-based approach for interpreting reputation in Twitter," *Data Sci. Eng.*, vol. 3, no. 2, pp. 86–100, 2018.

[17] 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.

[18] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *Proc. NSDI*, vol. 13, Apr. 2013, pp. 29–42.

[19] Q. Huang *et al.*, "SketchVisor: Robust network measurement for software packet processing," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 113–126.

[20] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, Aug. 2003.

[21] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proc. 29th Int. Colloq. Automata, Lang. Program.*, Jul. 2002, pp. 693–703.

[22] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. ICDT*. Berlin, Germany: Springer, 2005.

[23] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 164–176.

[24] A. Kumar, M. Sung, J. Xu, and J. Wang, "Data streaming algorithms for efficient and accurate estimation of flow size distribution," in *Proc. ACM SIGMETRICS*, Jun. 2004, pp. 177–188.

[25] C. Hu *et al.*, "DISCO: Memory efficient and accurate flow statistics for network measurement," in *Proc. IEEE 30th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2010, pp. 665–674.

[26] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better netflow for data centers," in *Proc. NSDI*, Mar. 2016, pp. 311–324.

[27] D. Zhuo *et al.*, "Understanding and mitigating packet corruption in data center networks," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 362–375.

[28] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM SIGCOMM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 123–137.

[29] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, "Measuring ISP topologies with rocketfuel," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 133–145, 2002.

[30] Z. Zhang *et al.*, "Optimizing cost and performance in online service provider networks," in *Proc. NSDI*, Apr. 2010, p. 3.

[31] Y. Zhang, M. Roughan, W. Willinger, and L. Qiu, "Spatio-temporal compressive sensing and Internet traffic matrices," in *Proc. ACM SIGCOMM Conf. Data Commun.*, vol. 39, no. 4, pp. 267–278, Oct. 2009.

[32] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from Googles network infrastructure," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 58–72.

[33] V. T. Lam, M. Mitzenmacher, and G. Varghese, "Carousel: Scalable logging for intrusion prevention systems," in *Proc. NSDI*, Apr. 2010, p. 24.

[34] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the self-similar nature of Ethernet traffic," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 23, no. 4, pp. 183–193, Oct. 1993.

[35] E. Rozner, J. Seshadri, Y. Mehta, and L. Qiu, "SOAR: Simple opportunistic adaptive routing protocol for wireless mesh networks," *IEEE Trans. Mobile Comput.*, vol. 8, no. 12, pp. 1622–1635, Dec. 2009.

[36] Y. O. Soon, E.-K. Lee, and M. Gerla, "Adaptive forwarding rate control for network coding in tactical manets," in *Proc. Mil. Commun. Conf.*, Oct./Nov. 2010, pp. 1381–1386.

[37] B. Yu, C.-Z. Xu, and M. Guo, "Adaptive forwarding delay control for VANET data aggregation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 1, pp. 11–18, Jan. 2012.

[38] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, Nov. 2010, pp. 267–280.

[39] G. Cormode, B. Krishnamurthy, and W. Willinger, "A manifesto for modeling and measurement in social media," *First Monday*, vol. 15, no. 9, pp. 1–18, 2010.

[40] T. Benson, A. Akella, and D. A. Maltz, "Unraveling the complexity of network management," in *Proc. NSDI*, Apr. 2009, pp. 335–348.

[41] I. N. Bozkurt, Y. Zhou, and T. Benson, "Dynamic prioritization of traffic in home networks," in *Proc. CoNEXT Student Workshop*, Dec. 2015, pp. 1–3.

[42] *Open-Source P4 Implementation of Features Typical of an Advanced l2/l3 Switch*. Accessed: Sep. 2018. [Online]. Available: https://github.com/p4lang/switch

[43] L. A. Jeni, J. F. Cohn, and F. De La Torre, "Facing imbalanced data–recommendations for the use of performance metrics," in *Proc. Proc. Humaine Assoc. Conf. Affect. Comput. Intell. Interact.*, Sep. 2013, pp. 245–251.

[44] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing Bloom filter: Challenges, solutions, and comparisons," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1912–1949, 2nd Quart., 2018.

[45] H. Dai, L. Meng, and A. Liu, "Finding persistent items in distributed datasets," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 1403–1411.

[46] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong, "Finding persistent items in data streams," *Proc. VLDB Endowment*, vol. 10, no. 4, pp. 289–300, 2016.

[47] H. Dai *et al.*, "Identifying and estimating persistent items in data streams," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2429–2442, Dec. 2018.

[48] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: A novel counter architecture for per-flow measurement," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 36, no. 1, pp. 121–132, 2008.

[49] V. Braverman and R. Ostrovsky, "Generalizing the layering method of indyk and woodruff: Recursive sketches for frequency-based vectors on streams," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Berlin, Germany: Springer, 2013.

[50] T. Bu, J. Cao, A. Chen, and P. P. C. Lee, "Sequential hashing: A flexible approach for unveiling significant patterns in high speed networks," *Comput. Netw.*, vol. 54, no. 18, pp. 3309–3326, 2010.

[51] M. Mitzenmacher, "Compressed bloom filters," *IEEE/ACM Trans. Netw.*, vol. 10, no. 5, pp. 604–612, Oct. 2002.

[52] B.-Y. Choi, J. Park, and Z.-L. Zhang, "Adaptive packet sampling for accurate and scalable flow measurement," in *Proc. IEEE Global Telecommun. Conf.*, vol. 3, Nov./Dec. 2004, pp. 1448–1452.

[53] S. Balaji *et al.*, "Erasure coding for distributed storage: An overview," *Sci. China Inf. Sci.*, vol. 61, no. 10, 2018, Art. no. 100301.

[54] X. Tang, S.-T. Xia, C. Tian, Q. Huang, and X.-G. Xia, "Special focus on distributed storage coding," *Sci. China Inf. Sci.*, vol. 61, no. 10, 2018, Art. no. 100300.

[55] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Trans. Database Syst.*, vol. 15, no. 2, pp. 208–229, Jun. 1990.

[56] *Barefoot Tofino: World's Fastest P4-Programmable Ethernet Switch ASICs*. Accessed: Sep. 2018. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

[57] *The Source Codes of Our and Other Related Algorithms*. Accessed: Sep. 2018. [Online]. Available: https://github.com/BlockLiu/ElasticSketchCode

[58] *The Open Virtual Switch Website*. Accessed: Sep. 2018. [Online]. Available: http://openvswitch.org

[59] *The CAIDA Anonymized Internet Traces*. Accessed: Sep. 2018. [Online]. Available: http://www.caida.org/data/overview/

[60] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting Bloom filter," *IEEE/ACM Trans. Netw.*, vol. 22, no. 4, pp. 1092–1105, Aug. 2014.

[61] S. Z. Kiss, É. Hosszu, J. Tapolcai, L. Rónyai, and O. Rottenstreich, "Bloom filter with a false positive free zone," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 1412–1420.

[62] A. Goyal, H. Daumé, III, and G. Cormode, "Sketch algorithms for estimating point queries in NLP," in *Proc. Joint Conf. Empirical Methods Natural Lang. Process. Comput. Natural Lang. Learn.*, Jul. 2012, pp. 1093–1103.

[63] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *Proc. 4th ACM SIGCOMM Conf. Internet Meas.*, Oct. 2004, pp. 207–212.

[64] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[65] M. T. Goodrich and M. Mitzenmacher, "Invertible Bloom lookup tables," in *Proc. 49th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep. 2011, pp. 792–799.

[66] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," in *Proc. ACM SIGMETRICS Joint Int. Conf. Meas. Modeling Comput. Syst.*, Jun. 2006, pp. 145–156.

[67] D. M. W. Powers, "Applications and explanations of Zipf's law," in *Proc. Joint Conf. New Methods Lang. Process. Comput. Natural Lang. Learn.*, Jan. 1998, pp. 151–160.

Tong Yang received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently a Research Assistant Professor with the Computer Science Department, Peking University. He published articles in SIGCOMM, SIGKDD, SIGMOD, SIGCOMM CCR, VLDB, ATC, ToN, ICDE, and INFOCOM. His research interests include network measurements, sketches, IP lookups, Bloom filters, sketches, and KV stores.

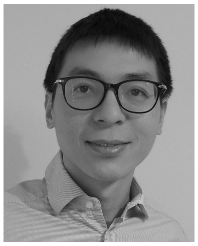
Jie Jiang is currently pursuing the master's degree with Peking University, advised by Tong Yang. His research interests include indexing, data sketches, and data stream processing systems.

Peng Liu is currently pursuing the master's degree with Peking University, advised by Tong Yang. He has participated several articles in network area. He is interested in network and data stream processing.

Qun Huang received the bachelor's degree in computer science from Peking University in 2011, and the Ph.D. degree from Department of Computer Science and Engineering, The Chinese University of Hong Kong, in 2015. Before joining ICT, he was a Researcher at Huawei Future Network Theory Lab in Hong Kong from 2015 to 2017. He is currently an Associate Professor with the Institute of Computing Technology, Chinese Academy of Sciences (ICT-CAS). He is supported by CAS Pioneer Hundred Talents Program.

Junzhi Gong graduated from Peking University, advised by Tong Yang. He is currently pursuing the Ph.D degree with Harvard University. He has some publications in the area of networking and data streaming processing. His research interests include network measurement and data stream processing systems.

Yang Zhou graduated *(summa cum laude)* from Peking University, advised by Tong Yang. He is currently pursuing the Ph.D. degree with Harvard University, advised by Minlan Yu. He is broadly interested in streaming algorithms, networked systems, and data-intensive systems.



Rui Miao received the B.S. degree from the University of Electronic Science and Technology of China in 2005, the M.S. degree from Tsinghua University in 2009, and the Ph.D. degree from the University of Southern California in 2018. He was a Visiting Scholar with Yale University in 2017. He is currently a Researcher with the Alibaba Group. He has around ten publications and two U.S. patents. His research has focused on monitoring and managing data center networks by leveraging emerging techniques, including software-defined networking and network virtualization. He is also interested in designing and building distributed systems with high scalability and reliability.

Xiaoming Li is currently a Professor in computer science and technology and the Director of Institute of Network Computing and Information Systems (NCIS) with Peking University, China. His current research interest is in search engine and web mining. He led the effort of developing a Chinese search engine (Tianwang) since 1999, and is the Founder of the Chinese web archive (Web InfoMall).

Steve Uhlig received the Ph.D. degree in applied sciences from the University of Louvain, Belgium, in 2004. From 2004 to 2006, he was a Post-Doctoral Fellow of the Belgian National Fund for Scientific Research (F.N.R.S.). From 2004 to 2006, he was a Visiting Scientist with the Intel Research Cambridge, U.K., and with the Applied Mathematics Department, University of Adelaide, Australia. From 2006 to 2008, he was with the Delft University of Technology, The Netherlands. Prior to joining Queen Mary, he was a Senior Research Scientist with Technische Universität Berlin/Deutsche Telekom Laboratories, Berlin, Germany. Since January 2012, he has been the Professor of Networks and Head of the Networks Research Group, Queen Mary University of London. From 2012 to 2016, he was a Guest Professor with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His thesis received the annual IBM Belgium/F.N.R.S. Computer Science Prize 2005.