Elastic Sketch: Adaptive and Fast Network-wide Measurements

Paper #5, 14 pages

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 ~ 45.2 times faster speed and 2.0 ~ 273.7 smaller error rate.

1 INTRODUCTION

1.1 Background and Motivation

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 [1–9]. Recently, sketch-based solutions[1] [8, 10] have been the best choice for network measurements [2, 3, 11, 12], owing to their higher accuracy compared to sampling methods [2, 4, 12] and fast speed.

Existing measurement solutions [4, 8, 10, 12–17] mainly focus on a good trade-off among accuracy, speed and memory usage. The state-of-the-art UnivMon [2] 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.

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

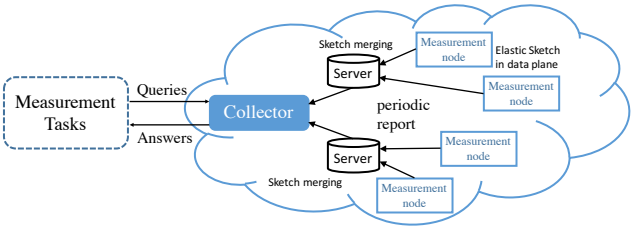


Figure 1: Network-wide measurements. Servers can be used to merge sketches when the network is large.

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 [2, 12, 18]. As shown in Figure 1, in data centers, administrators can deploy many measurement nodes, which periodically report sketches to a collector [2, 12, 18]. It requires available bandwidth for measurements, which share the same data plane as the user traffic. However, in data centers, network congestion is common. It can happen frequently within a single second [19] and be as large as more than half of the network bandwidth [9]. 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 [20–24]. 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 [25], 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) [26, 27], which is naturally variable and could vary drastically. For example, some routing protocols or mechanisms are proposed to adjust the packet sending rate to optimize network performance [28–30]. 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 the processing speed by actively discarding the unimportant information.

The third characteristic is flow size distribution. It is known that most flows are small [31], referred to as mice flows, while a very few flows are large, referred to as elephant flows [4, 22, 32–34]. An elegant solution is to accurately separate elephant flows from mice 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 [2] and FlowRadar [18]. However, our experimental results in Section 7 show that UnivMon is practically not accurate, while FlowRadar is not memory efficient.

1.2 Our Solution

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 mice 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, as shown in Figure 1, 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 mice 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 mice flows, which is based on our observation that the IDs of mice 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 [35].

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 [2].

1.3 Key Contributions

- We propose a novel sketch, namely the Elastic sketch, which is adaptive to bandwidth, packet rate and flow size distribution. It is also generic, fast and accurate. We propose two key techniques, one to separate elephant flows from mice flows, and one 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.

2 BACKGROUND AND RELATED WORK

In this section, we first discuss the challenges of adaptive measurements. Second, we show how to achieve generality. Finally, we introduce the most well-known network measurement systems from the literature.

2.1 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 mice flows [31], while a few flows are elephant flows [4, 22, 32]. To achieve memory efficiency, one can manage to separate elephant flows from mice flows. As elephant flows are often more important than mice 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 [36]. Therefore, it is challenging to dynamically allocate more memory for the elephant flows.

2.2 Generic Method for Measurements

We focus on the following network measurement tasks those have been extensively studied.

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[4, 12]. We observe that *flow IDs of mice flows are not necessary* for these tasks. As most flows are mice flows, discarding IDs of mice flows can significantly save memory and bandwidth of transmission. For this, we need to separate elephant flows from mice 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, 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.

2.3 Network Measurements Systems

Recently, well-known systems for measurements include UnivMon [2], Trumpet [37], OpenSketch [11], FlowRadar [18], SketchVisor [12], Marple[38], Pingmesh[39], and DREAM[40]. Among them, FlowRadar and UnivMon are generic, and thus are the most related work to this paper.

FlowRadar [18] records all flow IDs and flow sizes in a Bloom filter [41] and an Invertible Bloom Lookup Table (IBLT) [42]. To reduce memory usage, the authors propose an elegant solution of network-wide decoding. However, compared with sketches, its memory usage is still much higher.

UnivMon[2] is based on a key method named universal streaming [43]. Accuracy is guaranteed thanks to the theory of universal streaming. UnivMon is the first work to be generic, and achieves 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.

3 ELASTIC SKETCHES

In this section, we propose a new sketch, namely the Elastic sketch, to record the size of each flow. Then we show how our sketch adapts to the changes of three traffic characteristics.

3.1 Basic Version

Rationale: As mentioned above, we need to separate elephant flows from mice 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.

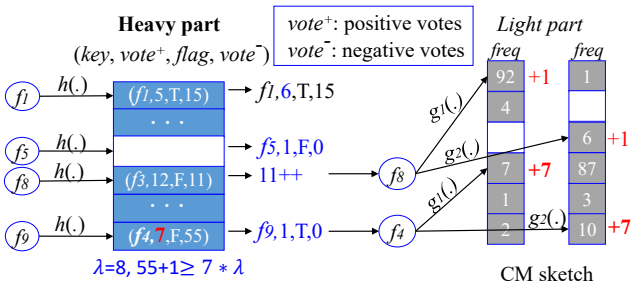


Figure 2: Basic version of Elastic. To insert f_9, after incrementing $votes^-$, $\frac{vote^-}{vote^+} \geqslant \lambda = 8$, hence f_4 is evicted from the heavy part and inserted into the light part.

Data structure: As shown in Figure 2, the data structure consists of two parts: a "heavy" part recording elephant flows and a "light" part recording mice flows. 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), while 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 [10] 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. A query is similar to an 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 into bucket $\mathcal{H}[h(f)\%w]$. Suppose the bucket stores $(key_1, vote^+, flag_1, vote^-)$. Similar to Ostracism, if f matches f_1, we increment $vote^+$. Otherwise, we increment $vote^-$, and decide whether to evict f_1 according to two votes. Specifically, there are four cases:

Case 1: The bucket is empty. We insert $(f, 1, F, 0)$ into it, where F mean false. The insertion ends.

Case 2: $f = key_1$. We just increment $vote^+$ by 1.

Case 3: $f \neq key_1$, and after incrementing $vote^-$, $\frac{vote^-}{vote^+} < \lambda$ (λ is a predefined threshold, *e.g.*, $\lambda = 8$)[3]. We insert $(f, 1)$ into the CM sketch: increment the hashed counters by 1.

Case 4: $f \neq key_1$, and after incrementing $vote^-$, $\frac{vote^-}{vote^+} \geqslant \lambda$. We "elect" flow f by setting the bucket to $(f, 1, F, 1)$, and evict flow key_1 to the CM sketch: increment the mapped counters by $vote^+$. Note that in this case the flag is set to true (T), because before f is elected, some votes of flow f may be inserted into the light part.

Query: The Elastic sketch can report the estimated size for any flow. For flows not in the heavy part, the CM sketch can report its size. There are two kinds of flows in the heavy part: 1) For each flow with the flag of false, its size is the corresponding vote$^+$ with no error; 2) For each flow with the flag of true, we need to add the corresponding vote$^+$ and the query result of the CM sketch.

The main shortcoming of this basic version is 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 mice flows significantly over-estimated.

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 [45]. We show only the following formula of the elephant collision rate P_{hc}, and the detailed proof is provided in Section A.1 of our technical report [44].

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

where H is the number of elephant flows and w is the number of buckets. 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 use two classic methods [46–54]: 1) by using multiple sub-tables (see Section 4.2); 2) by using multiple key-value pairs in one bucket (see Section 4.3).

Accuracy analysis: Here we analyze the accuracy without considering elephant collisions. 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 need to record the sizes of mice flows, and thus can use many small counters (*e.g.*, 8-bit counters), while traditional

[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.

[3]According to our experimental results on different datasets, we find when $\lambda \in [4, 128]$, the accuracy is optimal and has little difference, and we choose $\lambda = 8$. More detailed reason are provided in Section B.8 of our technical report [44].

sketch needs to use large counters (*e.g.*, 32-bit counters) to accommodate the elephant flows. Therefore, our light part can be very accurate. In sum, the accuracy of both elephant and mice flows is high.

3.2 Adaptivity to Available Bandwidth

Most flows are mice flows, thus the memory size of the light part is often much larger than that of the heavy part. To be adaptive to the available bandwidth, we propose to compress and merge the light part - the CM sketch. To the best of our knowledge, this is the first effort to compress sketches.

3.2.1 *Compression of Sketches.*

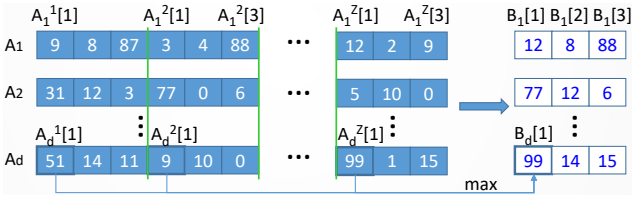


Figure 3: The Equal Division Compression algorithm.

Sum Compression (SC): As shown in Figure 3, assume a sketch $\mathbb{A}$ of size $zw' \times d$ (width $w = zw'$, depth d, z is an integer representing the compression rate). Our compression 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) We set $B_i[j] = \sum_{k=1}^{z} \{A_i^k[j]\}$ $(1 \leqslant i \leqslant d, 1 \leqslant j \leqslant z)$.

Maximum Compression (MC): Our algorithm is called Maximum Compression. The only difference with Sum Compression is to use "maximum" instead of "sum" in the last step: We set $B_i[j] = max\{A_i^1[j], A_i^2[j], ..., A_i^z[j]\}$ $(1 \leqslant i \leqslant d, 1 \leqslant j \leqslant z)$. The hash function after compression is $h_i(.)\%w\%w' = h_i(.)\%w'$ $(1 \leqslant i \leqslant d)$, thanks to the following classic Lemma.

Lemma 3.1. *Given an arbitrary integer i, two integers w and w', if w is divisible by w', then*

$$(i\%w)\ \%w' \ = \ i\%w' \tag{2}$$

Comparison of the two methods: As mentioned above, 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 using MM, the compressed $\mathcal{S}_1$ has much higher accuracy than $\mathcal{S}_2$.

Error bound of maximum compression (MC): Given a CM sketch with size $d \times zw$, we compress it into size of $d \times w$ using MC. Given an arbitrarily small positive number ϵ and an arbitrary flow f_j, the absolute error of the sketch after maximum compression is bounded by

$$Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} \leqslant \left\{1 - \left(1 - \frac{1}{\epsilon zw}\right)\left[1 - \frac{N}{zw(n_j + \epsilon N)}\right]^{z-1}\right\}^d \tag{3}$$

where n_j is the real size of f_j, $\hat{n}_j$ is the estimated size of f_j, and N is the total number of packets.

We have the following conclusions: 1) We prove that after Sum Compression, the error bound of the CM sketch does not change, while after maximum compression, 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 ~ 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. We refer the interested reader to the detailed proof and experiments in Section A.2, A.3, A.4, A.5, and B.7 of our technical report [44].

Forgetful Compression: When the collector receives many sketches, one commonly used method is to compress them, relying on a higher compression ratio for older sketches. As our compression method can be performed recursively, we can easily achieve this while keeping accuracy as high as possible. For example, we compress the sketch from i days ago into $1/2^i$ by using our compression algorithm.

3.2.2 *Merging of Sketches.*

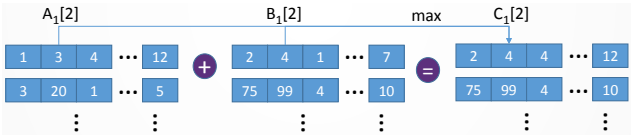


Figure 4: Maximum merging algorithm.

As shown in Figure 1, 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 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 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 4, 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 4, $C_1[2] = max\{A_1[2], B_1[2]\} = max\{3, 4\} = 4$. This merging method can be easily extended to multiple sketches. Obviously, after MM merging, the sketch still has

no under-estimation error. We can also merge two sketches of different widths, and the details are shown in Section A.7 of our technical report [44].

3.3 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 [28–30, 55]. 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, the state-of-the-art solution SketchVisor [12], 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 mice 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'. Therefore, each insertion needs one probe of a bucket in the heavy part. When packet rate goes down, we use our previous algorithms.

3.4 Adaptivity to Flow Size Distribution

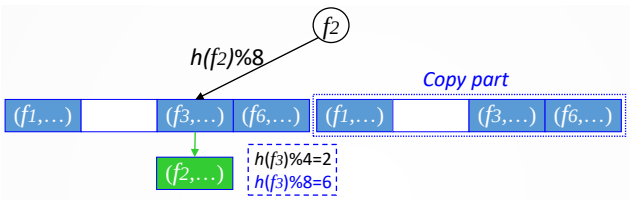


Figure 5: Duplication of the heavy part of Elastic. The original number of buckets in the heavy part is 4, and becomes 8 after duplication.

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 becomes full. At this moment, an incoming packet will probably be mapped into a bucket in which the flow is already larger than the predefined threshold. In this case, 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 3.1. 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.

Example: As shown in Figure 5, 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 $h(f_2)\%8 = 6$ 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.

4 OPTIMIZATIONS

4.1 Optimizing Light Part

Using CM sketch with d=1: For the CM sketch, a key metric is the depth d, *i.e.*, the number of arrays. Indeed, we can achieve higher accuracy if using d=3 or 4. However, we recommend setting $d = 1$, because of two reasons: 1) We care more about the feasibility of implementation and speed than accuracy; 2) Our sketch is already very accurate.

4.2 Hardware Version of Elastic Sketches

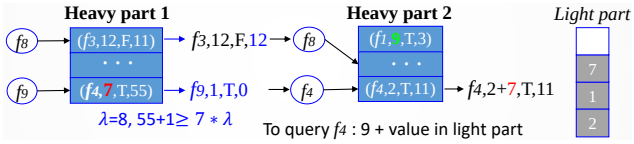


Figure 6: Hardware version of the Elastic sketch.

As mentioned above, the first classic solution for elephant collisions is using several sub-tables in the heavy part. Each sub-table is exactly the same as the heavy part of the basic version, but is associated with different hash functions. 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.

Examples: The insertion and query operations are slightly different from the basic version, and here we use examples to show the differences in Figure 6. 1) To insert f_8, in the first sub-table, the $vote^-$ is incremented by 1, and f_8 will be inserted into the next sub-table. 2) To insert f_9 in the first

sub-table, f_4 with flow size 7 is evicted, and inserted into the next stage. In the second sub-table, f_4 is mapped to the bucket with f_4. In this case, we just increment the value from 2 to 9. 3) To query a flow, as it could appear in multiple heavy parts, we need to add all the values.

According to our experimental results, using 4 subtables is a good trade-off between accuracy and feasibility of some hardware implementations, such as P4.

4.3 Software Version of Elastic Sketches

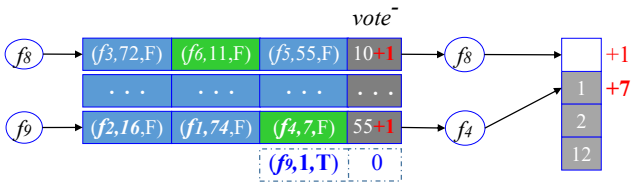


Figure 7: The software version of the Elastic sketch.

As mentioned above, the second classic solution for elephant collisions is: letting each bucket in the heavy part store several flows. This allows several elephant flows be recorded in one bucket, thus the elephant collision rate drops significantly. 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. 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.

Examples: We use two examples in Figure 7 to only show the differences between this software version and the basic version. 1) Given an incoming packet with flow f_8, we first hash it into a bucket. The bucket is full, and has no f_8. We increment the $vote^-$ from 10 to 11. The smallest flow is f_6 with flow size 11. Because $11 \leqslant 11 * \lambda = 11 * 8$, we do not evict flow f_6, but insert f_8 into the light part. 2) Given an incoming packet with flow f_9, we first hash it into a bucket. The bucket is full, and has no f_9. We increment the $vote^-$ from 55 to 56, because $56 \geqslant 7 * \lambda = 7 * 8$, we evict flow f_4 into the light part. After the eviction, we set the hashed bucket to $(f_9, 1, T)$, and set $vote^-$ to 0.

5 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: Given two Elastic sketches in two adjacent windows, for each flow in the heavy parts of these two sketches, we compare its flow size in the two sketches, and if the difference is larger than the predefined threshold, we report it as a heavy change.

Estimation of Flow Size Distribution, Entropy, and Cardinality: These three tasks care about both the elephant flows and mice 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 $(n_0, n_1, ..., n_{255})$ 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.

1) Estimating flow size distribution: We first calculate the distribution of the light part using the method in [17], then sum it with the distribution of the heavy part.

2) Estimating entropy: 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.

3) Cardinality: 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 [56]. The cardinality is the sum of the two numbers.

6 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. More implementation details are provided in our technical report, and the source code from all platforms is available at Github [44].

6.1 Hardware Version Implementations

P4 Implementation: We have fully built a P4 prototype of the Elastic sketch on top of a baseline switch.p4 [35] and compiled on a programmable switch ASIC [57]. We add 500 lines of P4 code that implements all the registers and metadata 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

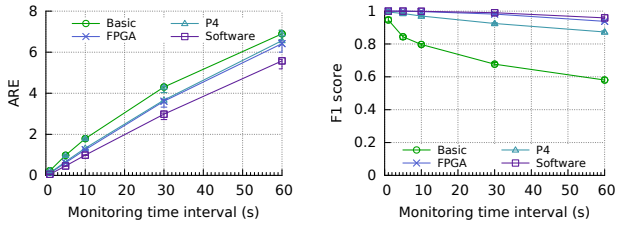
Table 1: Additional H/W resources used by Elastic sketch, normalized by the usage of the baseline switch.p4.

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%

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. We recommend $\lambda' = 32$, and the reason behind is shown in Section B of our technical report. 3) When a flow $(f, vote^{+})$ is evicted by another flow $(f_1, vote_1^{+})$, we set the bucket to $(f_1, vote^{+} + vote_1^{+})$. As mentioned in Section 4.2, we recommend using 4 subtables in the hardware version. In this way, we only need 4*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 1 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 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.

Comparison of the four versions: In sum, there are four versions of the Elastic sketch, and we compare the accuracy of them. Experimental results are shown in Figure 8. We compare the accuracy of these four versions for two tasks: flow size estimation and heavy hitter detection. As shown in Figure 8, the software, hardware, and P4 versions are always more accurate than the basic version. Specifically, when using monitoring time interval of 5s, for flow size



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

Figure 8: Accuracy comparison for three versions of Elastic on the tasks of flow size estimation and heavy hitter detection. Results are evaluated using the CAIDA4 trace (Table 2). Each algorithm uses 600KB memory. The heavy part in Elastic is 150KB.

estimation, the software and hardware version are 2.14, 1.6 times, and 1.46 times more accurate than the basic version, respectively; for heavy hitter detection, these three versions are 1.18, 1.18, and 1.17 times more accurate, respectively.

FPGA Implementation: We implement the Elastic sketch on a Stratix V family of Altera FPGA (model 5SEEBF45I2). The capacity of the on-chip RAMs (Block RAM) 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. The clock frequency of our implemented FPGA is 162.6 MHz, meaning processing speed of 162.6 Mpps.

GPU Implementation: We use the CUDA toolkit [58] to write programs on GPU to accelerate the insertion time of Elastic sketch. Two techniques, batch processing and multi-streaming, are applied to achieve the acceleration. We use an NVIDIA GPU (GeForce GTX 1080, the frequency is 1607 MHz. It has 8 GB GDDR5X memory and 2560 CUDA cores).

6.2 Software Version Implementations

CPU Implementation using SIMD: We use SIMD (Single Instruction Multiple Data) instructions to accelerate the processing speed of the heavy part. With the AVX2 instruction set, we can compare 8 32-bit integers with another set of 8 32-bit integers in a single instruction. In this way, the processing speed is much faster.

Multi-Core Implementation: We have implemented a multi-thread version of our Elastic sketch, and bind each thread in a CPU core.

OVS Implementation: We implement a prototype software switch. Our implementation is based on OpenVSwitch (OVS) [59], 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. Our implementation employs a multiple-threaded design to achieve scalability. The data plane is responsible to intercept packets and parse headers. It leverages a shared memory channel to dispatch the parsed headers to a multiple-threaded process, which performs the actual packet recording.

7 EXPERIMENTAL RESULTS

This section has five parts. After the experimental setup, we first compare the accuracy of Elastic against well-known algorithms for six typical measurement tasks. Then, we study the memory and bandwidth usage of our compression and merging algorithms. Next, we evaluate the three aspects of elasticity (*i.e.*, adaptive to bandwidth, packet rate, and traffic distribution) of Elastic. Finally, we compare the processing speed of Elastic against well-known algorithms for six typical tasks, and show the throughput of Elastic across different platforms.

7.1 Experimental Setup

Traces: We use four one-hour public traffic traces collected in Equinix-Chicago monitor from CAIDA [60]. The details of these traces are shown in Table 2. 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).

Table 2: CAIDA traces used in the evaluation.

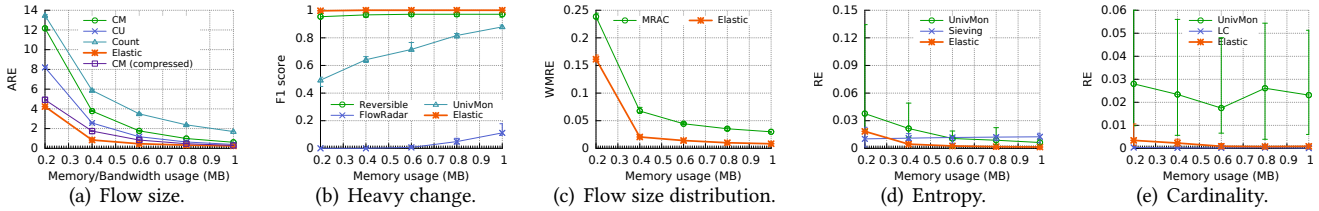
Trace	Date	# packets	# flows (SrcIP)
CAIDA1	2015/02/19	1164.9M	2.6M
CAIDA2	2015/05/21	1081.0M	3.9M
CAIDA3	2016/01/21	1835.1M	8.9M
CAIDA4	2016/02/18	1799.7M	8.4M

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. We use *ARE* to evaluate the accuracy of flow size (FS) estimation and heavy hitter (HH) detection. Note that the value of ARE for flow size estimation could be larger than anticipated, since the sizes of mice flows are often over-estimated while they are in the denominator of the ARE formula, leading to large average value of relative error.
- 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. We use F_1 *score* to evaluate the accuracy of heavy hitter and heavy change (HC) detection.
- *WMRE (Weighted Mean Relative Error) [12, 17]:* $\frac{\sum_{i=1}^{z}|n_i-\widehat{n}_i|}{\sum_{i=1}^{z}(\frac{n_i+\widehat{n}_i}{2})}$, 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. We use $WMRE$ to evaluate the accuracy of the flow size distribution (FSD).
- *RE (Relative Error):* $\frac{|True-Estimated|}{True}$, where $True$ and $Estimate$ are the true and estimated values, respectively. We use *RE* to evaluate the accuracy of entropy and cardinality estimations.
- *Throughput:* million packets per second (Mpps). We use $Throughput$ to evaluate the processing speed of the six tasks.

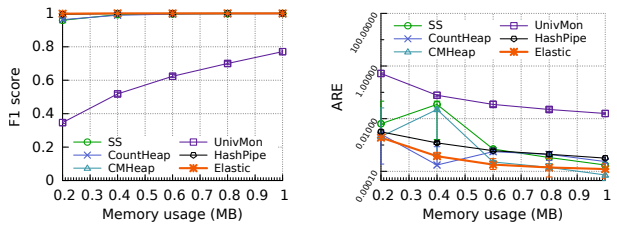
Setup: When comparing with other algorithms, we use the **software version** of Elastic. Specifically, we store 7 flows and a vote$^-$ for each bucket in the heavy part, and use one hash function and 8-bit counters in the light part. For each algorithm in each task, the default memory size is 600KB. Detailed configurations for each task are as follows:

- *Flow size estimation:* We compare four approaches: CM [10], CU [4], Count [14], and Elastic. For CM, CU, and Count, we use 3 hash functions as recommended in [61].
- *Heavy hitter detection:* We compare six approaches: Space-Saving (SS) [15], Count/CM sketch [10, 14] with a min-heap (CountHeap/CMHeap), UnivMon [2], HashPipe [16] 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.
- *Heavy change detection:* We compare Reversible sketch [62], FlowRadar [18], UnivMon, and Elastic. For Reversible, we use 4 hash functions as recommended in [62]. For FlowRadar, we use 3 hash functions in both the Bloom filter [41] and the IBLT part [42]; 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 [17] and Elastic.
- *Entropy estimation:* We compare UnivMon, Sieving [63], and Elastic. UnivMon uses the same setting as before. We use 8 sampling groups in Sieving.
- *Cardinality estimation:* We compare UnivMon, linear counting (LC) [56], and Elastic. UnivMon uses the same setting as before.



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

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



(a) F1 score. (b) ARE.

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

7.2 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 (§3.2.1) 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, ..., 1 MB, respectively. We find that our compression algorithm significantly improves the accuracy of CM sketch, making it nearly approach the accuracy of Elastic.

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.

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.

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).

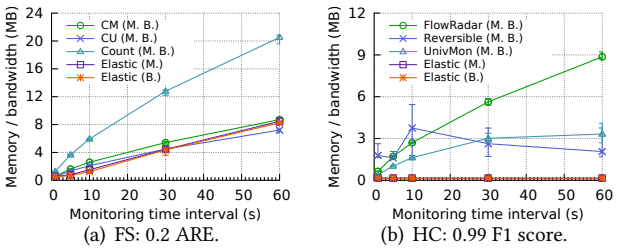
7.3 Memory and Bandwidth Usage

We measure the memory and bandwidth usage of different algorithms to achieve a fixed target accuracy, using different traces and different monitoring time intervals. Here, "memory" refers to the memory that is originally allocated to and used by the measurement algorithms, while "bandwidth" refers to the amount of data that needs to be transferred after each measurement epoch. When measuring the bandwidth usage of Elastic, we set the original memory to 16MB[4] with 500KB heavy part, run the maximum compression algorithm (§3.2.1), and measure the memory usage after compression (as the bandwidth usage) to achieve the fixed target accuracy. For the other measurement algorithms, their "memory" is equal to "bandwidth".

Monitoring time intervals (Figure 11(a)-(b)): We find that for flow size estimation, Elastic uses less memory and bandwidth than other algorithms for most monitoring time intervals; for heavy change detection, Elastic always uses much less memory and bandwidth than other algorithms. Specifically, Elastic uses 150KB memory or bandwidth to achieve 99% precision and recall rates for heavy change detection, irrespective of the monitoring time interval.

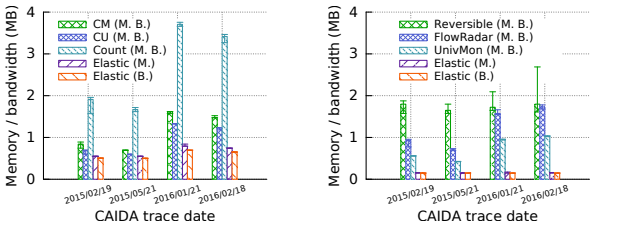
Traces (Figure 12(a)-(b)): We find that for flow size estimation and heavy change detection, Elastic always uses less memory and bandwidth than the other algorithms. We observe that for flow size estimation, the bandwidth usage of Elastic is always less than its memory usage, consistently with Theorem A.4 in Section A.4 of our technical report [44]. The reason that the bandwidth usage does

[4]The latest generation of switching ASICs has 50-100MB SRAM [64].



(a) FS: 0.2 ARE. (b) HC: 0.99 F1 score.

Figure 11: Memory (M.) and bandwidth (B.) usage for flow size estimation and heavy change detection to achieve target accuracy under different monitoring time intervals.



(a) FS: 0.2 ARE. (b) HC: 0.99 F1 score.

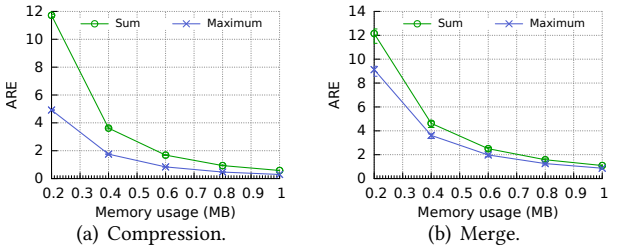
Figure 12: Memory (M.) and bandwidth (B.) usage for flow size estimation and heavy change detection to achieve target accuracy on different traces.

not significantly outperform the memory usage is that Elastic itself has achieved extremely high accuracy and thus the compression algorithm cannot easily improve it further.

7.4 Elasticity

7.4.1 *Adaptivity to Bandwidth.*

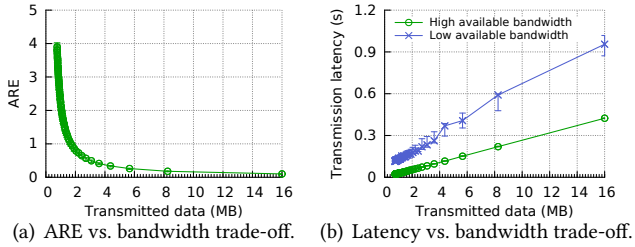
We first evaluate the accuracy of different compression and merging algorithms. From Figure 13(a)-(b), we find that the maximum algorithms always achieve better accuracy than the sum algorithms for both aggregation and merging. Specifically, maximum compression is between 1.24 and 2.38 times more accurate than sum compression, while maximum merging is between 1.26 and 1.33 times more accurate than sum merging.



(a) Compression. (b) Merge.

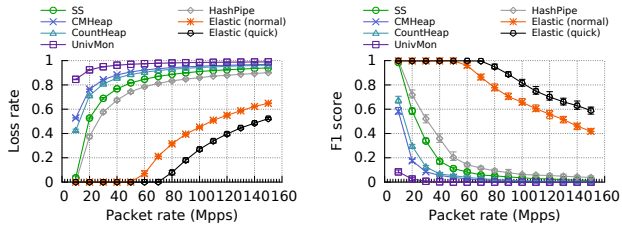
Figure 13: Accuracy comparison of different compression and merging algorithms for CM sketch in flow size estimation.

Next, we constrain our NIC bandwidth to 0.5Gbps, and use this 0.5G NIC to evaluate the impact of available bandwidth. Figure 14(a)-(b) show the results, where low available



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

Figure 14: 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). For more details, please refer to §7.3



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

Figure 15: 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. For more details, please refer to §6.

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.

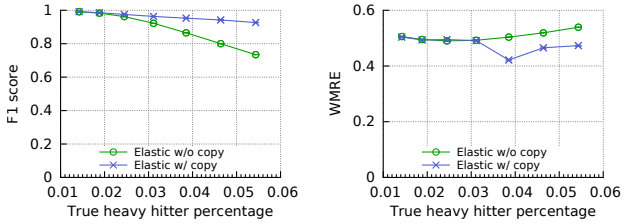
7.4.2 *Adaptivity to Packet Rate.*

From Figure 15(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.

7.4.3 *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 [65] and get multiple traces with different percentages of true heavy hitters. From Figure 16(a)-(b), we find that the copy operation (§3.4) successfully avoids the accuracy degrading when traffic distribution changes.



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

Figure 16: Benefits of copy operation (§3.4) for heavy hitter detection and flow size distribution under different traffic distributions.

7.5 Processing Speed

7.5.1 *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 17, 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.

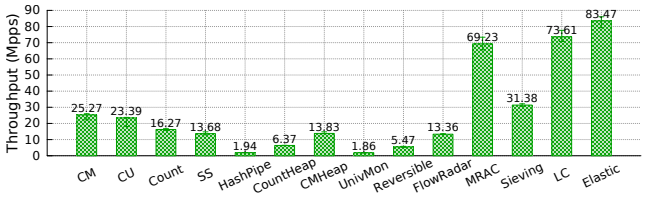


Figure 17: Processing speed comparison for six tasks on CPU platform.

7.5.2 *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. The two servers are connected directly through the NICs. From Figure 18, 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.

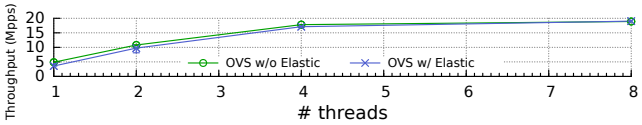


Figure 18: Processing speed evaluation for Elastic in OVS.

7.5.3 *Other Platforms.*

From Figure 19, 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).

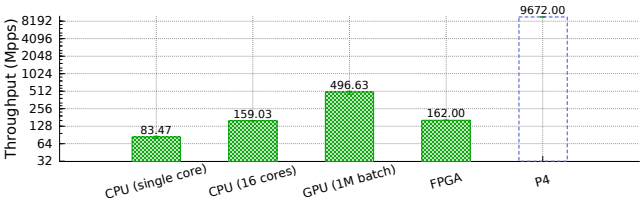


Figure 19: Processing speed comparison for Elastic on different platforms. For the implementation of CPU with 16 cores, the master core sends flow IDs to 16 slave cores in a polling manner. We equally (for both heavy and light parts) divide the 600KB of memory to the 16 slave cores. We deploy the Elastic 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.

8 CONCLUSION

Fast and accurate network measurements are important and challenging in today's networks. Indeed, with current highly variable traffic characteristics, changes in available bandwidth, packet rate, and flow size distribution can and do vary drastically at times. 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 above traffic characteristics. The two key techniques in our sketch are (1) Ostracism to separate elephant flows from mice 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. We have anonymously released the source code implemented on the six platforms at Github [44].

REFERENCES

[1] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. VLDB*, 2002.

[2] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016.

[3] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo Caggiani Luizelli, and Erez Waisbard. Constant time updates in hierarchical heavy hitters. *arXiv preprint arXiv:1707.06778*, 2017.

[4] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3), 2003.

[5] Er Krishnamurthy, Subhabrata Sen, and Yin Zhang. Sketchbased change detection: Methods, evaluation, and applications. In *In ACM SIGCOMM Internet Measurement Conference*. Citeseer, 2003.

[6] Xin Li, Fang Bian, Mark Crovella, Christophe Diot, Ramesh Govindan, Gianluca Iannaccone, and Anukool Lakhina. Detection and identification of network anomalies using sketch subspaces. In *Proc. ACM IMC*, 2006.

[7] MyungKeun Yoon, Tao Li, Shigang Chen, and J-K Peir. Fit a spread estimator in small memory. In *Proc. IEEE INFOCOM*, 2009.

[8] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

[9] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM, 2009.

[10] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1), 2005.

[11] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI*, volume 13, 2013.

[12] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017.

[13] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter braids: a novel counter architecture for per-flow measurement. *ACM SIGMETRICS Performance Evaluation Review*, 36(1), 2008.

[14] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Automata, languages and programming*, 2002.

[15] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. Springer ICDT*, 2005.

[16] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*. ACM, 2017.

[17] Abhishek Kumar, Minho Sung, Jun Jim Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proc. ACM SIGMETRICS*, 2004.

[18] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, 2016.

[19] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*. ACM, 2015.

[20] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring isp topologies with rocketfuel. *ACM SIGCOMM Computer Communication Review*, 32(4), 2002.

[21] Zheng Zhang, Ming Zhang, Albert G Greenberg, Y Charlie Hu, Ratul Mahajan, and Blaine Christian. Optimizing cost and performance in online service provider networks. In *NSDI*, 2010.

[22] Yin Zhang, Matthew Roughan, Walter Willinger, and Lili Qiu. Spatio-temporal compressive sensing and internet traffic matrices. In *ACM SIGCOMM Computer Communication Review*, volume 39. ACM, 2009.

[23] Mojgan Ghasemi, Partha Kanuparthy, Ahmed Mansy, Theophilus Benson, and Jennifer Rexford. Performance characterization of a commercial video streaming service. In *Proceedings of the 2016 Internet Measurement Conference*. ACM, 2016.

[24] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*. ACM, 2009.

[25] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016.

[26] Michael Mitzenmacher, George Varghese, et al. Carousel: scalable logging for intrusion prevention systems. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. USENIX Association, 2010.

[27] Will E Leland, Murad S Taqqu, Walter Willinger, and Daniel V Wilson. On the self-similar nature of ethernet traffic. In *ACM SIGCOMM computer communication review*, volume 23. ACM, 1993.

[28] Eric Rozner, Jayesh Seshadri, Yogita Mehta, and Lili Qiu. Soar: Simple opportunistic adaptive routing protocol for wireless mesh networks. *IEEE transactions on Mobile computing*, 8(12), 2009.

[29] Y Oh Soon, Eun-Kyu Lee, and Mario Gerla. Adaptive forwarding rate control for network coding in tactical manets. In *MILITARY COMMUNICATIONS CONFERENCE, 2010-MILCOM 2010*. IEEE, 2010.

[30] Bo Yu, Cheng-Zhong Xu, and Minyi Guo. Adaptive forwarding delay control for vanet data aggregation. *IEEE Transactions on Parallel and Distributed systems*, 23(1), 2012.

[31] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010.

[32] Graham Cormode, Balachander Krishnamurthy, and Walter Willinger. A manifesto for modeling and measurement in social media. *First Monday*, 15(9), 2010.

[33] Theophilus Benson, Aditya Akella, and David A Maltz. Unraveling the complexity of network management. In *NSDI*, 2009.

[34] Ilker Nadi Bozkurt, Yilun Zhou, Theophilus Benson, Bilal Anwer, Dave Levin, Nick Feamster, Aditya Akella, Balakrishnan Chandrasekaran, Cheng Huang, Bruce Maggs, et al. Dynamic prioritization of traffic in home networks. 2015.

[35] Open-source p4 implementation of features typical of an advanced l2/l3 switch. https://github.com/p4lang/switch.

[36] László A Jeni, Jeffrey F Cohn, and Fernando De La Torre. Facing imbalanced data–recommendations for the use of performance metrics. In *Affective Computing and Intelligent Interaction (ACII), 2013 Humaine Association Conference on*. IEEE, 2013.

[37] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016.

[38] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017.

[39] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al.

Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM Computer Communication Review*, volume 45. ACM, 2015.
[40] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Dream: dynamic resource allocation for software-defined measurement. *ACM SIGCOMM Computer Communication Review*, 44(4), 2015.
[41] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 1970.
[42] Michael T Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*. IEEE, 2011.
[43] Vladimir Braverman and Rafail 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*. Springer, 2013.
[44] The source codes of our and other related algorithms. https://github.com/ElasticSketch/ElasticSketch.
[45] Tian Bu, Jin Cao, Aiyou Chen, and Patrick PC Lee. Sequential hashing: A flexible approach for unveiling significant patterns in high speed networks. *Computer Networks*, 54(18), 2010.
[46] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. *ACM SIGCOMM Computer Communication Review*, 35(4), 2005.
[47] Adam Kirsch, Michael Mitzenmacher, and George Varghese. Hash-based techniques for high-speed packet processing. In *Algorithms for Next Generation Networks*. Springer, 2010.
[48] Berthold Vöcking. How asymmetry helps load balancing. *Journal of the ACM (JACM)*, 50(4), 2003.
[49] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 2014.
[50] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013.
[51] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014.
[52] Hyeontaek Lim, Donsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. USENIX, 2014.
[53] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, 2013.
[54] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011.
[55] Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang. Adaptive packet sampling for accurate and scalable flow measurement. In *Global Telecommunications Conference, 2004. GLOBECOM'04. IEEE*, volume 3. IEEE, 2004.
[56] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2), 1990.
[57] Barefoot tofino: World's fastest p4-programmable ethernet switch asics. https://barefootnetworks.com/products/brief-tofino/.
[58] Nvidia cuda c programming guide, version 9.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[59] The open virtual switch website. http://openvswitch.org.
[60] The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.
[61] Amit Goyal, Hal Daumé III, and Graham Cormode. Sketch algorithms for estimating point queries in nlp. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Association for Computational Linguistics, 2012.
[62] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.
[63] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. Data streaming algorithms for estimating entropy of network traffic. In *Proc. ACM SIGMETRICS*, 2006.
[64] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017.
[65] David MW Powers. Applications and explanations of zipf's law. In *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*. Association for Computational Linguistics, 1998.

A APPENDIX

In this section, we show some details those are not provided in the submission due to space limitation, including detailed derivation proofs and reasons of some claims.

A.1 Probability of Elephant Collisions

THEOREM A.1. *Within any bucket in the heavy part of the Elastic sketch, the probability of elephant collisions is*

$$P_{hc} = 1 - \left(\frac{H}{w} + 1\right) e^{-\frac{H}{w}} \tag{4}$$

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

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.*,

$$Pr\{Z = i\} = e^{-\frac{H}{w}} \frac{\left(\frac{H}{w}\right)^i}{i!} \tag{5}$$

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

$$\begin{aligned} P_{hc} &= 1 - Pr\{Z = 0\} - Pr\{Z = 1\} \\ &= 1 - \left(\frac{H}{w} + 1\right) e^{-\frac{H}{w}} \end{aligned} \tag{6}$$

□

A.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 [10], we can easily get the error bound of the CM sketch

$$Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} \leqslant \left(\frac{1}{\epsilon w}\right)^d \tag{7}$$

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 A.2. *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*

$$Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} \leqslant \left\{1 - \left(1 - \frac{1}{\epsilon zw}\right)\left[1 - \frac{N}{zw(n_j + \epsilon N)}\right]^{z-1}\right\}^d \tag{8}$$

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 with 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. In this way, the estimated size of f_j in the first array ($\hat{n}_j^1$) is

$$\hat{n}_j^1 = max(V_1 + n_j, V_2, V_3, \cdots, V_z) \tag{9}$$

And we have

$$\begin{aligned} &Pr\{\hat{n}_j^1 \geqslant n_j + \epsilon N\} \\ =&Pr\{max(V_1 + n_j, V_2, V_3, \cdots, V_z) \geqslant n_j + \epsilon N\} \\ =&1 - Pr\{max(V_1 + n_j, V_2, V_3, \cdots, V_z) < n_j + \epsilon N\} \\ =&1 - Pr\{V_1 + n_j < n_j + \epsilon N\} \prod_{i=2}^{z} Pr\{V_i < n_j + \epsilon N\} \end{aligned} \tag{10}$$

According to Markov inequality, it is easy to derive that

$$\begin{aligned} Pr\{V_1 + n_j < n_j + \epsilon N\} &= Pr\{V_1 < \epsilon N\} \\ &\geqslant 1 - \frac{E(V_1)}{\epsilon N} \\ &= 1 - \frac{1}{\epsilon zw} \end{aligned} \tag{11}$$

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

$$\begin{aligned} Pr\{V_i < n_j + \epsilon N\} &\geqslant 1 - \frac{E(V_i)}{n_j + \epsilon N} \\ &= 1 - \frac{N}{zw(n_j + \epsilon N)} \end{aligned} \tag{12}$$

Therefore, we have

$$\begin{aligned} &Pr\{\hat{n}_j^1 \geqslant n_j + \epsilon N\} \\ =&1 - Pr\{V_1 + n_j < n_j + \epsilon N\} \prod_{i=2}^{z} Pr\{V_i < n_j + \epsilon N\} \\ \leqslant&1 - \left(1 - \frac{1}{\epsilon zw}\right) \prod_{i=2}^{z} \left[1 - \frac{N}{zw(n_j + \epsilon N)}\right] \\ =&1 - \left(1 - \frac{1}{\epsilon zw}\right) \left[1 - \frac{N}{zw(n_j + \epsilon N)}\right]^{z-1} \end{aligned} \tag{13}$$

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

$$\hat{n}_j = min(\hat{n}_j^1, \hat{n}_j^2, \cdots, \hat{n}_j^d) \tag{14}$$

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

$$\begin{aligned} &Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} \\ =&Pr\{min(\hat{n}_j^1, \hat{n}_j^2 \cdots \hat{n}_j^d) \geqslant n_j + \epsilon N\} \\ =&\prod_{i=1}^{d} Pr\{\hat{n}_j^i \geqslant n_j + \epsilon N\} \\ \leqslant&\prod_{i=1}^{d} \left\{1 - \left(1 - \frac{1}{\epsilon zw}\right)\left[1 - \frac{N}{zw(n_j + \epsilon N)}\right]^{x-1}\right\} \\ =&\left\{1 - \left(1 - \frac{1}{\epsilon zw}\right)\left[1 - \frac{N}{zw(n_j + \epsilon N)}\right]^{z-1}\right\}^d \end{aligned} \tag{15}$$

□

A.3 Error Bound of CM sketches using SC Compression

Here we prove that the error bound does not change after using the SC Compression algorithm.

Theorem A.3. *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*

$$Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} \leqslant \left(\frac{1}{\epsilon w}\right)^d \tag{16}$$

Clearly that the error bound is identical to that of the CM sketch not using the SC 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

$$\hat{n}_j^1 = n_j + \sum_{i=1}^{z} X_i \tag{17}$$

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

$$\begin{aligned} Pr\{\hat{n}_j^1 \geqslant n_j + \epsilon N\} &= Pr\{\hat{n}_j^1 - n_j \geqslant \epsilon N\} \\ &\leqslant \frac{E(\hat{n}_j^1 - n_j)}{\epsilon N} = \frac{E(\sum_{i=1}^{z} X_i)}{\epsilon N} \\ &= \frac{\sum_{i=1}^{z} E(X_i)}{\epsilon N} = \frac{z \cdot \frac{N}{zw}}{\epsilon N} \\ &= \frac{1}{\epsilon w} \end{aligned} \tag{18}$$

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

$$Pr\{\hat{n}_j \geqslant n_j + \epsilon N\} = Pr\{\hat{n}_j^1 \geqslant n_j + \epsilon N\}^d \leqslant \left(\frac{1}{\epsilon w}\right)^d \tag{19}$$

□

A.4 Error Bound Comparison

In this subsection, We compare the error bound of the CM sketch before and after our maximum compression.

Theorem A.4. *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

$$\begin{aligned} P_{MC} &= \left\{1 - \left(1 - \frac{1}{\epsilon z w}\right)\left[1 - \frac{N}{zw(n_j + \epsilon N)}\right]^{z-1}\right\}^d \\ &< \left[1 - \left(1 - \frac{1}{\epsilon z w}\right)\left(1 - \frac{N}{zw\epsilon N}\right)^{z-1}\right]^d \\ &= \left[1 - \left(1 - \frac{1}{\epsilon z w}\right)^z\right]^d \end{aligned} \tag{20}$$

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

$$F(z) = \left(1 - \frac{x}{z}\right)^z \tag{21}$$

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

$$\begin{aligned} F(z) = \left(1 - \frac{x}{z}\right)^z &= 1 \cdot \left(1 - \frac{x}{z}\right)^z \\ &\leqslant \left[\frac{1 + z(1 - \frac{x}{z})}{z+1}\right]^{z+1} \\ &= \left(1 - \frac{x}{z+1}\right)^{z+1} = F(z+1) \end{aligned} \tag{22}$$

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

$$\begin{aligned} P_{MC} &< \left[1 - \left(1 - \frac{1}{\epsilon z w}\right)^z\right]^d \\ &= [1 - F(z)]^d \\ &\leqslant [1 - F(1)]^d \\ &= [1 - (1 - x)]^d = x^d \\ &= P_{CM} \end{aligned} \tag{23}$$

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

A.5 Proof of No Under-estimation Error

Theorem A.5. *After using the MC algorithm, the CM sketch and the CU 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 or the CU 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

$$A[g(f_i)\%w] \leqslant A'[g(f_i)\%(w/z)] \tag{24}$$

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. □

A.6 Why CU sketches Cannot be Implemented in P4

The CU sketch [4] achieves higher accuracy, but it needs to increment the smallest counters for each incoming packet and is hard to be implemented in P4 switches. This is because the P4 switch only supports to find the matched key in

multiple hash locations but cannot find the smallest counters with the matched key. In addition, to increment the smallest counters, the P4 switch needs a long atomic operation to hold multiple memory cells for atomic read-test-write operations. This longer atomic operation duration can be a disaster for today's P4 switch with highly parallelized packet processing.

A.7 Maximum Merging with different sizes

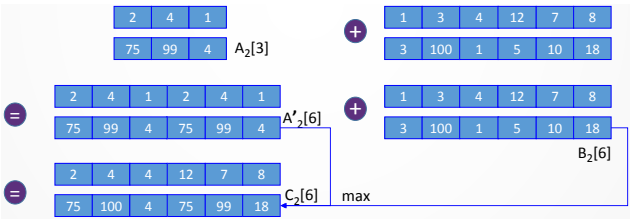


Figure 20: Maximum Merging algorithm for sketches of different size.

In real applications, one cannot be sure that two sketches have the same size. When two sketches have a different size, 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 20). 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: As shown in Figure 20, given a sketch $\mathbb{A}$ with size 2×3 and a sketch $\mathbb{B}$ with size 2×6. As the `least common multiple` of 3 and 6 are 6, then we copy sketch $\mathbb{A}$ and get a new sketch $\mathbb{A}'$ with size of 2×6. 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 CMA algorithm. For example, after merging, $C_2[6] = max\{A'_2[6], B_2[6]\} = max\{4, 18\} = 18$.

B ALL IMPLEMENTATION DETAILS

In this section, we first describe the hardware implementation of the elastic sketch on P4, FPGA, and GPU Platforms. Then, we describe the software implementation on CPU, multi-core CPU, and OVS platforms. The source code from all platforms is available at Github [44].

B.1 P4 Implementation

We have fully built a P4 prototype of the Elastic sketch on top of a baseline switch.p4 [35] and compiled on a programmable switch ASIC [57]. 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.

Table 3: 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%

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. We recommend $\lambda' = 32$, and the reason behind is shown in Section B of our technical report. 3) When a flow $(f, vote^{+})$ is evicted by another flow $(f_1, vote_1^{+})$, we set the bucket to $(f_1, vote^{+} + vote_1^{+})$. As mentioned in Section 4.2, we recommend using 4 subtables in the hardware version. In this way, we only need 4*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 3 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 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.

B.2 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 ~ 64. We therefore choose 8 and the operation $*8$ is replaced by $<< 3$.

We implement the Elastic sketch on a 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 [44].

B.3 GPU Implementation

We use the CUDA toolkit [58] to write programs on GPU to accelerate the insertion time of Elastic sketch. Two techniques, batch processing and multi-streaming, are applied to achieve the acceleration. GPU has a large amount 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.

B.4 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 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.

B.5 Multi-Core Implementation with Merging

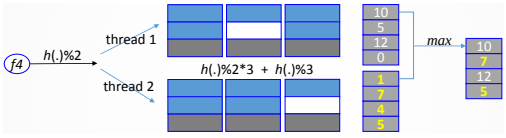


Figure 21: Merging of small elastic sketches on multi-core CPU.

Multi-core CPUs are popular nowadays, so we tailor the elastic sketch implementation to multi-core. As shown in Figure 21, we first build one small elastic sketch for each thread. For each incoming packet, we map its flow ID to a specific thread. 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 21, after merging, the hash function becomes h(.)%2 * 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 21, the first counters of the two light parts are 10 and 1, and the first counter after merging is max{10,1}=10.

B.6 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 OpenVSwitch (OVS) [59], 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.

Our implementation employs a multiple-threaded design to achieve scalability. The data plane is responsible to intercept packets and parse their headers. It leverages a shared memory channel to dispatch the parsed headers to a multiple-threaded process, which performs the actual packet recording.

B.7 Compressing Speed with/without SIMD

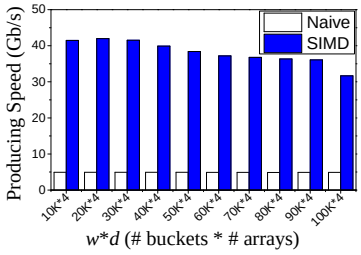


Figure 22: The compressing speed of the CM sketch (MB/s).

Our experimental results show that the producing speed of SF-sketch using SIMD is 6.5 ∼ 8.5 *times faster than that without SIMD.* As shown in Figure 22, the x-axis is $w * d$, where d is the number of arrays and w is the number of buckets in each array. Each bucket includes 16 counters, and each counter is 32-bit long to achieve word alignment. The y-axis is the compressing speed which indicates how many Gigabytes (GB) in the CM sketch is processed in one second. As the memory size of the CM sketch increases, the producing speed drops slowly. Specifically, when $w * d$ is 40K*4, the compressing time is only 2 milliseconds when using SIMD and only one CPU core, which is small enough to be ignored in most applications.

B.8 Choosing Optimal Values of λ and λ'

For the basic version, hardware version, and software version, we use λ. As the value of λ increases, the number of evictions will decreases. As mentioned above, the error occurs only in the light part, thus the value of λ has little effect on the accuracy. According to our experimental results on different datasets, we find when $\lambda \in [4, 128]$, the accuracy is optimal and has little difference, and thus we choose $\lambda = 8$, while $*8$ can be achieved by <<3.

For the P4 version of the Elastic sketch, it does not has flag, and during eviction, there will be error because the size of the incoming flow is set to the sum of the two flow sizes instead of 1. Therefore, 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 16, but when varying λ' from 16 to 128, the accuracy increases little. Therefore, we choose $\lambda' = 32$, while $*32$ can be achieved by <<5.