Elastic Sketch: Adaptive and Fast Network-wide Measurements

Tong Yang  
Peking University  
yangtongemail@gmail.com

Jie Jiang  
Peking University  
jie.jiang@pku.edu.cn

Peng Liu  
Peking University  
liu.peng@pku.edu.cn

Qun Huang  
Institute Of Computing Technology, CAS  
huangqun@ict.ac.cn

Junzhi Gong  
Peking University  
gongjunzhi@pku.edu.cn

Yang Zhou  
Peking University  
zhou.yang@pku.edu.cn

Rui Miao  
Alibaba Group  
miao.rui@alibaba-inc.com

Xiaoming Li  
Peking University  
lxm@pku.edu.cn

Steve Uhlig  
Queen Mary University of London  
steve.uhlig@quml.ac.uk

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.

CCS CONCEPTS

• Networks → Network monitoring; Network measurement;

KEYWORDS

Sketches; Network measurements; Elastic; Compression; Generic

ACM Reference Format:

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[8, 10] have been widely accepted in network measurements [2, 3, 11, 12], thanks to their higher accuracy compared to sampling methods [2, 4, 12] and their 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,

In this paper, sketches refers to data streaming algorithms that can be used for network measurements.
such as network congestion, scans and DDoS attacks. In such cases, traffic characteristics vary drastically, significantly degrading the measurement performance. Therefore, it is desirable to achieve accurate network measurements when traffic characteristics vary a lot.

The first traffic characteristic is the available bandwidth. In data centers, administrators care more about the state of the whole network than a single link or node, known as network-wide measurements [2, 12, 18]. 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 [20–27]. 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 [28], 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) [29, 30], 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 [31–33]. 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 [34], referred to as mouse flows, while a very few flows are large, referred to as elephant flows [4, 22, 35–37]. An elegant solution is to accurately separate elephant flows from mouse flows, and use different data structures to store them. However, the flow size distribution varies. One might think we can predict traffic and allocate appropriate size of memory for sketches in advance. It may be easy to predict the number of elephant flows in one hour, but hard at timescales of seconds or milliseconds. Therefore, it is desirable to design an elastic data structure which can dynamically allocate appropriate memory size for elephant flows.

In summary, this leads us to require our sketch to be elastic: adaptive to bandwidth, packet rate, and flow size distribution. Besides them, there are three other requirements in measurements: 1) generic, 2) fast, and 3) accurate. First, each measurement node often has to perform several tasks. If we build one data structure for each task, processing each incoming packet requires updating all data structures, which is time- and space-consuming. Therefore, one generic data structure for all tasks is desirable. Second, to be fast, the processing time of each packet should be small and constant. Third, being accurate implies that the error rate should be small enough when using a given amount of memory. Among all existing solutions, no solution is elastic, and only two well known solutions claim to be generic: UnivMon [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 mouse flows in the light part.

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

To make our solution “generic”, we do the following. 1) To be generic in terms of measurement tasks, we keep all necessary information for each packet, but discard the IDs of mouse flows, which is based on our observation that the IDs of mouse flows are memory consuming but practically useless. 2) To be generic in terms of platforms, we propose a software and a hardware version of the Elastic sketch, to make our sketch easy to be implemented on both software
and hardware platforms. Further, we tailor a P4 version of the Elastic sketch, given the popularity of this platform [38].

Owing to the separation and discarding of unnecessary information, our sketch is accurate and fast: experimental results show that our sketch achieves 44.6 ~ 45.2 times faster speed and 2.0 ~ 273.7 smaller error rate than the state-of-the-art: UnivMon [2].

1.3 Key Contributions

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

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 $S_1$, $S_2$ with the memory size of $M$ and $M/2$, and then we can send $S_2$ to the collector when the available bandwidth is small. A better solution is to build only $S_1$, and quickly compress it into a half. It is not hard for the compressed $S_1$ to achieve the same accuracy with $S_2$. However, it is challenging for the compressed $S_1$ to achieve much higher accuracy than $S_2$, which is one design goal of this paper.

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

Third, in real network traffic, the flow size distribution is skewed and variable. “Skewed” means most flows are mouse flows [34], while a few flows are elephant flows [4, 22, 35]. To achieve memory efficiency, one can manage to separate elephant flows from mouse flows. As elephant flows are often more important than mouse flows, it is desirable to assign appropriate memory size for the elephant flows. Unfortunately, the number of elephant flows is not known in advance and hard to predict [39]. 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 mouse flows are not necessary for these tasks. As most flows are mouse flows, discarding IDs of mouse flows can significantly save memory and bandwidth of transmission. For this, we need to separate elephant flows from mouse flows. To address this problem, we leverage the spirit of Ostracism, and propose a fast and accurate separation algorithm. Finally, our sketch is both generic and memory efficient.
Another meaning of generic is that the algorithm can be implemented on various platforms. For small companies, the traffic speed may be not high, and measurement on CPU is a good choice. For large companies, the traffic speed could be very high, and then hardware platforms should be used for measurements to catch up with the high speed. Therefore, the measurement solution should be generic, and can make good performance trade-off on different platforms.

2.3 Network Measurements Systems

Recently, well-known systems for measurements include UnivMon [2], Trumpet [40], OpenSketch [11], FlowRadar [18], SketchVisor [12], Marple[41], Pingmesh[42], and DREAM[43]. 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 [44] and an Invertible Bloom Lookup Table (IBLT) [45]. 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 [46]. 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

3.1 Basic Version

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

3.1.1 Data Structure and Operations.

Data structure: As shown in Figure 1, the data structure consists of two parts: a "heavy" part recording elephant flows and a "light" part recording mouse flows. The heavy part \( \mathcal{H} \) is a hash table associated with a hash function \( h(.) \). Each bucket of the heavy part records the information of a flow: flow ID (key), positive votes (vote\(^+\)), negative votes (vote\(^-\)), and flag. Vote\(^+\) records the number of packets belonging to this flow (flow size). Vote\(^-\) records the number of other packets. The flag indicates whether the light part may contain positive votes for this flow.

The light part is a CM sketch. A CM sketch [10] consists of \( d \) arrays \( (L_1, L_2, ..., L_d) \). Each array is associated with one hash function, and is composed of \( w \) counters. Given an incoming packet, the CM sketch extracts the flow ID, computes \( d \) hash functions to locate one counter per array, and increments the \( d \) counters (we call them \( d \) hashed counters) by 1. The query is similar to the insertion: after obtaining the \( d \) hashed counters, it reports the minimum one.

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

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

Case 2: \( f = f_1 \). We just increment vote\(^+\) by 1.

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

Case 4: \( f \neq f_1 \), and \( \frac{\text{vote}^-}{\text{vote}^+} \geq \lambda \) after incrementing vote\(^-\) by 1. We "elect" flow \( f \) by setting the bucket to \( (f, 1, T, 1) \), and evict flow \( f_1 \) to the CM sketch: increment the mapped counters by vote\(^+\). Note that in this case the flag is set to T.

\(^2\)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 [47].
(true), because some votes of flow $f$ may be inserted into the light part before $f$ is elected. **Query:** For any flow not in the heavy part, the light part (the CM sketch) reports its size. For any flow $f$ in the heavy part, there are two cases: 1) The flag of $f$ is false. Its size is the corresponding vote\(^4\) with no error; 2) The flag of $f$ is true. We need to add the corresponding vote\(^6\) and the query result of the CM sketch.

3.1.2 Accuracy Analysis.

The estimated value of a flow in the Elastic sketch has the following error bound, and the detailed proof is provided in our technical report [47].

**Theorem 3.1.** Let vector $f = (f_1, f_2, ..., f_n)$ denote the size vector for a stream, where $f_i$ denotes the size of the $i$-th flow. Given two parameters $e$ and $\delta$, let $w = \lceil \frac{e}{\delta} \rceil$ (e is Euler’s number) and $d = \lceil \ln \frac{1}{\delta} \rceil$. Let Elastic with $d$ ($d$ is the number of counter arrays) and $w$ ($w$ is the number of counters in each array) records the stream with $f$. The reported size $\hat{f}_i$ by Elastic for flow $i$ is bounded by

$$\hat{f}_i \leq f_i + e\|f\|_1^4 < f_i + e\|f\|_1$$

(1)

with probability at least $1 - \delta$, where $f_L$ denotes the size vector of the sub-stream recorded by the light part.

According to Theorem 3.1, the estimation error of Elastic is bounded by $\|f\|_1$, instead of $\|f\|_1$ in Count-Min. In practice, often, most packets of a stream are recorded in the heavy part, $\|f_L\|_1$ is usually significantly smaller than $\|f\|_1$. Thus Elastic has a much tighter error bound than Count-Min when the parameters ($d$ and $w$) are the same.

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

The accuracy of Elastic drops in the worst case – elephant collisions: when two or more elephant flows are mapped into the same bucket, some elephant flows are evicted to the light part and could make some mouse flows significantly 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 [48]. 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 [47].

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

(2)

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 [49–57]: 1) by using multiple key-value pairs in one bucket (see Section 4.3).

3.2 Adaptivity to Available Bandwidth

To adapt to the available bandwidth, we propose to compress the sketches before sending them. Most flows are mouse flows, thus the memory size of the light part is often much larger than that of the heavy part. In this section, we will show how to compress and merge the light part - CM sketch. To the best of our knowledge, this is the first effort to compress sketches.

3.2.1 Compression of Sketches.

To compress a sketch, our key idea is first to group the counters, and then merge counters in the same group into one counter.

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

**Lemma 3.2.** Given an arbitrary integer $i$, two integers $w$ and $w'$, if $w$ is divisible by $w'$, then $(i \% w) \% w' = i \% w'$.
For example, \((10\%) \times 3 = 10\% 3\). This lemma will be repeated leveraged in this paper.

**Merging:** we propose two merging methods. The first method is to sum up the counters in each group, i.e., \(B_i[j] = \sum_{k=1}^{z} A_i^k[j]\). We name this method **Sum Compression** (SC). As mentioned in Section 2, to adapt to available bandwidth, one can build two CM sketches \(S_1\) and \(S_2\) with memory size of \(M\) and \(M/2\). A better solution is to compress \(S_1\) to a half. Using SC, the compressed \(S_2\) has the same accuracy as \(S_2\), while SC does not take advantage of the information recorded by \(S_1\). The second method is **Maximum Compression** (MC). Instead of “sum”, we can use “maximum”, i.e., \(B_i[j] = \max\{A_i^1[j], A_i^2[j], ..., A_i^z[j]\}\). Compared with SC, the sum operation in MC uses more information in \(S_1\), and thus has better accuracy.

**Error bound of maximum compression** (MC): Given a CM sketch with size \(d \times 2w\), we compress it into size of \(d \times w\) using MC. Given an arbitrarily small positive number \(\epsilon\) and an arbitrary flow \(f_j\), the absolute error of the sketch after maximum compression is bounded by

\[
Pr(\hat{n}_j \geq n_j + \epsilon N) \leq \left(1 - \left(1 - \frac{1}{\epsilon^2 w}\right)^{\frac{N}{\epsilon w(n_j + \epsilon N)}}\right)^d
\]

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.

About SC and MC, 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 Multiple Data). 4) There is no need for de-compression. 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 [47].

### 3.2.2 Merging of Sketches.

As shown in Figure 3, one can use servers to save bandwidth. Each server receives many sketches from measurement nodes, merges them, and then sends them to the collector. For the sake of merging, we need to use the same hash functions for all sketches. If they have common flow IDs, we propose to use a naive method — Sum merging. Otherwise, we propose a novel method, namely Maximum merging.

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

**Maximum Merging for same-size sketches:** Our algorithm is named Maximum Merging (MM). As shown in Figure 4, given two sketches \(A\) and \(B\) of size \(w \times d\), we build a new sketch \(C\) also of size \(w \times d\). We simply set \(C_i[j] = \max\{A_i[j], B_i[j]\}\) \((1 \leq i \leq d, 1 \leq j \leq w)\). For example in Figure 4, \(C_i[2] = \max\{A_i[2], B_i[2]\} = \max\{3, 4\} = 4\). 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.8 of our technical report [47].

![Figure 4: Maximum merging algorithm.](image)

### 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 [31–33, 58]. 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 mouse flows. The insertion process of the heavy part is almost unchanged except in the following case: if a flow \(f\) in a bucket is replaced by another flow \(f’\), the flow size of \(f’\) is set to...
the flow size of \( f \). Therefore, each insertion needs one probe of a bucket in the heavy part. When packet rate goes down, we use our previous algorithms.

Fortunately, this strategy achieves much higher speed at the cost of slightly degrading the accuracy. When this strategy is activated, we do not discard the light part, but only do not update it during insertion, as light part will still be used during queries. This means that only information recorded by the light part when high packet rate occurs is lost. This strategy does not affect much the query accuracy in most cases, since the packet rate is usually low.

### 3.4 Adaptivity to Flow Size Distribution

A key metric of the flow size distribution is the number of elephant flows. As it can vary a lot, it is hard to determine the size of the heavy part. To address this issue, we need to make the heavy part adaptive to changes in the traffic distribution. We propose a technique to dynamically double the heavy part. It works as follows. Initially, we assign a small memory size to the heavy part. As more and more elephant flows are inserted, the heavy part will become full. We define a threshold \( T_l \). If an incoming packet is mapped into a bucket in which all flows are larger than \( T_l \), we regard the bucket is full. If the number of full buckets exceeds a threshold, we regard the heavy part is full. When the heavy part becomes full, we propose the following copy operation: just copy the heavy part and combine the heavy part with the copy into one. The hash function is changed from \( h(\cdot)\%w \) to \( h(\cdot)\%(2w) \). Again, this copy operation works thanks to Lemma 3.2. After the copy operation, half of the flows in the buckets should be removed. The remove operation can be performed incrementally. For each insertion, we can check all flows in the mapped bucket, and on average half of the flows are not mapped to that bucket and can be removed. Even though some buckets may end up not being cleaned, this does not negatively impact the algorithm.

**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 \( f_2 \) is. We compute \( h(f_3)\%8 = 6 \) and find that it should be mapped to the bucket in the copy part. Therefore, we replace \( f_2 \) by \( f_3 \).

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

In addition to enlarging the heavy part, we can also actively compress heavy part. The compression method is similar to Maximum Compression (MC, Section 3.2.1). Different from MC, for the heavy part, we merge buckets (key, \( vote^+ \), flag, \( vote^- \)) rather than counters. Take merging two buckets as an example. Given two buckets, for the two keys in the buckets, we query their frequencies in the Elastic sketch, and keep the larger one, and evict the other one into the light part. The compression operation endows the Elastic sketch the ability to actively release memory when needed.

### 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

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 \( f_6 \), in the first sub-table, the \( vote^- \) is incremented by 1, and \( f_6 \) will be inserted into the next sub-table. 2) To insert \( f_9 \) in the first sub-table, \( f_9 \) with flow size 7 is evicted, and inserted into the next stage. In the second sub-table, \( f_9 \) is mapped to the bucket with \( f_3 \). 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 P4Switch.

4.3 Software Version of Elastic Sketches

![Diagram]

Figure 7: Software version of the Elastic sketch.

As mentioned above, the second classic solution for elephant collisions is: letting each elephant flow in the heavy part store several flows. This allows several elephant flows to 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 $11 \leq 11 * \lambda = 7 * 8$, 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 $56 \geq 7 * \lambda = 7 * 8$, we evict flow $f_9$ into the light part. After the eviction, we set the hashed bucket to $(f_0, 1, T)$, 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: For two adjacent time windows, we build two Elastic sketches, respectively. To find heavy changes with threshold $T$, one common used method is to check all flows in each time window with size no less than $T$. Therefore, we check all flows in the heavy parts of the two sketches, and if the size difference of a flow in the two windows is larger than $T$, 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 mouse flows. For flows in the heavy part, we can get their information directly. For flows in the light part, we can get the needed information from the counter distribution. So at the end of each time window, we collect the counter distribution array $(n_0, n_1, ..., n_{255})$ 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 estimate the distribution of the light part using the basic MRAC algorithm proposed by [17], and then sum it with the distribution of the heavy part.

2) Estimating entropy: we compute the entropy based on the flow size distribution as $- \sum (i \times \frac{n_i}{m} \log \frac{n_i}{m})$, 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 [59]. The cardinality is the sum of the two numbers.

For other tasks (e.g., DDoS, SuperSpreader, and more [60–62]), we will study how to apply Elastic in the future work.

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 [47].

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 [38] and compiled on a programmable switch ASIC [63]. 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
Table 1: Additional H/W resources used by Elastic sketch, normalized by the usage of the baseline switch. P4. The total memory usage of our P4Switch is the product of the baseline usage and the additional usage.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Baseline</th>
<th>Additional usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match Crossbar</td>
<td>474</td>
<td>5.9%</td>
</tr>
<tr>
<td>SRAM</td>
<td>288</td>
<td>12.5%</td>
</tr>
<tr>
<td>TCAM</td>
<td>102</td>
<td>0%</td>
</tr>
<tr>
<td>VLIW Actions</td>
<td>145</td>
<td>5.5%</td>
</tr>
<tr>
<td>Hash Bits</td>
<td>1605</td>
<td>2.3%</td>
</tr>
<tr>
<td>Stateful ALUs</td>
<td>4</td>
<td>75%</td>
</tr>
<tr>
<td>Packet Header Vector</td>
<td>277</td>
<td>0.36%</td>
</tr>
</tbody>
</table>

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 \(\frac{\text{vote}_{\text{all}}}{\text{vote}^+} \geq \lambda\)', 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 \((f, \text{vote}^+\)) is evicted by another flow \((f_1, \text{vote}_1^+)\), we set the bucket to \((f_1, \text{vote}_1^+ + \text{vote}_{\text{all}})\). As mentioned in Section 4.2, we recommend using 4 subtables in the hardware version. In this way, we only need 4*2 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 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 [64] 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

We have implemented the software version of Elastic on three software platforms: CPU, multi-core CPU, and OVS. Implementation details are provided in the our technical report [47], due to space limitation.

### 7 EXPERIMENTAL RESULTS

#### 7.1 Experimental Setup

**Traces:** We use four one-hour public traffic traces collected in Equinix-Chicago monitor from CAIDA [65]. The details of these traces are shown in technical report. 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 10th and 90th percentile error bars across

![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 CAIDA trace. Each algorithm uses 600KB memory. The heavy part in Elastic is 150KB.](image-url)
these 720 sub-traces. We use the CAIDA4 trace with a monitoring time interval of 5s as default trace, which contains 1.1M to 2.8M packets with 60K to 110K flows (SrcIP). Due to space limitations, we only show the results with the source IP as the flow ID; the results are qualitatively similar for other flow IDs (e.g., destination IP, 5-tuple).

**Evaluation metrics:**

- **ARE (Average Relative Error):** $\frac{1}{n} \sum_{i=1}^{n} \frac{|f_i - \hat{f}_i|}{f_i}$, where $n$ is the number of flows, and $f_i$ and $\hat{f}_i$ 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 mouse flows are often over-estimated while they are in the denominator of the ARE formula, leading to large average value of relative error.
- **F1 score:** $\frac{2 \times \text{PR} \times \text{RR}}{\text{PR} + \text{RR}}$, 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 F1 score to evaluate the accuracy of heavy hitter and heavy change (HC) detection.
- **WMRE (Weighted Mean Relative Error) [12, 17]:** $\frac{\sum_{i=1}^{n} |n_i - \hat{n}_i|}{\sum_{i=1}^{n} \frac{n_i + \hat{n}_i}{2}}$, where $z$ is the maximum flow size, and $n_i$ and $\hat{n}_i$ 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):** $\frac{|\text{True} - \hat{\text{True}}|}{\text{True}}$, 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. The heavy part does not dynamically resize except for the experiments of adaptivity to traffic distribution (Section 7.4.3). 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 [66].
- **Heavy hitter detection:** We compare six approaches: Space-Saving (SS) [15], Count/CU 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 UnivMon, 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 [67], FlowRadar [18], UnivMon, and Elastic. For Reversible, we use 4 hash functions as recommended in [67]. For FlowRadar, we use 3 hash functions in both the Bloom filter [44] and the IBLT part [45]; 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 [68], and Elastic. UnivMon uses the same setting as before. We use 8 sampling groups in Sieving.
- **Cardinality estimation:** We compare UnivMon, linear counting (LC) [59], and Elastic. UnivMon uses the same setting as before.

### 7.2 Accuracy

Figure 9(a)-(c) 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).

Observed Worst Cases: Here, we show the observed worst cases of Elastic in the flow size estimation, instead of the average errors shown in the above figures. Notice that the estimation error of Elastic comes from two parts: 1) Some elephant flows are recorded in the light part due to the hash collisions in the heavy part, and this may incur overflows of counters in the light part. 2) Some flows collide at the same counter in the light part. In our experiments, over different traces, we observe that at most 2 flows have under-estimation error, and the maximum absolute error is 254 (i.e., a flow with size 1 is mapped to an overflowed counter). In each trace, there are about 110,000 flows and the maximum flow size is about 17,000. It means Elastic has small relative errors even in the worst case.

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 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.
observe that for flow size estimation, the bandwidth usage of Elastic is always less than its memory usage, consistently with Theorem A.7 in Section A.5 of our technical report [47]. The reason that the bandwidth usage does 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.

![Figure 13: Accuracy comparison of different compression and merging algorithms for CM sketch in flow size estimation.](image)

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 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.

![Figure 14: ARE and transmission delay comparisons for different sketch sizes in flow size estimation.](image)

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.

![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. The average number of heavy hitters in each traces is about 397. For more details, please refer to §6.](image)

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 [69] 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.

![Figure 16: Benefits of copy operation (§3.4) for heavy hitter detection and flow size distribution under different traffic distributions.](image)
We conduct this experiment on a server with two CPUs (Intel Xeon E5-2620V5@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.

![Processing speed comparison for six tasks on CPU platform.](image1)

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

7.5 Processing Speed

7.5.1 CPU Platform (single core).

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.

![Processing speed evaluation for Elastic in OVS.](image2)

Figure 18: Processing speed evaluation for Elastic in OVS.

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 E5-2620V5@2.4GHZ), 378GB 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.

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).

![Processing speed comparison for Elastic on different platforms.](image3)

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 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.

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 mouse flows and (2) sketch compression to improve scalability. Our sketch is generic to measurement tasks and works across different platforms. To demonstrate this, we implement our sketch on six platforms: P4, FPGA, 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.

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.

9 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and our shepherd, Arvind Krishnamurthy, for their thoughtful suggestions. We would like to thank Tao Li and Haiying Du for FPGA implementation of our algorithms based on FAST [70] platform, Yucheng Huang for P4Switch implementation, Yukai Zhao and Dongseng Yang for conducting experiments, Siang Gao for GPU implementation, Shigang Chen, Chuanxiong Guo and Minlan Yu for their valuable suggestions. This work is supported by Primary Research & Development Plan of China (2016YFB1000304), National Basic Research Program of China (973 Program, 2014CB340405), NSFC (61672061), the Open Project Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences. Steve Uhlig is partly supported by the UK’s Engineering and Physical Sciences Research Council (EPSRC) under the EARL: sdn EnAbled MeasuRement for aLL project (Project Reference EP/P025374/1).
REFERENCES


