InstaMeasure: Instant Per-flow Detection Using Large In-DRAM Working Set of Active Flows

Rhongho Jang
University of Central Florida (UCF) & Inha University
r.h.jang@knights.ucf.edu

Seongkwang Moon
Inha University
Incheon, Korea
skmoon@isrl.kr

Youngtae Noh
Inha University
Incheon, Korea
ytnoh@inha.ac.kr

Aziz Mohaisen
University of Central Florida
Orlando, USA
mohaisen@ucf.edu

DaeHun Nyang
Inha University
Incheon, Korea
nyang@inha.ac.kr

***Abstract*—In the zettabyte era, per-flow measurement becomes more challenging for the data center owing to the increment of both traffic volumes and the number of flows. Also, the swiftness of detection of anomalies (e.g., congestion, link failure, DDoS attack, and so on) becomes paramount. For fast and accurate traffic measurement, managing an accurate working set of active flows (WSAF) from massive volumes of packet influxes at line rates is a key challenge. WSAF is usually located in high-speed but expensive memory, such as TCAM or SRAM, and thus the number of entries to be stored is quite limited. To cope with the scalability issue of WSAF, we propose to use In-DRAM WSAF with scales, and put a compact data structure called FlowRegulator in front of WSAF to compensate for DRAM's slow access time by substantially reducing massive influxes to WSAF without compromising measurement accuracy. To verify its practicability, we further build a per-flow measurement system, called InstaMeasure, on an off-the-shelf Atom (lightweight) processor board. We evaluate our proposed system in a large scale real-world experiment (monitoring our campus main gateway router for 113 hours, and capturing 122.3 million flows). We verify that InstaMeasure can detect heavy hitters (HHs) with 99% accuracy and within 10 ms (detection is faster for heavier HHs) while providing the one million flows record with only tens of MB of DRAM memory. InstaMeasure's various performance metrics are further investigated by the packet trace-driven experiment using one-hour CAIDA dataset, where the target of measurement was all the 78 million L4 flows for one-hour.**

***Keywords*-Anomaly detection; traffic measurement; sketch**

I. Introduction

We are inching closer to the zettabyte era with ever-increasing volumes of traffic on the Internet. According to a Cisco's report [1], the annual Internet traffic will reach 3.3ZB per year by 2021. To deal with the rapidly surging demands on network bandwidth, per-port bandwidth now reaches 100 Gbps, or even more. To improve the utilization of the deployed network equipment (e.g., switch and router) by traffic engineering and secure networks, the role of traffic measurement becomes more important than ever, especially for data centers, where large volumes of traffic are moved between different sites or even with a single datacenter. Therefore, to enable fine-grained network traffic control, per-flow measurement (5-tuple: source IP address/port number, destination IP address/port number, and protocol) and its treatment become more crucial.

Thanks to the high-speed network traffic, measurement algorithms now have to cope with enormous incoming data rates (*i.e.,* larger number of flows) with tight deadlines (*i.e.,* real-time). We stress that instant measurement is highly necessary for the data center traffic engineering (TE) and network anomaly detection. For example, if denial of service (DoS) attack causes an influx of packets at 100 Gbps, the detection delay of 100 ms will cause 1.2GB data to hit a server or a network. Therefore, to eschew large bandwidth payment, instant anomaly detection is essential.

For per-flow measurement, sketch-based techniques have been greatly enhanced over several decades, starting with original proposals such as Flajolet-Martin (FM) sketch and Alon *et al.*'s approximate frequency measurement [2], [3] [4]–[10]. Unlike their counterparts (e.g., Netflow [11], sflow [12], jflow [13], etc.), sketch-based counting algorithms only require a small amount of memory to measure a large volume of traffic in real-time. To decrease memory usage, most works have used statistically shared counters [10], matrices [2], and Bloom filters [8] as statistical noise from each estimation can be removed at the time of estimation (or decoding). To enhance estimation accuracy, maximum likelihood estimation is usually adopted, thereby introducing a substantial amount of additional computations. Due to their designs, most of the sketch-based decoding algorithms involve hundreds of hash calculations (*i.e.,* computationally hard) and memory accesses from statistically mixed random blocks [14] to obtain meaningful statistics (e.g., heavy hitters, DDoS attack, flow size distribution and entropy, etc.) [8]–[10]. For this reason, offline decoding in a high-performance server is commonly accepted in practice but inherently incurs huge network delay. Particularly, for a software switch that is to be wildly used in a data center server, remote decoding undoubtedly increases the network congestion which degrades the user experience. Thus, online decoding is highly necessary for instant measurement and further timely detection.

To enable instant measurements, scalability, as well as online decoding of measurement algorithms, are essential. This is because sketches are quickly saturated, and cannot count anymore when a flow grows, forcing the saturated sketch to be sent to a remote collector over the network and resulting in a high detection latency. For the scalability, instead of sending out a saturated sketch to a collector, we can decode and store the value into a table in a switch (or router) for hours or even days. By doing that, a switch can always refer to the table that

keeps track of flows and their sizes. However, this approach requires not only online decoding capability of the underlying sketch, but also the scalability of the table, because our target time scale is very long—an hour to a week. Naturally, we considered the working set of active flows (hereafter, WSAF), which should be maintained by a switching fabric/software (e.g., Openswitch) for measurement and further refinements (e.g., routing, TE, and so forth). A WSAF is a type of cache of a full flow table, which can be found usually in TCAM (Ternary Content Addressable Memory), CAM, or sometimes SRAM for fast switching (or forwarding) [15]. NetFlow uses TCAM for storing WSAF in which an entry consists of a flow ID and the counting value, while OpenSketch takes advantage of TCAM and SRAM [11], [16]. The number of entries in the table cannot be large because those types of memories are quite expensive. To support scalability by increasing the WSAF's capacity, we can put WSAF in DRAM instead of the expensive memory (*i.e.,* incentive to cost-effectiveness). However, there is a speed issue for In-DRAM WSAF: a packet arrival rate is too fast to handle by In-DRAM WSAF, owing to the DRAM's speed and WSAF table's hash collision.

Unfortunately, most sketch-based algorithms lack scalability and online decoding capabilities. Our approach to solving these two problems is 1) to use a counting algorithm that can perform online decoding and 2) to put a flow regulator before WSAF to slow down the incoming packet rate to WSAF. To realize both ideas, we designed a highly scalable counting and flow regulating algorithm called FlowRegulator. By design, instead of directly inserting or updating every packet of flow into the WSAF table, FlowRegulator (*i.e.,* a small cache buffer) retains a fraction of flow counts. By doing so, we can suppress frequent WSAF updates in DRAM; thereby FlowRegulator can support the large-scale influx of flows with the use of cost-effective large DRAM. Consequently, FlowRegulator relaxes the necessity of precious memories (TCAM or SRAM) for maintaining large WSAF, and further enables us to build a highly scalable and fast measurement system. We realize that in a system called InstaMeasure, and deliver the following contributions:

- We design FlowRegulator to overcome the lack of scalability and online decoding capabilities. To verify its practicability, we further build a large-scale per-flow measurement system called InstaMeasure (Section III).
- To show InstaMeasure's feasibility and practicability, we implemented prototype of InstaMeasure using an off-the-shelf Atom processor board, and extended InstaMeasure to a multi-core measurement system (Section IV).
- We evaluate the performance of InstaMeasure in several scenarios. First, we evaluate the estimation accuracy and processing speed of InstaMeasure with 78 million L4 flows in one-hour CAIDA dataset by varying parameters (e.g., memory usage, the number of cores, packet per second, *etc.*.). Second, we conduct a real-world campus network experiment for 113 hours by connecting InstaMeasure to a mirroring port of the main gateway router, capturing 9.11 billion packets, 122.3 million flows, and 8.5TB bytes. InstaMeasure successfully measured the whole L4 flows both in packets and in bytes where the standard errors of both estimations were smaller than 0.65%. As one key application, InstaMeasure detected heavy hitters with 99.8% accuracy within 10 ms in the worst case—the prefix *Insta* comes from this tight time-bound (Section V).

II. Motivation

Our large WSAF in DRAM is in contrast to the small WSAF in TCAM (*i.e.,* industry practice), which uses a central collector over the network at high rates. In DRAM, we can store much more flows; thereby, we do not need a remote collector for decoding. However, the downside is that we cannot evade the "sluggishness" of DRAM.

Managing a WSAF at packet arrival rate: DRAM's access speed is limited to process packets arriving at a line rate (e.g., 40 or 100 Gigabit Ethernet), so today's online measurement algorithms assume fast but expensive SRAM for processing sketches. Due to SRAM's prohibitive cost, only tens of megabytes are available to a counting algorithm [16]. Thus, instead of storing all the information of flows in SRAM, a measurement algorithm stores only a sketch or a summary in SRAM that does not have flow information (*i.e.,* flow ID and its 5-tuple). A set of flow IDs in a table, a mapping between a sketch and a flow, or even a reversible sketch during a measurement period are normally stored in DRAM. This use of DRAM is necessary and common in practice [10], [16], but managing flow IDs are quite challenging, and insertion-per-second (hereafter, ips) to the structure should be as high as packets-per-second (hereafter, pps). Also, in NetFlow, there exists a WSAF table in which ips should be high enough to process pps at a line rate. Under the constraint where $\{\text{ips} = \text{pps}\}$ (insertion and lookup at WSAF should be done at packet arrival rate), it is hard for WSAF to keep up with the speed if the traffic increases. Packet sampling might be a viable option, which is used by NetFlow, SFlow, and many sketch-based schemes. However, such an approach degrades the estimation accuracy essentially. NetFlow uses both sampling and TCAM to ensure speed, but the most popular switching silicon chips have tables that can hold only up to thousands of route entries in TCAM and CAM [15], which cannot support a large-scale WSAF for instant measurement.

FlowRegulator to relax the $\{\textbf{ips} = \textbf{pps}\}$ constraint: Instead of using TCAM or SRAM, we can use DRAM for WSAF by relaxing the ips requirement for the WSAF table. Thus, instead of directly inserting or updating every flow packet into the table, we put a small buffer called FlowRegulator to retain a fraction of flow counts before WSAF. FlowRegulator has a memory block (or a virtual vector initialized to all 0's) for every single flow, and whenever a packet comes in, the corresponding block is updated by setting a random bit of the block. When the block saturates (or a portion of the block has set to 1's), the resulting counting fraction (we note that this is not the total size of flow) is added up to the WSAF

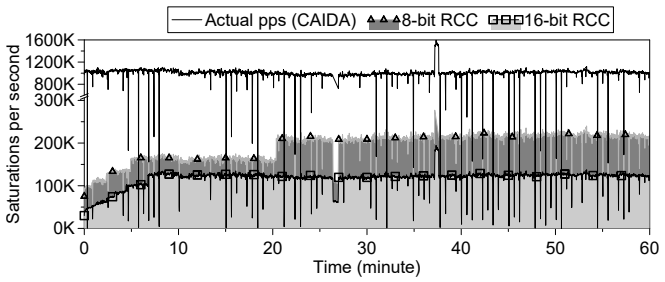


Fig. 1. RCC's saturation occurs in the speed of 12-19% of packet arrival rate (the black solid line), which is too frequent to compensate for SRAM's speed margin over DRAM's (5-10%) in CAIDA dataset.

(*i.e.,* a hash table in DRAM). Because FlowRegulator retains mice flows whose sizes are lower than the saturation condition, not all the packets are fed into WSAF, but only the packets that trigger the saturation condition are given to WSAF. This design greatly reduces ips even under a high pps condition.

How to build FlowRegulator: To develop FlowRegulator, we utilize sketch-based counting algorithms, because they can encode packets at line rates, and can accurately estimate flows with a small amount of memory. Additionally, they satisfy our requirements: *online decoding* for adding up to WSAF when the block is saturated and *scalability* to deal with a large number of flows. A hitherto known solution is RCC proposed by Nyang and Shin [17] because it already has online decoding capability, and proven to be useful for measurement in the wireless SDN environment [18]. To investigate its feasibility, we have tested RCC for its rate regulation (defined as Output ips/Input pps). Given that the access time of SRAM is 10-20 times faster than DRAM's (and even faster with TCAM), RCC's rate regulation should be less than 5%. However, its regulation and retention capacity (the maximum number of packets in a virtual vector) are not operationally sufficient. To show that, we conducted an offline experiment using a CAIDA dataset [19]. As shown in Fig. 1, the solid line shows the actual packet arrival rate in pps, which is 1 mpps (million packets per second) on average, but RCC's saturation frequency is around 19% (output rate is about 190 kips (thousand ips) for the 8-bit vector, and 12% for 16-bit vector, which is far higher than the speed margin of SRAM over DRAM. Thus, it is impossible to work with RCC for building FlowRegulator. One way to increase the rate regulation is to give RCC a larger virtual vector, but that does not expand the retention capacity. This will further be investigated in evaluation Section V.

Two-layer design for higher rate regulation: Here, our observation is that enlarging the virtual vector size increases the retention capacity just in an addictive manner, and thus, this is not a viable (*i.e.,* scalable) option. Instead, we designed a new counting algorithm for FlowRegulator, which has two layers of probabilistic counters to achieve the higher rate regulation. Note that the multi-layer sketch is not first introduced by this paper (e.g., [20]), but the only sketch-based data structure that supports online decoding. Our FlowRegulator plays a key role in retaining flows (from feeding into WSAF) for a while as well as counting flows. In the two-layer design, the second (higher) layer's one bit encodes multiple packets of a flow from a saturated sketch of the first (lower) layer. This design has substantially improved the rate regulation in a multiplicative manner. It enables higher rate regulation while not being detrimental to accuracy and speed while being scalable.

Saturation-based decoding for flows: Another aspect of FlowRegulator is counting elephant flows. Whenever a packet comes in a virtual vector, the estimation of the saturated vector is calculated by online decoding, and if saturated, the decoded counting value is finally accumulated to WSAF. This is called "saturation-based decoding" in contrast to "packet-arrival-based decoding". The latter is for actual online counting, and obviously, it is not feasible because of memory and computation speed. Saturation-based decoding has the property that it allows the only elephant flows (flow sizes greater than retention capacities of the sketch) get through FlowRegulator to reach the WSAF table, which prevents WSAF from exploding from a huge number of incoming mice flows. This is in contrast to NetFlow, which registers every flow, if not sampled, in the table regardless of its size. Owing to this, WSAF can keep the counters only for active elephant flows, which means FlowRegulator helps to maintain a WSAF with good quality. Notably, even though our FlowRegulator filters mice flows well, there are still mice flows that get through to WSAF (recall that FlowRegulator is a probabilistic counter). We note, however, that it is essential for some applications to have samples of mice flows (e.g., DDoS attack, SuperSpreader and entropy *etc.*). However, WSAF needs to evict the expired (or least significant) mice flows when the table is full. For FlowRegulator, instead of running a separate core periodically (NetFlow approach), when a new flow is inserted, and an empty slot is searched by hash chaining, garbage collection is performed. Using our WSAF in DRAM, we can also analyze flow behavior for long-term measurement. Considering that other sketch-based schemes send a sketch and flow ID information periodically to a remote collector for sketch decoding, the decoding can be regarded as a "delegation-based decoding". Comparing the three different approaches, namely the delegation-based, the packet-arrival-based (used as ground truth and a baseline), and the saturation-based decoding, we note that the packet-arrival decoding has the fastest detection time. However, the time difference between packet-arrival-based and the saturation-based decoding is within 10 ms, while the difference between packet-arrival-based and delegation-based decoding is tens of milliseconds (may increase depending on network delay). Therefore, our saturation-based decoding is substantially faster than delegation-based decoding.

III. FlowRegulator Design

Today's Internet traffic follows a Zipf-like distribution [21], and mice flows (e.g., 1-10 packets flows) are the majority of network flows, which is the main reason for WSAF cache saturation. The DRAM is relatively cheap; thus we have fewer

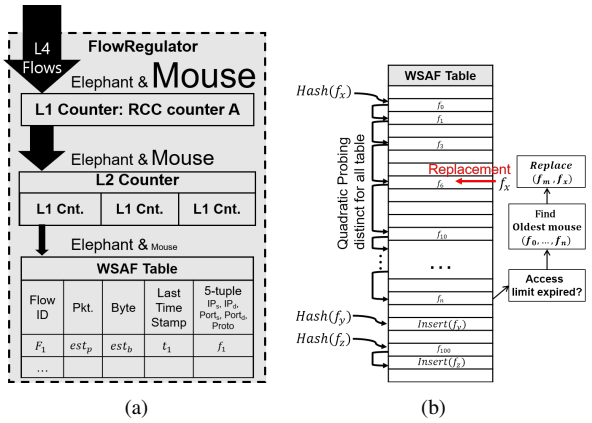


Fig. 2. Design of FlowRegulator: (a) Components of FlowRegulator (b) Probing limit-based second-chance replacement policy of WSAF Table

Algorithm 1: Two-layer FlowRegulator

Init L1[ ]
Init L2[$Noise_{min}$][ ],...,L2[$Noise_{max}$][ ]
forall Pkt_f **do**
  $(idx_f, vv_f) \leftarrow$ Hash(Pkt_f)
  $Noise_{L1} \leftarrow$ RCC_Encode(L1[idx_f], vv_f)
  if $Noise_{L1} \neq NULL$ **then**
    /*vv_f saturated in L1*/
    $Noise_{L2} \leftarrow$ RCC_Encode(L2[$Noise_{L1}$][idx_f], vv_f)
    if $Noise_{L2} \neq NULL$ **then**
      /*vv_f saturated in L2*/
      $unit \leftarrow$ RCC_Decode($Noise_{L1}$)
      $est_{pkt} \leftarrow unit \times$ RCC_Decode($Noise_{L2}$)
      $est_{byte} \leftarrow est_{pkt} \times$ Length(Pkt_f)
      ACC$_{WSAF}(f, est_{pkt}, est_{byte})$
    end
  end
end

constraints on its use, compared to SRAM and TCAM. To overcome its slow read/write access time, we designed a sketch-based FlowRegulator to regulate influx rates of packets in front of WSAF by retaining mice flows until they overflow (or saturate) sketches that they reside in. Note that most mice flows do not grow enough to overflow their sketches.

A. Two-layer sketch-based counter

Fig. 2(a) illustrates our design of FlowRegulator. The L1 counter is a sketch-based data structure introduced in RCC (Recyclable counter with confinement [17]). The authors of RCC proved that a small virtual vector (8-bit) provides a higher estimation accuracy. A major problem, however, is that if we use RCC for FlowRegulator, the 8-bit virtual vector can only count up to 9 packets in the best case. That means the structure can retain mice flows up to 9 packets and insertion operations of an elephant flow occur every 9 packets. This rate regulation is still not acceptable for In-DRAM WSAF: Fig. 1 of RCC's flow regulation rates for two vector sizes shows the vector size increment, which does not effectively increase the regulation rate. To address this problem, we use a two-layer sketch strategy to increase FlowRegulator's retention capacity significantly by designing the second layer sketch to count in multiple units of the first layer sketch. This multiplicative approach enables FlowRegulator to retain larger mice and to retain more packets of each elephant flow (up to around 100 packets for a single flow—10 times more than that of RCC).

As shown in Fig. 2(a), the L2 counter is a set of L1 counters. We categorized L1's estimation into three cases based on the noise level (*i.e.,* relevant to the number of bits set to 0). This is, for an 8-bit virtual vector, a single flow can set at most three bits (*i.e.,* 70%) of the 8-bit virtual vector to 1's; thus the estimation can be divided into three cases. We use those three different estimation values as the units of three counters in layer-2. For example, when the estimation of L1 is 5, the counter of unit 5 in L2 is chosen, and only one bit of the counter is set. If the estimation of the counter of unit 5 in L2 was 4, the total counting value would be 20 (=5×4). The encoding and decoding processes of L2 counters are designed to be the same as that of L1, and even the memory layout and the virtual vector's bit positions of every flow are the same (hash function reuse of L1 virtual vector). Thus, L2 counting only requires one additional memory access (in total, two memory accesses and one hash including L1 counting). By doing this, we obtained around 1.02% flow regulation rate; thus the insertion request rate to WSAF table could be reduced substantially (See Section V).

B. WSAF table management

Our FlowRegulator can retain most mice flows, but not all of them. There still is a probability for mice flows to pass through FlowRegulator and to be inserted into the WSAF table owing to noise. These mice flows lead to memory space wastes and frequent hash collisions (*i.e.,* probing of active flows increases). We address this problem by using a probe limit-based and second-chance replacement algorithm to evict mice flows from WSAF table to save memory space and increase probing speed. Moreover, the probe limit-based approach allows us to use specific parameters (*i.e.,* table size $m = 2^n, h(k,i) = hash(k) + 0.5i + 0.5i^2 \ mod \ m$) for probing all table positions in $[0, m-1]$ to achieve a high load factor. See Fig. 2(b).

C. Byte counter

InstaMeasure has another desirable feature that provides packet and byte counting at the same time. Based on the packet counting technique, we utilize a sampling-based approach to perform byte estimation. When a flow f saturates FlowRegulator, an estimated packet number (est) will be accumulated to WSAF table using the f_{id}. We use the size of the last arrived packet len to multiply with est and accumulate $len \times est$ to the byte counting field of WSAF table. Even though the idea is straightforward, it works quite accurately (< 1% error rate, see Section V.B) and efficiently (one extra multiplication).

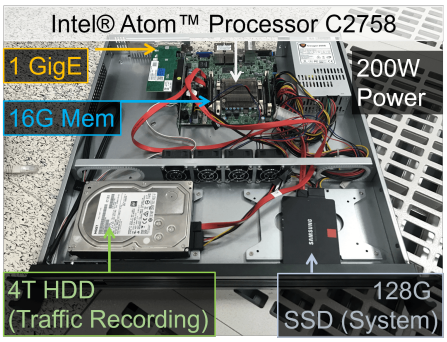


Fig. 3. InstaMeasure as a measurement device

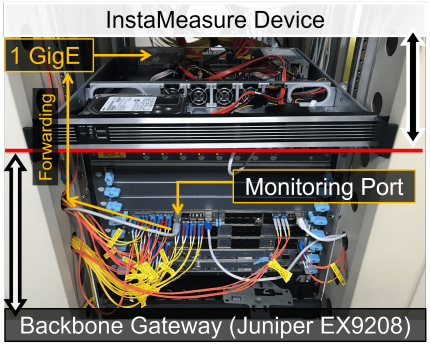


Fig. 4. Configuration of real-world experiment

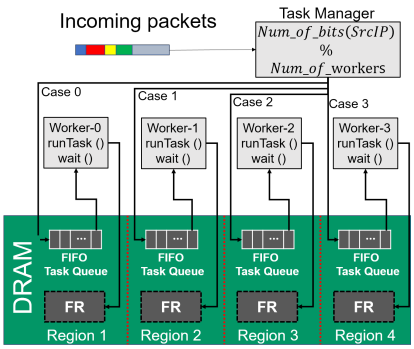


Fig. 5. Multi-core flow regulation

D. Algorithm

L1 counter of FlowRegulator has a simple word array structure, where the size of each word is selectable (32 or 64 bits depending on processor). When a packet arrives from flow f, FlowRegulator computes a hash function using 5-tuple extracted from the packet (line 4). The hash value is used for two purposes, 1) to extract virtual vector vv_f (*i.e.,* bit positions confined in a word—virtual vector confinement technique as in [17]), and 2) to determine vv_f's word location (idx_f) at L1 counter (L1[idx_f]). Once idx_f and vv_f are decided, `RCC_Encode` performs encoding of the sketch until vv_f of L1[idx_f] saturates and returns a noise level ($Noise_{L1}$) (line 7). L2 is a set of L1 counters. When the saturation happens in L1, one of the counters in L2 will be selected depending on $Noise_{L1}$ to perform second layer counting using the same idx_f and vv_f (line 9). When vv_f is saturated in L2, FlowRegulator estimates the total packet number (est_{pkt}) by multiplying `RCC_Decode`($Noise_{L1}$) and `RCC_Decode`($Noise_{L2}$), where the former is the number of packets at L1 at the saturation moment, and the latter is the frequency of saturation at L2 (lines 14-15). The estimation of byte volume (est_{byte}) is done by the saturation-based sampling approach. That is, the byte volume is calculated by multiplying est_{pkt} with the size of the packet that triggered the L2 saturation (line 15). Finally, FlowRegulator accumulates est_{pkt} and est_{byte} to the WSAF table using flow ID f (line 16) either by insertion or by update.

IV. Implementation

We prototyped InstaMeasure in an off-the-shelf device with 8-Core Atom processors. The estimation accuracy and the processing speed of InstaMeasure were evaluated by a packet-driven experiment using 1-hour CAIDA dataset (1-4 cores used). Further, we set up a real-world experiment using InstaMeasure device at the backbone gateway router of our campus network for 113 hours autonomously and ran a use case: heavy hitter detection (1 core used).

A. Hardware description

Fig. 3 shows the hardware setup of our InstaMeasure device. We used a Supermicro motherboard A1SRi-2758F that embeds 8-Core Intel Atom processor C2758 ($312) which has a 4MB cache memory (448KB for L1 cache and 4096KB for L2 cache). In total, 16G (2x8G) DDR3 1600MHz memory was used with a 200W power supply. We used a 128G SSD for running Linux 16.04 server (x86) and 4T HDD to record the network trace for offline analysis. For fast packet processing, we implemented InstaMeasure based on DPDK (version 17.11.2) to bypass the kernel. Note that our choice of the CPU is reasonable as Atom series CPU appears in many modern routers/switches including bare metal switches [22].

B. Real-world experiment setup

Our campus uses 2 Gbps bandwidth in total (1 Gbps for up-link and 1 Gbps for downlink), and the backbone gateway router uses a Juniper EX9208 switch, as shown in Fig. 4. Since, for logistical reasons, the gateway could not be programmed for this experiment, we used the mirroring port of the gateway to perform our measurement. The purpose of this experiment is to check InstaMeasure's performance (CPU and memory use) and scalability (accuracy for 113 hours) (See section V.D for results). We also ran a use case of heavy hitter detection. Because the mirroring port starts to drop packets when port capacity is exceeded, the estimation accuracy was evaluated by comparing results of InstaMeasure to results obtained by the recorded traffic experiencing the same packet drop. Due to the policy of our school, we were permitted to access only the up-link although for a long time. Moreover, we evaluated the processing speed and heavy hitter detection delay using the CAIDA dataset and artificially-generated traffic, to cope with non-deterministic mirroring delays caused by port buffering in our real-world experiment.

C. Multi-core traffic measurement system

To perform faster encoding and decoding by taking advantage of the multi-core Atom processor, we implemented InstaMeasure as a multi-core traffic measurement system. Fig. 5 shows a case of the four-core model. As shown, we allocate memory blocks exclusively to each worker core to avoid memory collision, where each worker core maintains an independent FlowRegulator structure with a FIFO task/packet queue. A worker continuously monitors its task queue and performs encoding and (if necessary) decoding whenever each

packet arrives. An additional manager core is responsible for allocating packets to a worker's queue. To evenly distribute packets to be processed, the number of 1 bit of source IP address is used to determine which queue the packet goes into. As will be shown in Section V.C, InstaMeasure scales based on the number of core.

D. Parameters

The main component of FlowRegulator is the two-layer counter. To construct FlowRegulator, we used a total of four small counters, one for L1 and three for L2 as described in section III. Thus, when we use a 32KB L1 counter, the total size of the two-layer counter is 128KB. Moreover, in the multi-core system, the total memory usage is M times of the number of worker cores, where M is the memory allocated to the L1 counter. For the four-core system, the allocated memory will be 128KB×4=512kB.

In a lab experiment, we evaluated the accuracy of a single core FlowRegulator using the CAIDA dataset by varying the memory usage of the L1 counter from 32KB to 512KB (in total, we had 128KB-2048KB when including the three L2 counters for FlowRegulator). In the real-world experiment, we used 128KB of memory with a single core worker. FlowRegulator's processing speed was shown to be fast enough to process 10 Gbps link (see section V). For the memory usage of the WSAF hash table, we fixed the total entry numbers to 2^{20} for all experiments including the multi-core case.

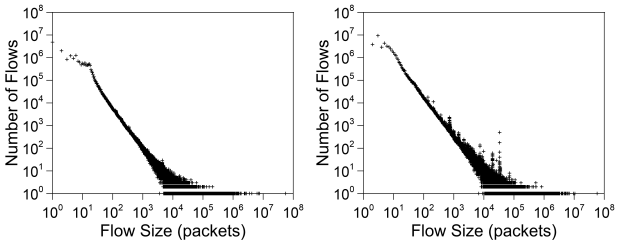
As shown in Fig. 2(a), the size of each hash table entry is 33 bytes to include a flow ID (32 bit hash of 5-tuple), packet counter (32 bits), byte counter (32 bits), timestamp (64 bits) and the 5-tuple (104 bits). Thus, the total DRAM space required for the hash table is only 33MB. If we allocate more DRAM, e.g., 1GB, it can run for several days autonomously and without interruptions on a 10 Gbps link.

V. Evaluation

First, we evaluate the estimation accuracy and processing speed of InstaMeasure with the CAIDA dataset by varying parameters (e.g., memory usage, the number of cores, pps (packets-per-second), ips (insertion-per-second) *etc.*). Second, we demonstrate the feasibility of InstaMeasure by showing results of real-world experiment.

A. Datasets

- **CAIDA Anonymized Internet Trace 2016.** [19] We used one-hour (13:00-14:00, 6th of April, 2016) network traffic trace that was collected at the Equinix-Chicago data center on an OC-192 link (maximum load of 10 Gbps). We merged trace data of both directions (*i.e.,* between Chicago and Seattle) in the order of timestamp to evaluate InstaMeasure with larger-scale network trace. As a result, our dataset contains 3.7 billion IPv4 packets (include UDP, TCP, and ICMP), 78 million L4 flows, and the highest speed was 1.5 mpps (million pps). This scale is substantially large and beyond current sketch-based measurement's capability. See Fig. 6(a) for the traffics distribution of the dataset.
- **113-hour backbone gateway traffic on campus network.** We implemented our InstaMeasure in an off-the-shelf device and measured up-link traffics (1 Gbps bandwidth) at the backbone gateway (Juniper EX9208 switch) of our campus for 113 hours in total. For further analysis, we also recorded 5-tuple, the packet size and the timestamp of every single packet. In total, about 8.5TB of traffic, 9.1 billion packets (broken down into 6.4% of UDP and 93.6% TCP) and 122.3 billion L4 flows were observed in 113 hours. See Fig. 6(b) for the distribution.



(a) CAIDA (b) Backbone gateway of Campus

Fig. 6. Distribution of CAIDA dataset and 113 hours campus traffics.

B. Evaluation of FlowRegulator

WASF ips relaxation. In Fig. 7, the x-axis represents the timeline of our merged CAIDA dataset, and the solid black line on the top represents the actual pps of the trace. Below the pps line, RCC's and FlowRegulator's regulation rates are shown in red squares and blue diamonds, respectively. The figure shows that RCC relaxes ips to feed packets to WSAF table at the speed of 112 kips (thousand ips), which corresponds to 12% regulation rate. FlowRegulator effectively regulated flows to pass only 1.02% with 128KB DRAM memory, Considering that WSAF is usually stored in SRAM or TCAM, and SRAM is 10-20 times faster than DRAM, FlowRegulator has sufficient margin, while RCC does not have as can be seen in Fig. 7. Even for WSAF in TCAM, which is faster than SRAM, FlowRegulator can be configured to have enough margin by adjusting the vector size or even the number of layers.

Regulation rate vs. sketch size. Because FlowRegulator's role is to slow down the insertion request rate to WSAF, we evaluate how effectively it achieved this goal. Fig. 8(a) shows comparatively the retention capacity of each virtual vector by varying its size. For RCC, the growth rate of the retention capacity is very slow; thus its retention capacity is only 77 packets even with a 64-bit virtual vector. We note that to use 64-bit virtual vector the confinement size should be at least 256 bits, which incurs 8 memory accesses and 8 hash computations for every packet in a 32-bit system, which is not acceptable for FlowRegulator. Compared to RCC, FlowRegulator's retention capacity grows very quickly as the size increases, and thus a 16-bit vector (8 bits for each layer) is enough to retain a hundred flows. To fairly compare FlowRegulator of two layers to RCC of a single layer, FlowRegulator's vector size is

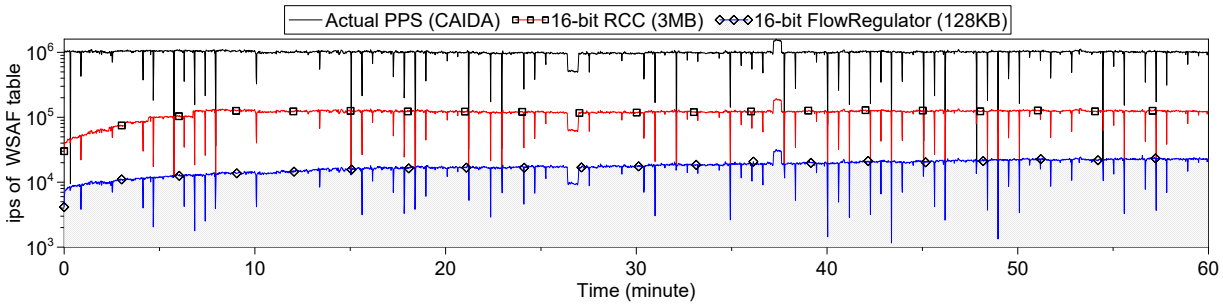
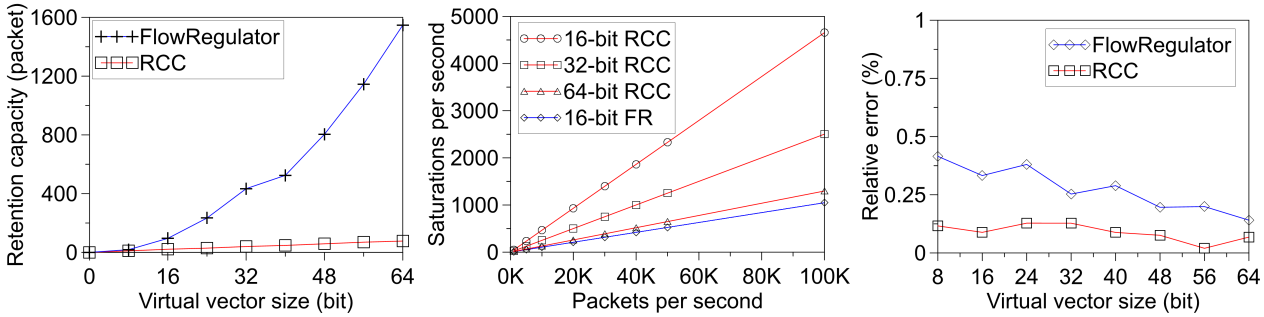


Fig. 7. WASF relaxation: FlowRegulator (FR) and RCC ips of CAIDA dataset

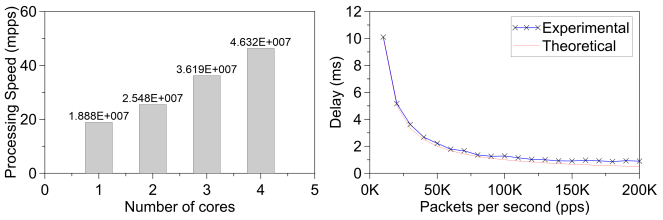


(a) Flow retention capacity (b) Saturation frequency (c) Decode error per saturation

Fig. 8. FlowRegulator's retention capacity and saturation frequency outperforms RCC's, paying a little degradation of accuracy.

defined to include all the vectors where a packet can reside—since we are interested in the number of packets retained by a virtual vector. Since FlowRegulator's design has two layers, it would be twice of L1 counter's virtual vector size. Fig. 8(b) shows the saturation frequency of a sketch for a single flow comparatively, which indicates that the insertion request rate to WSAF is decreased (better for WSAF) as the frequency becomes low. The figure shows that RCC with 64-bit virtual vector seems to be barely comparable to FlowRegulator, but it is impractical as we mentioned above. Also, in the real world, a sketch accommodates a large number of flows, so the saturation rate is much higher than that in the analysis as shown in Fig. 7. Thus, even a larger vector for RCC should be utilized. Consequently, as shown in Fig. 7, FlowRegulator provides enough retention capacity to suppress the insertion request frequency, which cannot be achieved by RCC.

On cost. Two-layer design of FlowRegulator, however, pays a small penalty of accuracy degradation, which is shown in Fig. 8(c). The overall accuracy of FlowRegulator is lower than that of RCC with a single layer, but the difference is very small except when the vector size is 8 bits (4 bits for each layer). We note that FlowRegulator implementation for all the experiment has 16-bit long vector. Another cost might be the detection latency: because FlowRegulator relies on sketch saturation-based decoding, an event such as heavy hitter cannot be detected immediately, but when the flow is registered in the WSAF table. This, in turn, delays the detection. However, as shown in Fig. 9(b), the delay is less than 10 millisecond, which is negligible compared to tens of milliseconds of delay in most

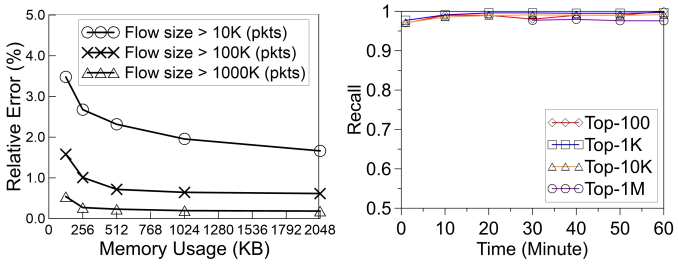


(a) Speed of FlowRegulator (b) Heavy hitter detection latency

Fig. 9. InstaMeasure's processing speed scales well, and its detection latency of heavy hitters is under 1 ms if a heavy hitter consumes more than 100 kpps.

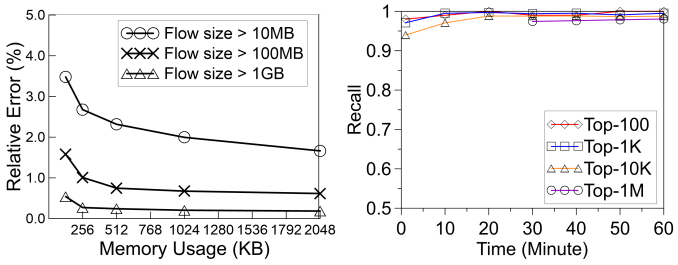
frameworks (e.g., [23]). Also, in the same figure, we draw that significant attackers use more bandwidth, and thus can be caught earlier than slow attackers, who are less important in volume-based attacks.

Processing speed of FlowRegulator. To evaluate the encoding speed of FlowRegulator, we used our off-the-shelf device in Fig. 3; it is equipped with an 8-core 2.4 GHz Atom processor and 16G DRAM. We pre-loaded the CAIDA dataset into memory and focused on how many packets FlowRegulator can process per second. Fig. 9(a) shows the processing speed of FlowRegulator by varying the number of cores. As shown, FlowRegulator could process 18.88 mpps (on average) with a single core. Clearly, a one-core FlowRegulator can measure the OC-192 link of the CAIDA dataset even when the traffic is 64-byte packets, The processing speed with 2 cores increased to 25.48 mpps. Three and four core FlowRegulator still achieved higher processing speed: 36.19 mpps and 46.32 mpps, respec-



(a) Error rate vs. memory usage (b) Quality of packet Top-K list

Fig. 10. Accuracy of packet counting (CAIDA one-hour trace)



(a) Error rate vs. memory usage (b) Quality of byte Top-K

Fig. 11. Accuracy of byte counting (CAIDA one-hour trace)

tively. We note that FlowRegulator's memory usage does not affect processing speed but only on the accuracy.

In conclusion, this experiment shows that FlowRegulator—even using an Atom processor and DRAM– has enough processing speed that can be sufficiently used for 10 Gbps high-speed links without any packet loss.

Detection latency. We conducted an experiment to show the heavy hitter detection delay caused by our FlowRegulator's saturation-based decoding in a 1 Gbps network environment. We used a high-end desktop to generate traffic with various speeds (10-200 kpps) to InstaMeasure device. At the same time, our device performed heavy hitter detection in parallel. A fixed threshold (T=0.05% of link capacity) was used to detect heavy hitters and recorded the first detected time using both packet-arrival-based and saturation-based decoding. As shown in Fig. 9(b), when the traffic generator was in a low transmission rate, the detection delay was more than 10 ms. However, as the transmission rate increased, the detection delay decreased sufficiently. When the speed was 10 kpps, the average delay was around 10 ms and 1 ms at the rate of 130 kpps. Note that byte volume-based heavy hitter detection delay is almost the same as with the packet counting-based one. This is mainly because our byte volume counting depends on the packet counting.

C. Accuracy of packet and byte counters

We used the one-hour CAIDA dataset and ran a single core InstaMeasure to evaluate the estimation accuracy (packets and bytes) while varying the memory usage of our L1 counter (*i.e.,* 32KB-512KB). Then, we compared each estimated flow size (both in packets and in bytes) with the ground-truth. Since InstaMeasure can measure a flow larger than a million packets, we divided flows into three intervals depending on the size and evaluated the average error of each interval.

Packet counter. Fig. 10(a) shows the averaged error rates of all L4 flows of the packet counter after one-hour measurement. When the total memory usage was 128KB, the average error rate of flows that have more than 1000K packets was 0.56% and 1.54% for 100K+ flows. For relatively small flows (10K+ flows), it was 3.48%. As shown in the figure, it decreased as more memory was used. When we increased the memory to 256KB, InstaMeasure achieved 0.28% of average error rate for 1000K+ packet flows, 0.99% for 100K+ flows and 2.79% for 10K+ packet flows. Further, when the amount of memory was 2048KB, InstaMeasure achieved the highest accuracy, with 0.19% (1000K+), 0.58% (100K+) and 1.76% (10K+) error rates, respectively.

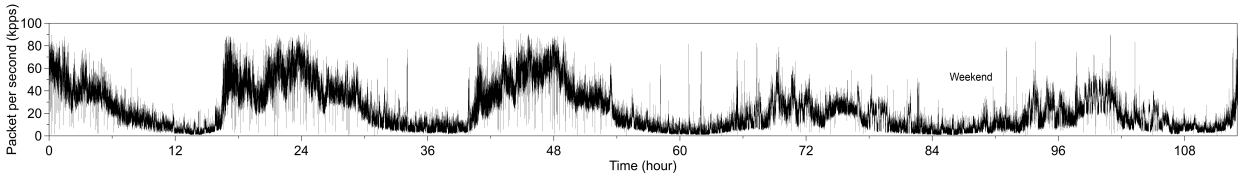
Byte counter. Fig. 11(a) shows the averaged error rates of all L4 flows of the byte counter. When the memory usage was 128KB, the average error rate of 1GB+ sized flows was 0.54%, 1.57% for 100MB+ sized flows, and 3.47% for 10MB+ sized flows. Same as with the packet counter, the accuracy of the byte counter also increased when more memory was given. For 128KB memory, the average error rates were 0.27%, 1.00%, and 2.67% respectively. For 2048KB of memory, InstaMeasure achieved 0.18% error rate for 1GB+ sized flows, 0.61% for 100MB+ sized flows and 1.66% for 10MB+ sized flows.

Top-K identification. Owing to InstaMeasure's high accuracy for millions of flows, Top-K identification problem can be scaled up to Top-million. Moreover, InstaMeasure can provide two kinds of Top-K flow lists at the same time: Packet Top-K and Byte Top-K. For evaluation, we fixed the memory usage of the counter to 10MB and used a standard recall metric to measure the quality of packet number-based and byte volume-based Top-100, 1K, 10K and 1M lists using the CAIDA dataset, with updates done every 10 minutes. Fig. 10 and Fig. 11 show that the recall rates of byte/packet Top-K are mostly above 95%.

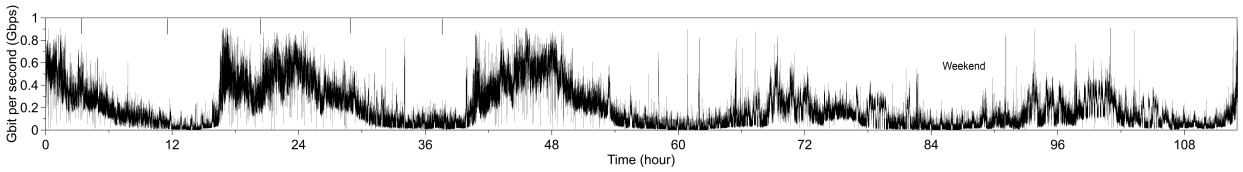
Comparison. We also report that we conducted an experiment with CSM [10] using 60MB, which corresponds to around two times of the largest memory used in InstaMeasure. The vector size of 10,000 was chosen to be large enough to count the maximum flow size. However, decoding the entire dataset did not terminate, and we failed to conduct experiments on one-hour data. Instead, we ran CSM over one-minute data and checked the accuracy. Instead of decoding all flows, we limited our decoding to the top-100 and top-1,000 flows and checked the accuracy. We found that the average error rate was 2.4% for the top-100 and 8.53% for the top-1,000, which is much higher compared to InstaMeasure using even one-hour data.

D. Monitoring in the wild

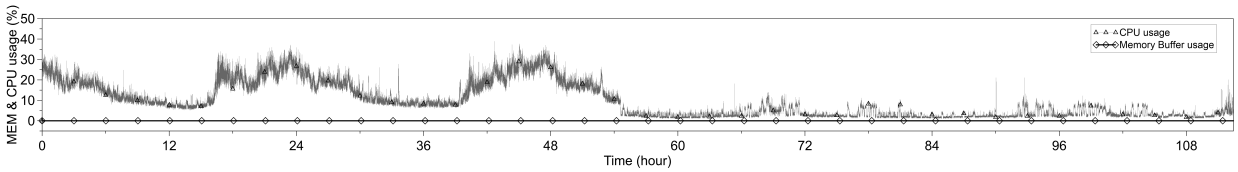
We observed that the traffic collected in our campus for 113 hours had the typical Zipf-like distribution as other network traces did. During 113 hours, 9.1 billion packets of 122.3 billion L4 flows were measured simultaneously both in packets and in bytes. InstaMeasure used a single Atom processor



(a) Packet per second (kpps)



(b) Gigabit per second (Gbps)



(c) CPU load of 1 core InstaMeasure

Fig. 12. Monitoring in the wild

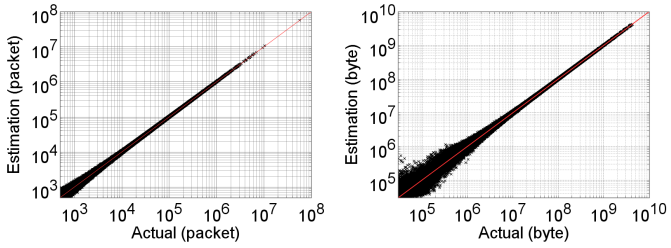


Fig. 13. Estimation result of 133 hour real-world experiment using 12MB sketch. Accuracy of packet counting (left) and byte counting (right). Each point stands for each flow. To see how accurate estimation is, check how close every point is to the reference line $y = x$.

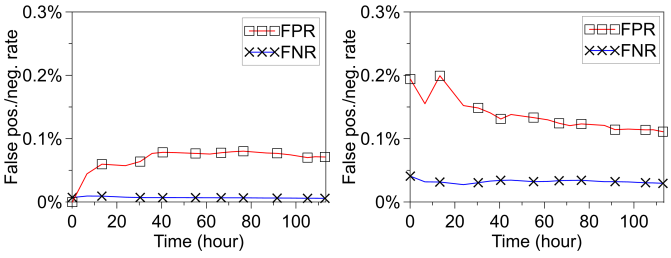


Fig. 14. False positive and false negative rates of packet heavy hitter detection (left) and byte volume heavy hitter detection (right).

core, 128KB for the sketch, and 33MB for the WSAF table. Sketches and WSAF table are all in DRAM.

Accuracy. Fig. 13 shows the estimation accuracy by standard error for the real-world experiment. For packet counting, we report 0.54% standard error over 350 flows of which size is 1000K+, 1.61% over 11,047 flows for 100K packets, 3.46% over 104292 flows for 10K+ packets. For byte counting, we report 0.63% over 414 flows of which byte size is 1G+, 1.74% over 12,125 flows of 100MB+, 3.65% over 107,726 flows of 10MB+. This accuracy matches the accuracy observed in the lab experiment with the CAIDA dataset.

Overheads. Our campus network's traffic volume is shown as a time series in Fig. 12(a). We observed that the amount of traffic reached a peak during the daytime, whereas less traffic was observed at the weekend and night. InstaMeasure's CPU workload and the queue memory usage during the 113 hours are shown in Fig. 12(c). the core's workload matches the traffic pattern, and the core usage did not go over 40% at any point. As for the queue (represented in black diamonds in the figure), it did not grow noticeably. The results confirmed that InstaMeasure implemented on Atom board worked well for the 1 Gbps network monitoring, and for a quite long time.

Heavy hitter detection. Fig. 14 shows InstaMeasure's heavy hitter detection accuracy in terms of false positive/negative rate. Owing to InstaMeasure capability of counting both in packets and in bytes, it can detect both packet heavy hitters and byte heavy hitters. False negative rates in both cases are negligible, and the false positive rates of packet/byte heavy hitters are less than 0.1% and 0.2%, respectively.

VI. Related Work

A large volume of works on sketch-based measurement have been done to leverage its estimation accuracy for traffic engineering and anomaly detection [5]–[10], [17]. Among them, Estan and Varghese's work was on heavy hitter detection during a measurement period [4], which was followed by several other works [5], [24]–[26]. Recently, Basat *et al.* proposed an elephant-flow identification and a Top-K counting

algorithms [27], [28]. Their Top-K is quite limited (up to top-512). InstaMeasure is concerned with the larger scale of Top-K, e.g., tens of thousands to millions.

Notable works on real-time measurement system include OpenSketch, which utilized various sketches and specialized hardware: e.g., TCAM and SRAM [16]. FlowRadar, which took advantage of a recently proposed hash data structure called IBLT (Invertible Bloom Lookup Table) to resolve the hash collision problem [23], [29], and UnivMon, which uses a single universal sketch [30]. Especially, FlowRadar's view on WSAF is similar to InstaMeasure, although it tried to solve non-deterministic insertion time by IBLT's constant time insertion, instead of relaxing the $\{\text{ips} = \text{pps}\}$ constraint.

VII. Conclusion

In this work, we have developed InstaMeasure for instant flow detection, by counting of packets and bytes in high-speed networks. Our approach is different from conventional measurement frameworks in that we reduced detection delay by introducing a new notion of very large In-DRAM working set of active flows. As a result, we could obtain measurement results with under 1 ms detection delay, which is negligibly small compared to tens or even hundreds of milliseconds in conventional approaches. We demonstrated InstaMeasure's performance and feasibility through extensive analyses, thus opening a new direction in per-flow measurement.

Acknowledgement

This research was supported by the Grobal Research Laboratory (GRL) Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2016K1A1A2912757). This work has supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2017R1A2B4010657). DaeHun Nyang and Aziz Mohaisen are the corresponding authors.

References

[1] , "The zettabyte era: Trends and analysis," http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html.

[2] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for database applications," *Journal of Computer System Science*, vol. 31, pp. 182–209, 1985.

[3] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 137–147, 1999.

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

[5] X. A. Dimitropoulos, P. Hurley, and A. Kind, "Probabilistic lossy counting: an efficient algorithm for finding heavy hitters," *Computer Communication Review*, vol. 38, no. 1, p. 5, 2008.

[6] S. Cohen and Y. Matias, "Spectral bloom filters," in *Proceedings of the 2003 ACM International Conference on Management of Data, SIGMOD 2003, San Diego, California, USA, June 9-12, 2003*, 2003, pp. 241–252.

[7] A. Kumar, J. Xu, and J. Wang, "Space-code bloom filter for efficient per-flow traffic measurement," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, pp. 2327–2339, 2006.

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

[9] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: a novel counter architecture for per-flow measurement," in *Proceedings of the 2008 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2008, Annapolis, MD, USA, June 2-6, 2008*, 2008, pp. 121–132.

[10] T. Li, S. Chen, and Y. Ling, "Fast and compact per-flow traffic measurement through randomized counter sharing," in *Proceedings of the 30th IEEE International Conference on Computer Communications, INFOCOM 2011, 10-15 April 2011, Shanghai, China*, 2011, pp. 1799–1807.

[11] "NetFlow," http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[12] "sFlow," http://www.sflow.org/.

[13] "jFlow," https://www.juniper.net/us/en/local/pdf/app-notes/3500204-en.pdf.

[14] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the 2017 ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, 2017, pp. 113–126.

[15] "Td-routing: Supported route table entries," https://docs.cumulusnetworks.com/display/DOCS/Routing#Routing-SupportedRouteTableEntries.

[16] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, 2013, pp. 29–42.

[17] D. Nyang and D. Shin, "Recyclable counter with confinement for real-time per-flow measurement," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 3191–3203, 2016.

[18] R. Jang, D. Cho, Y. Noh, and D. Nyang, "Rflow$^+$: An sdn-based WLAN monitoring and management framework," in *Proceedings of the 2017 IEEE International Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, 2017, pp. 1–9.

[19] "The cooperative association for internet data analysis, equinix chicago data center," https://www.caida.org, [Apr 06 2016].

[20] M. Chen, S. Chen, and Z. Cai, "Counter tree: A scalable counter architecture for per-flow traffic measurement," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1249–1262, 2017.

[21] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *Proceedings of the 1999 IEEE International Conference on Computer Communications, INFOCOM 1999, New York, NY, USA, March 21-25, 1999*, 1999, pp. 126–134.

[22] "All about bare metal switch," https://bm-switch.com/.

[23] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, 2016, pp. 311–324.

[24] R. Karp, S. Shenker, and C. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Transactions on Database Systems*, vol. 28, no. 1, pp. 51–55, 2003.

[25] N. Kamiyama and T. Mori, "Simple and accurate identification of high-rate flows by packet sampling," in *Proceedings of the 2006 IEEE International Conference on Computer Communications, INFOCOM 2006, 23-29 April 2006, Barcelona, Catalunya, Spain*, 2006.

[26] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in streaming data," *TKDD*, vol. 1, no. 4, pp. 2:1–2:48, 2008.

[27] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Randomized admission policy for efficient top-k and frequency estimation," in *Proceedings of the 2017 IEEE International Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, 2017, pp. 1–9.

[28] ——, "Optimal elephant flow detection," in *Proceedings of the 2017 IEEE International Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, 2017, pp. 1–9.

[29] M. T. Goodrich and M. Mitzenmacher, "Invertible bloom lookup tables," in *Proceedings of the 49th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2011, Allerton Park & Retreat Center, Monticello, IL, USA, 28-30 September, 2011*, 2011, pp. 792–799.

[30] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM Special Interest Group on Data Communication, SIGCOMM 2016, Florianopolis, Brazil, August 22-26, 2016*, 2016, pp. 101–114.