SketchINT: Empowering INT with TowerSketch for Per-flow Per-switch Measurement

Kaicheng Yang*, Yuanpeng Li*, Zirui Liu*, Tong Yang*†, Yu Zhou‡,
Jintao He*, Jing'an Xue§, Tong Zhao*, Zhengyi Jia§, Yongqiang Yang§

***Abstract*—[1] Network measurement is indispensable to network operations. Two most promising measurement solutions are In-band Network Telemetry (INT) solutions and sketching solutions. INT solutions provide fine-grained per-switch per-packet information at the cost of high network overhead. Sketching solutions have low network overhead but fail to achieve both simplicity and accuracy for per-flow measurement. To keep their advantages, and at the same time, overcome their shortcomings, we first design SketchINT to combine INT and sketches, aiming to obtain all per-flow per-switch information with low network overhead. Second, for deployment flexibility and measurement accuracy, we design a new sketch for SketchINT, namely TowerSketch, which achieves both simplicity and accuracy. The key idea of TowerSketch is to use different-sized counters for different arrays under the property that the number of bits used for different arrays stays the same. TowerSketch can automatically record larger flows in larger counters and smaller flows in smaller counters. We have fully implemented our SketchINT prototype on a testbed consisting of 10 switches. We also implement our TowerSketch on P4, single-core CPU, multi-core CPU, and FPGA platforms to verify its deployment flexibility. Extensive experimental results verify that 1) TowerSketch achieves better accuracy than prior art on various tasks, outperforming the state-of-the-art ElasticSketch up to 13.9 times in terms of error; 2) Compared to INT, SketchINT reduces the number of packets in the collection process by $3 \sim 4$ orders of magnitude with an error smaller than 5%.**

***Index Terms*—Network measurement; INT; Sketch**

I. Introduction

A. Background and Motivation

Network measurement is essential to various network operations, including traffic engineering [1], [2], anomaly detection [3]–[5], failure troubleshooting [6], [7], network accounting and billing [8], flow scheduling [9], [10], and congestion control [11]. Among all existing works, two kinds of measurement solutions are widely acknowledged as the most promising: In-band Network Telemetry (INT) solutions [12]–[15] and sketching solutions [8], [16]–[23].

The first kind is the INT solutions. INT solutions obtain per-switch information by configuring the switches to insert predefined packet-level information, *i.e.*, *INT information*, into each incoming packet. As per-switch information is as important as per-flow information, more and more commodity switches start to support INT. However, the main shortcoming of INT is its high *network overhead* incurred by collecting INT information: 1) many additional packets, and 2) large additional bandwidth usage. Specifically, to perform per-flow measurement, current solution for collecting INT information is to mirror the header of each packet with the INT information to a global analyzer in each switch (postcard mode [14]) or only the sink switches (passport mode [12]). Both modes at least double the number of packets in the network and consume large bandwidth.

The second kind is the sketching solutions. Sketches are a kind of probabilistic data structures used to measure per-flow information with small memory usage and low bandwidth overhead. The development of sketches undergoes two phases: 1) *simple sketches* which are inaccurate, and 2) *sophisticated sketches* which are accurate. In the first phase, typical sketches include sketches of Count-Min (CM) [16], Conservative Update (CU) [8], and Count [17]. These sketches are simple and easy to use. However, they suffer from poor accuracy because they do not match the practical network traffic which is often highly skewed: most flows are small and a small amount of large flows contribute to most traffic [8], [24], [25]. In the second phase, typical sketches include ElasticSketch [19], NitroSketch [20], and ASketch [23].

These sketches improve accuracy at the cost of complicated data structures and operations. Specifically, the state-of-the-art solution, ElasticSketch [19] uses a voting technique to separate large flows from small flows, and ASketch [23] maintains the top-k flows by checking their sizes during each insertion operation. Compared with the simple sketches, sophisticated sketches have two shortcomings: 1) they are more complicated in terms of data structures and operations; 2) they have many more parameters which need to be carefully tuned. These two shortcomings hinder their implementation in practice, especially in hardware, such as FPGA and P4-capable switches [26]. For example, in Tofino switches [27], a classic 3-array CM sketch can be implemented within just one stage. In contrast, an ElasticSketch consumes 9 stages.

Due to the complexity of sophisticated sketches, operators in industrial community still prefer the most simple sketch – the CM sketch despite its poor accuracy. However, sketches can hardly achieve per-switch measurement because of the following two reasons. 1) Until now, commodity switches still

[1]Kaicheng Yang, Yuanpeng Li, and Zirui Liu contribute equally to this paper. Tong Yang (yangtongemail@gmail.com) is the corresponding author.

*Department of Computer Science and Technology, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, China
†Peng Cheng Laboratory, Shenzhen, China
‡Department of Computer Science, Tsinghua University, China
§Huawei Cloud, Huawei Technologies, China

978-1-6654-4131-5/21/$31.00 ©2021 IEEE

do not support programming, which means sketches cannot be deployed on these switches. 2) While programmable switches have become increasingly popular in recent years, only simple sketches and a small fraction of sophisticated sketches can be deployed on programmable switches.

In summary, as shown in Table I, INT solutions have high network overhead, and existing sketching solutions suffer two shortcomings: 1) they cannot achieve both simplicity and accuracy, and 2) cannot measure per-switch information. This paper aims to overcome all these shortcomings, while achieving per-flow per-switch measurement, which refers to measuring the information of each flow at every switch/hop. In other words, the design goal of this paper is to *design a new solution for per-flow per-switch measurement, while achieving simplicity, accuracy and low network overhead simultaneously*.

Table I: Design goals.

Advantages	INT	Sketch	Goal
Per-flow per-switch measurement	✓	×	✓
Low network overhead	×	✓	✓
Simplicity and accuracy	✓	×	✓

B. Our Proposed Solution

As shown in Table I, INT solutions achieve per-flow per-switch measurement, and sketching solutions achieve low network overhead. To keep the advantages of both INT and sketching, and at the same time, overcome their shortcomings, we propose our first contribution, which is to combine INT and sketches, namely the **SketchINT** solution. For the sake of deployment flexibility, the ideal sketch for SketchINT should be simple and easy to use. For the sake of measurement accuracy, the ideal sketch for SketchINT should have high accuracy. Unfortunately, no existing sketch can achieve both simplicity and accuracy. Motivated by this, our second contribution is to design a new sketch, namely **TowerSketch**, which is as simple as simple sketches, while as accurate as sophisticated sketches. Our third contribution is to build a prototype to verify the effectiveness and efficiency of our proposed solutions.

Contribution I: combining INT and sketches. To combine the advantages of INT and sketches, we design SketchINT, which achieves per-flow per-switch measurement and low network overhead at the same time. The key design of SketchINT is to first compress all INT information into compact sketches for collection, rather than directly transmit them to the global analyzer as INT does. Specifically, SketchINT first aggregates the *per-packet* INT information into a small amount of *per-flow* information, then further encodes the per-flow information into compact sketches. In this way, the bandwidth usage and the number of packets are significantly reduced. Finally, SketchINT transmits these sketches to the global analyzer with jumbo frames to further reduce the number of packets.

SketchINT has 3 working modes, where the sketches are deployed in different places. First, SketchINT can deploy sketches on sink node switches, *i.e.*, edge switches. Second, considering that the memory resources of switches are relatively limited, SketchINT can deploy sketches on end-hosts for achieving higher measurement accuracy. Third, SketchINT can offload the sketches to FPGA-based SmartNICs, so as to save the expensive CPU resources in end-hosts for economic benefits. To support all the above three working modes, our sketch should be simple enough to be deployed on the three platforms: P4, CPU, and FPGA.

Contribution II: designing a simple and accurate sketch – TowerSketch. To be simple, TowerSketch just consists of several counter arrays and hash functions. To be accurate under the skewed network traffic, TowerSketch uses different-sized counters for different arrays while allocating the same amount of memory for each array. In this way, TowerSketch can automatically record larger flows in larger counters and smaller flows in smaller counters. As shown in Figure 1, TowerSketch organizes the counters into a tower shape consisting of d arrays. For every two adjacent arrays, the array at the higher level has fewer counters and its counters are larger in size. The property of TowerSketch is that *the numbers of bits used for different arrays are the same.* The insertion and query operations of TowerSketch are similar to those of the CM sketch (see § IV-A). In this way, TowerSketch can approach a state that every bit is counting. Thanks to the simplicity of our TowerSketch, it can be easily extended to support a wide range of measurement tasks, and can be implemented on various software and hardware platforms, such as P4 [26], [27], single-core CPU, multi-core CPU, and FPGA.

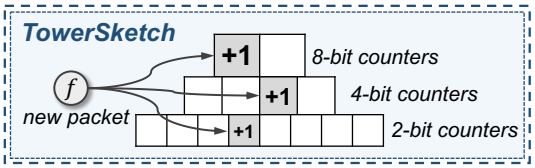


Figure 1: TowerSketch example.

Contribution III: building a SketchINT prototype. To verify the effectiveness of our combination of INT and sketches, and evaluate the performance of our proposed sketch, we have fully implemented a SketchINT prototype on a testbed consisting of 10 programmable switches and 8 end-hosts in a FatTree topology. This prototype verifies that our solution well achieves the design goal of measuring per-flow per-switch network information with high accuracy and low network overhead using simple operations. Further, our experimental results on this prototype system show that 1) TowerSketch achieves higher accuracy than ElasticSketch on various tasks. The error of TowerSketch is 13.9 times lower than ElasticSketch. 2) SketchINT can reduce the number of packets in the collection process by $3 \sim 4$ orders of magnitude compared to INT with ARE smaller than 5%. To make our results easy to reproduce, we have released all related source codes and datasets at Github[2] without identity information.

II. Related Work

In this section, we summarize the in-band telemetry solutions and the sketching solutions for network measurement. For other measurement solutions, such as sampling and probing solutions, please refer to reference [28]–[47].

[2]https://github.com/SketchINT-code/SketchINT

Table II: Comparison with existing solutions.

Systems	Switch	End-host	Coordinator	Bandwidth	Accuracy
SketchINT	INT-capable	Packet insertion (off-loadable to edge-switch)	×	Medium	High
INT	INT-capable	–	×	High	Full
OmniMon	Programmable	Hash table insertion	✓	Low	Full
LightGuardian	Programmable	Sketch restruction	×	Low	Low

In-band Telemetry Solutions: These solutions insert packet-level statistics into incoming packets in INT-compatible switches. Typical in-band telemetry solutions include INT [12]–[14], its successor PINT [15], and LightGuardian [48]. INT has two collection strategies: passport and postcard. In passport mode [12], switches insert packet-level statistics into each passing packet. Then sink switches mirror the packet headers and the desired INT information into new packets and forward these packets to the analyzer. In postcard mode [14], the mirroring and forwarding process happens in each switch. Both of the two modes at least double the number of packets in the network. As collecting INT information incurs significantly network overhead, PINT [15] chooses to insert packet-level statistics into each packet with a certain probability, which reduces network overhead at the cost of information missing. However, PINT cannot support some measurement tasks, *e.g.*, per-flow per-switch inflated latency detection (see §V-B), and its accuracy is also lower than INT. LightGuardian [48] compresses per-flow information into sketches on programmable switches. The switches periodically split their sketches into sketchlets (sketch fragments) and send the sketchlets to the analyzers by packet piggyback.

Sketching Solutions: There are a great number of sketching solutions, which can be further divided into two categories: simple sketches and sophisticated sketches. Typical simple sketches include CM [16], CU [8], Count [17], CMM [49] and CSM [50]. These sketches often consist of multiple arrays. Each array consists of many counters, and is associated with a hash function that maps flows to a counter in it. Simple sketches are easy to implement and transmit. However, as they equally treat large flows and small flows, the accuracy of these sketches is poor due to hash collisions. To address this problem, sophisticated sketches devise many mechanisms to explicitly separate large flows and small flows, incurring complicated data structures and operations. Typical sophisticated sketches include ElasticSketch [19], NitroSketch [20], ASketch [23], and more [21], [51]. Besides, there are a kind of dedicated sketches designed exclusively for specific measurement systems. Typical dedicated sketches include FlowRadar [22], OpenSketch [29], SketchLearn [52], BeauCoup [53], UnivMon [18], OmniMon [54], and more [55]–[57]. Among them, FlowRadar uses a variant of Invertible Bloom filter [58] to record flow-level information. UnivMon builds several sketches on the data plane to perform many measurement tasks, and uses a key method called universal streaming [59] to sample packets. However, due to its sampling techniques, UnivMon is inevitably not accurate in flow size estimation. OmniMon builds hash tables in end-hosts to record the IDs (5-tuple) of all flows. For different flows, a coordinator assigns different counters in switches to these flows, so as to achieve full accuracy.

We compare SketchINT with existing measurement solutions that can perform per-flow per-switch measurement from five aspects. As shown in Table II, 1) for requirement on switches, SketchINT and INT require much less. They only need switches to support INT, which will be supported by future commodity switches[3]. In contrast, OmniMon and LightGuardian need to deploy dedicated data structures on programmable switches. 2) For requirement on end-hosts, SketchINT inserts packets into TowerSketch in end-hosts, which can be offloaded to programmable edge-switches; INT has no requirement on end-hosts; OmniMon builds hash tables in end-hosts to store the active flows; LightGuardian collects sketch fragments from packets, and uses them to reconstruct complete sketches in end-hosts. 3) For requirement on coordinator, OmniMon needs a coordinator to assign different counters in switches to different flows, so as to avoid collisions and achieve full accuracy. In contrast, SketchINT, INT and LightGuardian do not have such requirement. 4) For bandwidth overhead, INT consumes a large amount of bandwidth for transmitting and collecting INT information; SketchINT consumes less bandwidth as it reduces most bandwidth overhead in the collection; OmniMon and LightGuardian consume the least as they only transmit their data structures. 5) For accuracy, INT and OmniMon achieve full accuracy; SketchINT achieves less accuracy because of hash collisions; LightGuardian achieves the least accuracy because of the limited memory in switches.

III. SketchINT Design

As aforementioned (see Table I), INT solutions come with unacceptable network overhead, while sketching solutions cannot acquire per-flow per-switch information. Therefore, we present **SketchINT**, a scalable network measurement system that integrates sketches and INT to keep their advantages simultaneously while overcoming their shortcomings.

As shown in Figure 2, the SketchINT system comprises three components. The first component is the SketchINT agent that encodes the INT metadata into compact TowerSketches for each incoming packet. Based on operator's monitoring intents, SketchINT agent can be flexibly deployed in three working modes, *i.e.*, the TowerSketch can be built in programmable edge switches, end-host CPUs, or SmartNICs to compactly encode per-flow per-switch information. The second component is the INT-compatible switch which inserts the desired per-switch INT metadata into packets. The last one is the global SketchINT analyzer which is deployed in a commodity

[3]We implement SketchINT prototype with Tofino switches, but actually it can be replaced with INT-capable commodity switches.

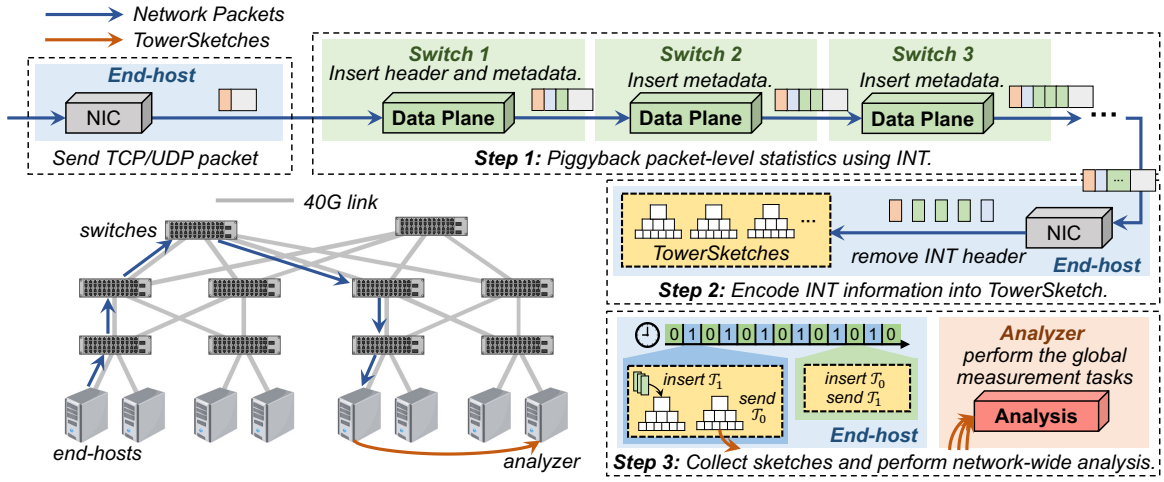


Figure 2: The architecture and workflow of SketchINT.

server collecting TowerSketches from all SketchINT agents. The analyzer is designed with high elasticity and scalability. We take SketchINT working in end-host CPU as an example. Figure 2 shows SketchINT's workflow, consisting of three phases. The remaining parts of this section demonstrate the design details of each phase.

1) Piggyback packet-level statistics using INT. First, SketchINT leverages the INT capability of switches to acquire per-switch packet-level statistics. For resource efficiency and compatibility, SketchINT customizes an INT-layer consisting of an INT instruction header and several INT metadata fields. The INT instruction header consists of a 16-bit *hop-count* indicating the number of passing switches. The INT metadata fields record the desired packet-level statistics in the passing switches. The INT layer is designed to be between the transport layer and the payload. For each packet, in each switch on its path, we insert an INT metadata field into the packet and increment its hop-count by one. In the switch at its first hop, we additionally insert the INT instruction header into the packet, and modify the DSCP field of IPV4 protocol to indicate this packet is an *INT-packet*.

2) Encode INT information into TowerSketches on end-hosts. Owing to the flexibility and sufficient memory of end-hosts, we build several TowerSketches (detailed in § IV) in each end-host to support a variety of measurement tasks. The SketchINT agent in each end-host first reads the INT metadata from the received packets, and then removes INT header and metadata to prevent interference to upper protocols and applications. The SketchINT agent inserts/encodes the information carried by INT metadata (*e.g.*, switch ID, internal latency) into TowerSketches. The encoding process is designed with high performance, introducing acceptable overhead when considering the traffic volume of end-hosts.

3) Collect sketches and perform network-wide analysis. The SketchINT agent in each end-host maintains two groups of TowerSketches $\mathcal{T}_0$ and $\mathcal{T}_1$, a group of active sketches and a group of idle sketches. The status of the two groups of sketches is periodically exchanged, *e.g.*, 5 seconds. For each incoming packet, the SketchINT agent inserts/encodes the INT information carried by its INT metadata into the active sketches. At the same time, the agent forwards the idle sketches to the global SketchINT analyzer. After forwarding, we clear the idle sketches by setting all counters to 0. After collecting all local sketches, the global SketchINT analyzer will have a complete view of the whole network, and can then perform further analysis for each flow in each switch.

IV. TowerSketches

In this section, we first introduce the well-known CM sketch [16]. Then we show the data structure and operations of our TowerSketch, and present the theoretical analysis. We list the main symbols in this paper and their meanings in Table III.

Table III: Main symbols used in this paper.

Symbol	Meaning
f	An arbitrary flow
n_j	Real size of flow f_j
$\hat{n}_j$	Estimated size of flow f_j
m	Number of distinct flows
m_i	Number of distinct flows with size of i
$\hat{m}_i$	Estimated number of distinct flows with size of i
d	Number of counter arrays in TowerSketch
$\mathcal{A}_i$	The i^{th} counter array
$h_i(\cdot)$	The hash function mapping a flow to a hashed counter in the i_{th} counter array $\mathcal{A}_i$
w_i	Number of counters in the i_{th} counter array $\mathcal{A}_i$
δ_i	Each counter in the i_{th} counter array $\mathcal{A}_i$ consists of δ_i bits

A. The Classic CM Sketch

The CM sketch consists of d counter arrays $\mathcal{A}_1, \cdots, \mathcal{A}_d$. Each array $\mathcal{A}_i$ has w counters and it uses a hash function $h_i(.)$ to randomly and uniformly map/hash a flow into a counter in it. When a packet of flow f arrives, CM calculates hash functions to find d counters: $\mathcal{A}_1[h_1(f)], \cdots, \mathcal{A}_d[h_d(f)]$, which are called the d *hashed counters* for convenience. CM just increments the d hashed counters by 1. To query the number of packets of flow f, CM returns the minimum value among the d hashed counters. Based on CM, the CU sketch slightly changes the insertion operation: CU [8] only increments the smallest counter(s). We use "counter(s)" because when there are multiple counters which are considered as the smallest, CU needs to increment them all. Compared with CM, CU

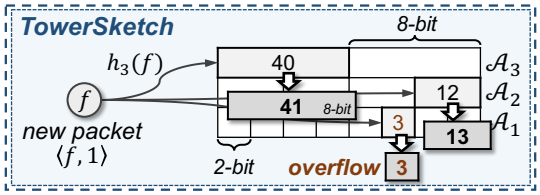


Figure 3: Data structure and example of TowerSketch.

significantly improves accuracy at the cost of not supporting pipeline implementation. Both CM and CU have no under-estimation errors.

Based on the above classic CM/CU sketch, our TowerSketch makes small but non-trivial improvement, aiming to automatically record larger flows in larger counters and small flows in small counters.

B. Data Structure and Operations

Rationale: The key idea of our TowerSketch is to use different-sized counters for different arrays while allocating the same amount of memory for each array. The array at the higher level has fewer counters and its counters are larger in size. For large flows, their small counters at low levels will be overflowed, and thus their frequencies will be kept in the large counters at high levels. For small flows, since the large counters at high levels are occupied by large flows, their frequencies will be kept in small counters. In this way, TowerSketch automatically records larger flows in larger counters and smaller flows in smaller counters.

Data Structure: As shown in Figure 3, TowerSketch consists of d arrays, $\mathcal{A}_1, \cdots, \mathcal{A}_d$. Each array $\mathcal{A}_i$ consists of w_i counters, and is associated with a hash function $h_i(.)$. The size of each counter in array $\mathcal{A}_i$ is δ_i bits. The key difference between our sketch and the CM/CU sketch is: the lower arrays have more counters which are smaller in size, while the higher arrays have fewer counters which are larger in size. Under the property that the number of bits used for different arrays is the same, we allocate the same amount of memory to each array with different counter size.

Example: As shown in Figure 3, the array at the bottom has 8 counters, each of which has 2 bits; the array at the top has 2 counters, each of which has 8 bits. All the three arrays have the same size of memory: 16 bits.

CM insertion: To record a packet with flow ID f, TowerSketch just increments the d hashed counters by 1. If a δ-bit counter overflows after increment, we mark it as an *overflowed counter* by setting its value to $2^\delta - 1$. That is to say, for a δ-bit counter, the maximum value it can record is $2^\delta - 2$. We consider the value of the overflowed counter as $+\infty$, which cannot be incremented or decremented any more.

Example: Figure 3 shows an example to insert a packet. As the counter $\mathcal{A}_1[h_1(f)]$ has overflowed (its value is $2^2-1=3$), we do not increment it.

CU insertion: TowerSketch can use the strategy of CU [8] to improve the accuracy. Instead of incrementing all the d hashed counters, CU insertion only increments the smallest counter(s) that are not overflowed.

Query: The query process for flow f returns the minimum value of the d hashed counters. Recall that we treat the value of an overflowed counter as $+\infty$.

C. Mathematical Analysis

In this subsection, we derive the error bound of TowerSketch. Let $\delta_0 = 0$. Note that $\delta_0 \leqslant \delta_1 \leqslant \cdots \leqslant \delta_d$. Given an arbitrary flow f_j, without loss of generality, we assume its real size n_j satisfies $2^{\delta_{t-1}} - 1 \leqslant n_j \leqslant 2^{\delta_t} - 1$, where $1 \leqslant t \leqslant d$. Let m be the number of flows and n be the sum of the real sizes of all flows, *i.e.*, $n = \sum_{j=1}^{m} n_j$.

Theorem 1 (Error Bound)**.** *Given an arbitrary small positive number ϵ, when $n_j + \epsilon \cdot n < 2^{\delta_t} - 1$, the estimation error of flow f_j is bounded by*

$$Pr\left\{\hat{n_j} \leqslant n_j + \epsilon \cdot n\right\} \geqslant 1 - \prod_{k=t}^{d} \left(\frac{1}{\epsilon \cdot w_k}\right)$$

Proof. We define an indicator variable $I_{j,k,l}$ as

$$I_{j,k,l} = \begin{cases} 1, & h_k(f_j) = h_k(f_l) \ \wedge \ j \neq l \\ 0, & \text{otherwise} \end{cases}$$

As the d hash functions are independent from each other, we have

$$E(I_{j,k,l}) = Pr\left\{h_k(f_j) = h_k(f_l)\right\} = \frac{1}{w_k}$$

We define another variable $X_{j,k} = \sum_{l=1}^{m} n_l \cdot I_{j,k,l}$, indicating the estimation error caused by hash collisions in counter $\mathcal{A}_k[h_k(f_j)]$. Then for $\forall k \geqslant t$, we have

$$\mathcal{A}_k[h_k(f_j)] = \begin{cases} n_j + X_{j,k}, & n_j + X_{j,k} < 2^{\delta_k} - 1 \\ +\infty, & \text{otherwise} \end{cases}$$

And we have

$$E\left(X_{j,k}\right) = E\left(\sum_{l=1}^{m} n_l \cdot I_{j,k,l}\right) = \sum_{l=1}^{m} n_l \cdot E\left(I_{j,k,l}\right) \leqslant \frac{n}{w_k}$$

Therefore, we have

$$\begin{aligned} Pr\{\hat{n_j} \geqslant n_j + \epsilon \cdot n\} &= Pr\left\{\forall k \geqslant t, \mathcal{A}_k[h_k(f_j)] \geqslant n_j + \epsilon \cdot n\right\} \\ &= Pr\left\{\forall k \geqslant t, n_j + X_{j,k} \geqslant n_j + \epsilon \cdot n\right\} \\ &= Pr\left\{\forall k \geqslant t, X_{j,k} \geqslant \epsilon \cdot n\right\} \\ &\leqslant Pr\left\{\forall k \geqslant t, \frac{X_{j,k}}{E(X_{j,k})} \geqslant \epsilon \cdot w_k\right\} \end{aligned}$$

According to the Markov inequality, we can derive that

$$\begin{aligned} Pr\left\{\hat{n_j} \geqslant n_j + \epsilon \cdot n\right\} &\leqslant \prod_{k=t}^{d} \left\{E\left(\frac{X_{j,k}}{E(X_{j,k})}\right) / \left(\epsilon \cdot w_k\right)\right\} \\ &= \prod_{k=t}^{d} \left(\frac{1}{\epsilon \cdot w_k}\right) \end{aligned}$$

Therefore, we have

$$\begin{aligned} Pr\left\{\hat{n_j} \leqslant n_j + \epsilon \cdot n\right\} &= 1 - Pr\left\{\hat{n_j} \geqslant n_j + \epsilon \cdot n\right\} \\ &\geqslant 1 - \prod_{k=t}^{d} \left(\frac{1}{\epsilon \cdot w_k}\right) \end{aligned}$$

□

From the above theorem, we can see that the smaller flow goes with the smaller t. Note that we have $\epsilon \cdot w_k > 1$ for $\forall k \in [0, d]$. Thus, we can conclude that the smaller flow has the smaller error.

D. Discussion

Comparison to CM/CU sketches. For large flows, as fewer counters are assigned to them, TowerSketch has slightly larger overestimation error on large flows due to the limited volume of small flows. For small flows, as much more counters are assigned to them, the overestimation error is significantly reduced. Therefore, the overall accuracy of TowerSketch is much higher than CM/CU sketches.

Comparison to PyramidSketch [51]. PyramidSketch also uses large counters to store large flows. However, its key idea is different from TowerSketch. TowerSketch uses several arrays with small and large counters to adapt to the skewed traffic pattern. PyramidSketch uses the counter sharing idea to combine multiple small-sized counters for large flows. Therefore, in the worst cases, the number of memory accesses in PyramidSketch is very large, while that of TowerSketch always keeps constant. Moreover, due to the use of lots of flag bits, PyramidSketch is not hardware friendly.

Comparison to ElasticSketch [19]. ElasticSketch is the state-of-the-art sketch in frequency estimation. It records the flow IDs of the large flows, and can estimate the large flows more accurately. However, it only uses an 8-bit CM sketch for the small flows, which results in poor accuracy when measuring small flows. On the contrary, our TowerSketch allocates a large number of 2-bit and 4-bit counter for small flows, which allows us to measure flow sizes at a much finer granularity, resulting in an overall higher accuracy.

V. Measurement Tasks

In this section, we elaborate on how our system performs the 6 representative local measurement tasks and the 4 representative global measurement tasks. We take SketchINT working in end-host CPU as an example. Note that besides these tasks, SketchINT also supports all other measurement tasks supported by INT (*e.g.*, routing path, per-switch packet drop). And SketchINT is also perfectly compatible with other INT-based mechanism (*e.g.*, HPCC [11]).

A. Local Measurement Tasks

This subsection presents 6 representative per-flow measurement tasks of TowerSketch, including 1) flow size estimation, 2) heavy hitter detection, 3) heavy change detection, 4) flow size distribution estimation, 5) entropy estimation, and 6) cardinality estimation. Although existing network measurement solutions support these tasks, TowerSketch significantly improves the accuracy in most of them. To support these tasks, the SketchINT agent builds one TowerSketch in each end-host, and inserts each incoming packet with its flow ID as the key.

Flow size estimation: estimating the flow size for any flow ID f_j[4]. TowerSketch directly estimates the flow size by returning the minimum value of the d hashed counters.

Heavy hitter detection: reporting flows whose sizes are larger than a threshold Δ_h. We build a tiny hash table to maintain the heavy hitters by recording their flow IDs. For each incoming packet of flow f_j, we insert it into TowerSketch and query its flow size $\hat{n_j}$. If $\hat{n_j} > \Delta_h$ and $\hat{n_j}$ is not in the hash table, we insert f_j to the hash table. To get all heavy hitters, we report all flow IDs in the hash table.

Heavy change detection: reporting flows whose sizes drastically change beyond a predefined threshold Δ_c in two adjacent time windows. We build one TowerSketch for each time window, and also use the hash table described above to maintain the flows whose sizes are larger than Δ_c. For each flow recorded in the two hash tables, we calculate its flow size difference by querying the two TowerSketches. If the difference exceeds Δ_c, we report it as a heavy change.

Flow size distribution estimation: estimating the distribution of flow sizes. We apply the basic MRAC algorithm [60] to each counter array of TowerSketch and then synthesize the results. Specifically, array $\mathcal{A}_i$ provides the estimated results with flow size in range $[2^{\delta_{i-1}} - 1, 2^{\delta_i} - 1)$.

Entropy estimation: estimating the entropy of flow sizes. After getting the estimation of flow size distribution, we can easily compute the entropy by the following formula: $-\sum \left(m_i \cdot \frac{i}{M} \log \frac{i}{M}\right)$, where m_i is the number of flows with size of i, and $M = \sum (i \cdot m_i)$.

Cardinality estimation: estimating the number of flows. We use the bottom array with the largest number of counters to estimate cardinality, and calculate the results using the linear counting algorithm [61].

B. Global Measurement Tasks

This subsection presents 4 global measurement tasks enabled by SketchINT, including 1) per-flow per-switch latency estimation, 2) per-flow per-switch inflated latency detection, 3) per-switch heavy hitter detection, and 4) per-switch heavy change detection. The first two tasks provide information of latency, which can be used as the basis of flow scheduling, load balancing, and congestion control. The remaining two tasks provide information of large flows, which is useful to problem diagnosis when a switch suffers problems (*e.g.*, congestion, packet drops, full Network Processing Unit (NPU) utilization). Note that none of existing sketching solutions can support these tasks, and compared with INT, SketchINT empowers these tasks with great scalability. To support these tasks, the SketchINT agent builds two TowerSketches, one for the first two latency tasks, and another for the rest two tasks.

Per-flow per-switch latency estimation: reporting the average internal latency in each switch for any given flow. For this task, we configure the switches to insert the switch ID and the internal latency into each incoming packet. In each end-host, the SketchINT agent builds a TowerSketch to record the latency information. We extend each counter in TowerSketch to a bucket consisting of two counters: 1) a *latency counter* recording the total latency of inserted packets, and 2) a *frequency counter* recording the number of inserted packets. For each incoming packet, we first acquire its *forwarding path*, *i.e.*, the recorded switch IDs, and the per-switch internal latency via INT. For each recorded switch S_i, we concatenate the switch ID S_i and the flow ID to form a new key, which is used to locate the d hashed buckets. For each hashed bucket,

[4]A flow ID can be any combination of 5-tuple: source IP address, source port, destination IP address, destination port, and protocol type.

we increment the latency counter by the internal latency in S_i, and increment the frequency counter by one. The global SketchINT analyzer periodically collects TowerSketches from end-hosts. To acquire the latency of flow f_j in its passing switch S_i, we query the corresponding TowerSketch using the concatenated key and report the average latency as the total latency divided by the frequency.

Per-flow per-switch inflated latency detection: reporting the frequency of inflated latency in each switch for any given flow. Inflated latency in switch S_i is defined as the latency which exceeds Δ_l times of the average latency in S_i, where Δ_l is a predefined threshold. We configure the switches as in latency estimation, and we still use the extended TowerSketches in latency estimation to perform this task. In addition, the SketchINT agent builds a tiny hash table. For each incoming packet of f_j, we insert it into the TowerSketch as described above. Every time we find that the latency of a packet in switch S_i exceeds Δ_l times of its estimated average latency, we insert a key consisting of the flow ID and the switch ID into our hash table. The global SketchINT analyzer periodically collects the hash tables and reports the inflated latency.

Per-switch heavy hitter detection: detecting heavy hitters for any switch. We configure the switches to insert the switch ID into each incoming packet. In each end-host, the SketchINT agent builds a TowerSketch and a tiny hash table. For each incoming packet, we acquire the forwarding path via INT. The insertion process of TowerSketch and the hash table is similar to local heavy hitter detection. The only difference is that we insert the flow ID and its forwarding path as a key-value pair to the hash table. The global SketchINT analyzer periodically collects the hash tables. For any given switch S_i, we check all hash tables. If a flow has S_i in its forwarding path, we report it as a heavy hitter in switch S_i.

Per-switch heavy change detection: detecting heavy changes for any switch. In end-hosts and switches, we use the same data structures and operations as in per-switch heavy hitter detection, except the threshold is set to the heavy change threshold Δ_c. For each flow ID recorded in the two adjacent hash tables, we calculate its flow size difference by querying two adjacent TowerSketches. If the difference exceeds Δ_c, we insert the flow ID and its forwarding path into a list. The analyzer periodically collects the lists. For any given switch S_i, we check all lists. If a flow has S_i in its forwarding path, we report it as a heavy change in switch S_i.

C. Discussions

The collection manner of SketchINT leads to losses of real-time capability to some extent. However, note that in each end-host, the collection process just transmits the TowerSketches and some large flows to the global analyzer. Therefore, in practice, SketchINT can collect information in a short period, so as to approach the real-time capability as much as possible.

VI. Implementation of Working Modes

With the great simplicity, TowerSketch can be implemented upon a diversity of platforms, giving operators great flexibility of deploying SketchINT. We have completed three types of TowerSketch implementations, corresponding to the three working modes, respectively. First, we present edge-switch-based TowerSketch which runs on P4-programmable edge switches. Second, considering the limited memory in switches, we present the kernel-based TowerSketch which runs on end-hosts with abundant memory resources. Third, to avoid consuming the expensive CPU resources in end-hosts, we present the FPGA-based TowerSketch which can run on FPGA-based SmartNIC for economic benefits. This section demonstrates the details of each implementation type.

A. TowerSketch on P4-capable Switches

We have implemented our TowerSketch on P4-capable Tofino switches, which can be used as edge switches. Then, edge switches can collect INT metadata fields and perform TowerSketch measurement tasks. To perform all measurement tasks in P4-capable switches, we need the same number of TowerSketches as the maximum hop count in data centers, which is usually 5. Since the Tofino switch processes packets in a pipeline manner, TowerSketch cannot support CU insertion. We implement TowerSketch using several registers and Stateful ALUs (SALU). For each counter array $\mathcal{A}_i$ consisting of w_i counters, we build a register with w_i register elements, where each element stores a corresponding counter in $\mathcal{A}_i$. Note that the registers in Tofino switch only support 8-bit, 16-bit and 32-bit elements, we use the register elements that are slightly larger than w_i to store the counters. For each incoming packet, we use its 5-tuple flow ID to locate the hashed element in each array with pairwise-independent hash functions. Then, we use the SALU to execute the hashed element in each array as described in Section IV.

We compare the resource usage of our TowerSketch with a baseline forwarding program `switch.p4`. Table IV shows the additional resource usage to build a TowerSketch with the following parameters: $d = 3$, $\delta_i = 2^{i+2}$, and $w_i = 2^{17-i}$. We find that compared with `switch.p4`, the additional resource usage is less than 8% across all resources except for SALU. The additional usage percentage of SALU is naturally higher than other resources because we need to use SALU to access the registers. Note that the ASIC processing throughput does not decrease as long as the resource usage can fit into the ASIC resource constraint.

Table IV: Resources usage on P4-capable switches.

Resource types	Baseline	Additional usage
Stateful ALU	16	18.75%
VLIW Actions	82	4.88%
TCAM	145	0.00%
SRAM	562	2.67%
Hash Bits	1851	4.86%
Tenary Crossbar	409	0.00%
Exact Crossbar	371	7.27%

B. TowerSketch on Single/Multi-core CPU

We implement TowerSketch in the user space on both single-core and multi-core CPU platforms. We integrate TowerSketch with a packet receiving program written in DPDK [62] to perform per-flow per-switch measurement. For multi-core CPU platform, we build TowerSketches shared by all

cores and use the lock mechanism for synchronization. To speed up the insertion, we can also abandon the complicated lock mechanism. Our experimental results show that the lock-free version of TowerSketch achieves much higher throughput with almost no loss in accuracy (see § VII-B).

C. TowerSketch on FPGA

To verify that our TowerSketch can be implemented on FPGA-based SmartNIC, we have implemented our Towersketch (using CM insertion) on Xilinx Virtex-7 VC709 with the following parameters: $d = 3$, $\delta_i = 2^{i+2}$, and $w_i = 2^{17-i}$. CU insertion cannot be efficiently implemented because the FPGA process packets in a pipeline manner. We use FPGA device (model XC7VX690TFFG1761-2) as the target platform, which has 433200 Slice LUTs, 866400 Slice Registers, and 1470 Block RAM Tiles (*i.e.*, 30.6Mb on-chip memory). The resource usage information is as follows: 1) TowerSketch uses 45.5 Block RAM Tile, 3.1% of the total on-chip Block RAM; 2) TowerSketch uses 686 LUTs, less than 1% of the 433200 total available. TowerSketch is fully pipelined, which can process one packet in every clock, and update the d hashed counters after eight clocks. The clock frequency of our FPGA is 365 MHz, meaning an insertion speed of 365 Mpps.

VII. Experimental Results

We conduct extensive experiments in our SketchINT prototype. We focus on the following issues:

- **How accurate can TowerSketch perform the 6 local measurement tasks?** We implement TowerSketch using C/C++ program, and evaluate the accuracy of TowerSketch on single-core CPU and multi-core CPU platforms.
- **How accurate can TowerSketch perform the 4 global measurement tasks?** We evaluate the accuracy of TowerSketch in our SketchINT prototype.
- **How much network overhead can SketchINT reduce in the collection process?** We compare the bandwidth usage and the number of generated packets in the collection process of SketchINT with the INT passport mode [12].

Evaluation metrics:

- *Average Absolute Error (AAE):* $\frac{1}{m}\sum_{i=1}^{n}|n_i - \hat{n_i}|$, where m is the number of flows, n_i and $\hat{n_i}$ are the actual and estimated flow sizes respectively.
- *Average Relative Error (ARE):* $\frac{1}{m}\sum_{i=1}^{m}\frac{|n_i-\hat{n_i}|}{n_i}$.
- *F_1 Score:* $\frac{2\cdot PR\cdot RR}{PR+RR}$, where PR (Precision Rate) refers to the ratio of the number of the correctly reported instances to the number of all reported instances, and RR (Recall Rate) refers to the ratio of the number of the correctly reported instances to the number of all correct instances.
- *Relative Error (RE):* $\frac{|True-Est|}{True}$, where $True$ and Est are the true and estimated values, respectively.
- *Weighted Mean Relative Error (WMRE) :* $\frac{\sum_{i=1}^{z}|m_i-\hat{m_i}|}{\sum_{i=1}^{z}\left(\frac{m_i+\hat{m_i}}{2}\right)}$, where m_i and $\hat{m_i}$ are the true and estimated numbers of the flows of size i respectively, and z is the maximum flow size [63].
- *Throughput:* Million packets per second (Mpps).

A. Testbed Setup

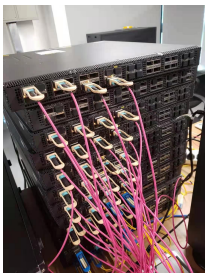


Figure 4: Testbed.

As shown in Figure 4, we implement the SketchINT prototype on a testbed consisting of 10 Tofino switches and 8 end-hosts with 40GbE links in a FatTree topology. The MTU of the network interface cards (NIC) is set to 9000B. In each switch, we insert an INT metadata field consisting of a 16-bit predefined switch ID and a 32-bit internal latency into each incoming packet. The period that active and idle sketches exchange their status is set to 5 seconds, implying that the global analyzer collects sketches from end-hosts in every 5 seconds. In INT passport mode, the mirrored packets are considered without payload.

B. Experimental Results on Local Tasks

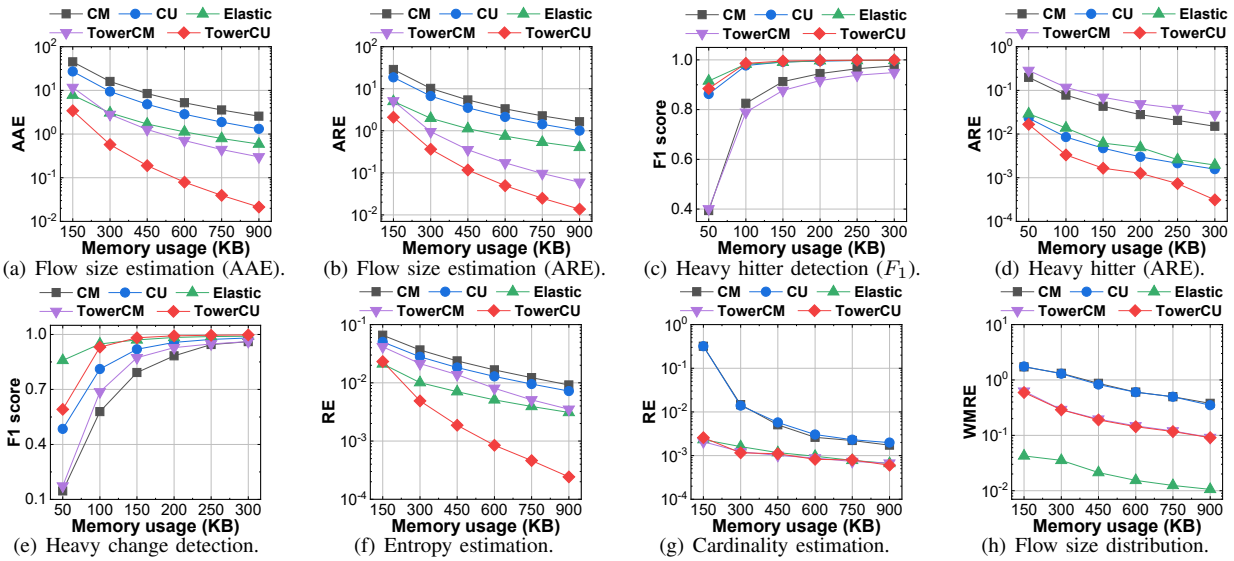
1) Experimental Setup: We use the anonymized IP traces collected in 2018 from CAIDA [64], which is widely used in prior works [63], [65]. Each trace contains about 2.3M packets of 170K flows, with a monitoring time interval of 5s.

We compare TowerSketch with the most widely used CM and CU sketches, and the state-of-the-art ElasticSketch. For TowerSketch, we set $d = 5$, and $\delta_i = 2^i$ for $\forall i \in [1, 5]$. We allocate the same amount of memory for each array with different counter size. For CM and CU, we use 3 hash functions as recommended in literature [66]. For ElasticSketch, we set its parameters as the original paper [19] recommends. We set the capacity of the hash table used in heavy change detection and heavy hitter detection to 1024. We use the famous MurmurHash [67] for all sketches. All experiments are repeated 100 times and the average results are reported. We vary the memory usage to evaluate the accuracy of different sketches. This is equivalent to evaluate the scalability of different sketches with respect to the number of flows that can be monitored concurrently under the same memory usage. The remaining settings are as follows:

- *Heavy hitter detection:* We set the heavy hitter threshold $\Delta_h = 500$, about 0.02% of the total packets.
- *Heavy change detection:* We set the heavy change threshold $\Delta_c = 250$, about 0.01% of the total packets.
- *Processing speed evaluation:* We conduct the flow size estimation experiments on single-core CPU and multi-core CPU. We allocate 2MB of memory to each algorithm. To enlarge the subtle difference between different algorithms, we use 5 hash functions for Tower, CM, and CU. On multi-core CPU, we implement both the lock version and the Lock-Free (LF) version of our TowerSketch.

2) Performance on Local Tasks:

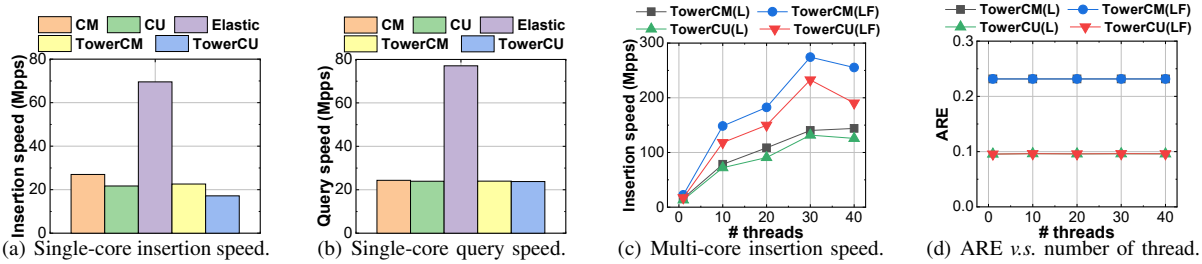
Flow size estimation (Figure 5(a)-(b)): We find that the average ARE of Tower is 13.9 times lower than CM, CU, and ElasticSketch. The results show that when using 900KB of memory, TowerCM achieves at least 1.9 times lower AAE and at least 6.8 times lower ARE than the other three algorithms, and TowerCU achieves at least 28 times lower AAE and at least 29 times lower ARE than the others.



(a) Flow size estimation (AAE). (b) Flow size estimation (ARE). (c) Heavy hitter detection (F_1). (d) Heavy hitter (ARE).

(e) Heavy change detection. (f) Entropy estimation. (g) Cardinality estimation. (h) Flow size distribution.

Figure 5: Performance on local measurement tasks, where **TowerCM** represents the TowerSketch using CM insertion and **TowerCU** represents the TowerSketch using CU insertion.



(a) Single-core insertion speed. (b) Single-core query speed. (c) Multi-core insertion speed. (d) ARE *v.s.* number of thread.

Figure 6: Processing speed on single-core CPU and multi-core CPU, where **L** represents the lock version and **LF** represents the lock-free version.

Heavy hitter detection (Figure 5(c)-(d)): We find that TowerCU always achieves better F_1 score than CM, CU, and ElasticSketch. The results show that when using 300KB of memory, the F_1 score of TowerCU reaches 0.9997, and the ARE of TowerCU achieves 3×10^{-4}, which is at least 5 times lower than the others.

Heavy change detection (Figure 5(e)): We find that TowerCU achieves better F_1 score than CM, CU, and ElasticSketch. When using 300KB of memory, the F_1 score of TowerCU is 0.998, while that of CM, CU, and ElasticSketch are 0.96, 0.98, and 0.99, respectively.

Entropy estimation (Figure 5(f)): We find that the average RE of TowerCU is 5.7 times lower than CM, CU, and ElasticSketch. The results show that when using 900KB of memory, the RE of TowerCU is 2.42×10^{-4}, which is at least 12.9 times lower than the other algorithms.

Cardinality estimation (Figure 5(g)): We find that Tower achieves comparable accuracy with ElasticSketch and better accuracy than CM and CU. The results show that when using 900KB of memory, the RE of TowerCU is 6.02×10^{-4}, while that of CM, CU, and ElasticSketch are 1.7×10^{-3}, 2×10^{-3}, and 6.62×10^{-4}, respectively.

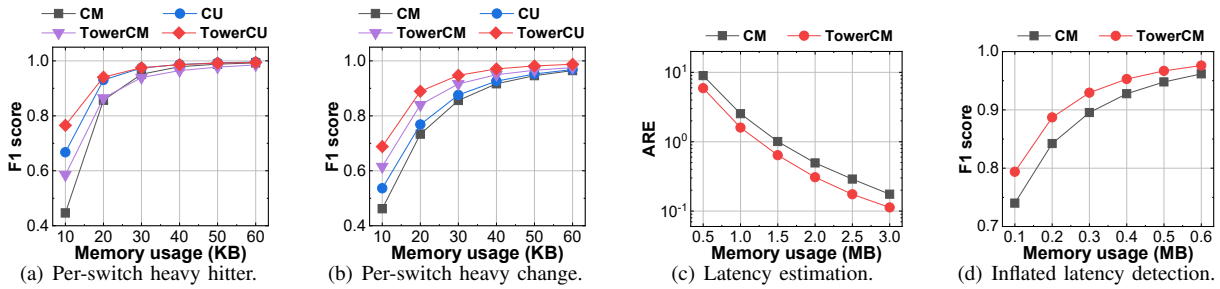
Flow size distribution estimation (Figure 5(h)): We find that the average WMRE of Tower is 4.0 times lower than CM and CU. The results show that when using 900KB of memory, the WMRE of Tower is 0.09, while that of CM and CU are 0.38 and 0.35, respectively.

Speed on CPU (Figure 6(a)-6(d)): We find that Tower achieves comparable processing speed with CM and CU on single-core CPU. The results show that the throughput of TowerCM is 22.6Mpps, while that of CM and CU are 27.1Mpps and 21.7Mpps, respectively. Among the five algorithms, ElasticSketch achieves the fastest speed, but our Tower has higher accuracy. On multi-core CPU, the lock-free version of Tower achieves much higher insertion speed, and the accuracy loss incurred by concurrency is negligible. The results show that when using 30 threads, the Lock-Free version of Tower can reach a throughput of 274Mpps, which is 1.96 times higher than the lock version.

In summary, compared with prior arts, our TowerCU achieves better accuracy in most local measurement tasks. This is because TowerSketch automatically stores large flows into large counters and small flows into small counters, and thus make full utilization of every bit. In addition, on multi-core CPU, the lock-free version of TowerSketch achieves extremely high throughput without compromising the accuracy, indicating that the complicated locking mechanism can be abandoned.

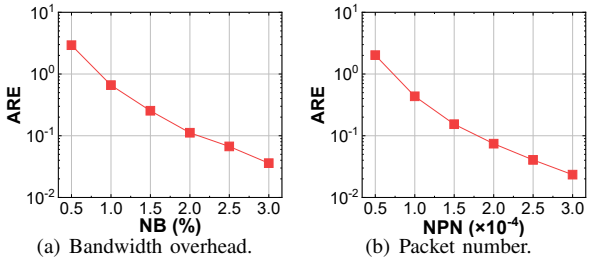
C. Experimental Results on Global Tasks

1) Experimental Setup: In our SketchINT system, we configure the eight end-hosts to send and receive traffic at the



(a) Per-switch heavy hitter. (b) Per-switch heavy change. (c) Latency estimation. (d) Inflated latency detection.

Figure 7: Performance on global measurement tasks.



(a) Bandwidth overhead. (b) Packet number.

Figure 8: Bandwidth and packet number overhead.

same time, and they use the Traffic Generator [68] to generate the traffic under DCTCP [69] distribution. Each end-host independently builds several TowerSketches to perform the measurement tasks. An analyzer collects the sketches every 5 seconds and performs further network-wide analysis. To provide ground-truth analysis, we dump all packets in the network into a trace every 5 second. Each trace contains about 16M packets and 73K flows.

Since ElasticSketch does not naturally support the global tasks, we just compare TowerSketch with CM and CU. For TowerSketch, we set $d = 3$, and $\delta_i = 2^{i+2}$ for $\forall i \in [1, 3]$. For CM and CU, we use 3 hash functions. For per-switch heavy hitter detection, we set $\Delta_h = 1500$. For per-switch heavy change detection, we set $\Delta_c = 750$. And for per-flow per-switch inflated latency detection, we set $\Delta_l = 5$.

2) Performance on Global Tasks:

Per-switch heavy hitter detection (Figure 7(a)): We find that TowerCU is more accurate than CM and CU. When using 10KB of memory, the F_1 score of TowerCU is 0.765, while that of CM and CU are 0.446 and 0.667, respectively. The F_1 score of TowerCU reaches 0.99 under 50KB of memory.

Per-switch heavy change detection (Figure 7(b)): We find that TowerCU is more accurate than CM and CU. Under 10KB of memory, the F_1 score of TowerCU is 0.688, while that of CM and CU are 0.462 and 0.536, respectively. Under 50KB of memory, the F_1 score of TowerCU reaches 0.98.

Latency estimation (Figure 7(c)): We find that Tower achieves at least 0.52× lower ARE than CM. Under 3MB memory, the ARE of Tower is 0.11, while that of CM is 0.18.

Inflated latency detection (Figure 7(d)): We find that Tower always achieves better F_1 score than CM. When using 600KB of memory, the F_1 score of Tower is 0.976, while CM is 0.961.

Bandwidth overhead and packet number comparison (Figure 8(a)-8(b)): We find that SketchINT can achieve almost the same accuracy as INT while only using less than 3% bandwidth of INT, reducing the number of packets by $3 \sim 4$ orders of magnitude. We evaluate the bandwidth overhead and the number of generated packets of SketchINT on latency estimation task. We use *Normalized Bandwidth (NB)* to represent the ratio of the bandwidth overhead in the collection process of SketchINT to that in the standard INT passport mode. We use *Normalized Packet Number (NPN)* to represent the ratio of the number of packets in the collection process of SketchINT to that in standard INT passport mode. The results show that only 3% NB and 0.03% NPN can achieve $< 5\%$ ARE on the latency estimation task.

In summary, SketchINT achieves high accuracy because TowerSketch can efficiently encode per-flow information. SketchINT supports various global measurement tasks as we can insert any desired INT information into the packets. By first aggregating per-packet INT information into per-flow information and then encoding it into TowerSketches, SketchINT significantly reduces the number of additional packets and bandwidth usage. We believe such a combination of INT and TowerSketch is promising and can support many more measurement tasks.

VIII. Conclusion

In this paper, we present SketchINT, which empowers INT with sketches to provide per-flow per-switch network measurement with low network overhead. For deployment flexibility, a simple and accurate sketch, namely TowerSketch, is designed to support multiple local and global measurement tasks. We have fully implemented a SketchINT prototype on a testbed consisting of 10 programmable switches and 8 end-hosts. We also verify that our TowerSketch can be implemented on four platforms: single-core CPU, multi-core CPU, FPGA and P4 switches. Extensive experimental results on the testbed verify that SketchINT provides per-flow per-switch measurement, while achieving simplicity, accuracy and low network overhead simultaneously.

Acknowledgment

We would like to thank our shepherd Eric Keller and the anonymous reviewers for their thoughtful feedback. We would like to thank Cheng Chen for his participation in SketchINT system implementation. This work is supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, National Natural Science Foundation of China (NSFC) (No. U20A20179), the project of "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications" (No. LZC0019), and Computing and Network Innovation Lab, Huawei Cloud.

REFERENCES

[1] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, pages 1–12, 2011.

[2] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational ip networks: Methodology and experience. *IEEE/ACM Transactions On Networking*, 9(3):265–279, 2001.

[3] Nick Duffield, Carsten Lund, and Mikkel Thorup. Estimating flow distributions from sampled flow statistics. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 325–336, 2003.

[4] Cristian Estan, Ken Keys, David Moore, and George Varghese. Building a better netflow. *ACM SIGCOMM Computer Communication Review*, 34(4):245–256, 2004.

[5] Jianning Mai, Chen-Nee Chuah, Ashwin Sridharan, Tao Ye, and Hui Zang. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 165–176, 2006.

[6] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, pages 71–85, 2014.

[7] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.

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

[9] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 455–468, 2015.

[10] Ziyang Li, Wei Bai, Kai Chen, Dongsu Han, Yiming Zhang, Dongsheng Li, and Hongfang Yu. Rate-aware flow scheduling for commodity data center networks. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[11] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. ACM New York, NY, USA, 2019.

[12] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *SIGCOMM*, 2015.

[13] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 3–14, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Cisco Nexus 9000 Series NX-OS Programmability Guide. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/92x/programmability/guide/b-cisco-nexus-9000-series-nx-os-programmability-guide-92x/b-cisco-nexus-9000-series-nx-os-programmability-guide-92x_chapter_0100001.html#id_95566.

[15] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *SIGCOMM*, pages 662–680, 2020.

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

[17] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *ICALP*, pages 693–703, 2002.

[18] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *SIGCOMM*, 2016.

[19] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *SIGCOMM*, pages 561–575, 2018.

[20] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *SIGCOMM*, pages 334–350. ACM New York, NY, USA, 2019.

[21] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. Toward nearly-zero-error sketching via compressive sensing. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 1027–1044, 2021.

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

[23] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1449–1463, 2016.

[24] Yin Zhang, Matthew Roughan, Walter Willinger, and Lili Qiu. Spatio-temporal compressive sensing and internet traffic matrices. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 267–278, 2009.

[25] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[26] P4-16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-checksums.

[27] Barefoot tofino: World's fastest p4-programmable ethernet switch asics. https://barefootnetworks.com/products/brief-tofino/.

[28] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *SIGCOMM*, pages 479–491, 2015.

[29] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI*, pages 29–42, 2013.

[30] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In *ICDCS*, pages 228–237, 2014.

[31] Fangfan Li, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. A large-scale analysis of deployed traffic differentiation practices. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 130–144. ACM New York, NY, USA, 2019.

[32] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: Active device and link failure localization in data center networks. In *NSDI*, pages 599–614, 2019.

[33] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 139–152. ACM, 2015.

[34] Hongyi Zeng, Ratul Mahajan, Nick McKeown, George Varghese, Lihua Yuan, and Ming Zhang. Measuring and troubleshooting large operational multipath networks with gray box testing. *Mountain Safety Res., Seattle, WA, USA, Rep. MSR-TR-2015-55*, 2015.

[35] Amogh Dhamdhere, David D Clark, Alexander Gamero-Garrido, Matthew Luckie, Ricky KP Mok, Gautam Akiwate, Kabir Gogia, Vaibhav Bajpai, Alex C Snoeren, and Kc Claffy. Inferring persistent interdomain congestion. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 1–15, 2018.

[36] Tal Mizrahi, Gidi Navon, Giuseppe Fioccola, Mauro Cociglio, Mach Chen, and Greg Mirsky. Am-pm: Efficient network telemetry using alternate marking. *IEEE Network*, 33(4):155–161, 2019.

[37] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *SIGCOMM*, pages 440–453, 2016.

[38] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. In *NSDI*, pages 595–612, 2017.

[39] Minlan Yu. Network telemetry: towards a top-down approach. *ACM SIGCOMM Computer Communication Review*, 49(1):11–17, 2019.

[40] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *12th {USENIX}*

Symposium on Operating Systems Design and Implementation ({OSDI} 16), pages 233–248, 2016.

[41] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 421–436, 2019.

[42] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. Stroboscope: Declarative network monitoring on a budget. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 467–482, 2018.

[43] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 453–456, 2018.

[44] Amogh Dhamdhere, Renata Teixeira, Constantine Dovrolis, and Christophe Diot. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *CoNEXT*, pages 1–12, 2007.

[45] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. detector: a topology-aware monitoring system for data center networks. In *ATC*, pages 55–68, 2017.

[46] John Sonchack, Oliver Michel, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 823–835, Boston, MA, July 2018. USENIX Association.

[47] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is advance knowledge of flow sizes a plausible assumption? In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 565–580, 2019.

[48] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 991–1010, 2021.

[49] Fan Deng and Davood Rafiei. New estimation algorithms for streaming data: Count-min can do more. *Webdocs. Cs. Ualberta. Ca*, 2007.

[50] Tao Li, Shigang Chen, and Yibei Ling. Per-flow traffic measurement through randomized counter sharing. *IEEE/ACM Transactions on Networking (TON)*, 20(5), 2012.

[51] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment*, 10(11):1442–1453, 2017.

[52] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *SIGCOMM*, pages 576–590, 2018.

[53] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *SIGCOMM*, pages 226–239, 2020.

[54] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 404–421, 2020.

[55] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Dream: dynamic resource allocation for software-defined measurement. *ACM SIGCOMM Computer Communication Review*, 44(4), 2015.

[56] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *CoNEXT*, pages 1–13, 2015.

[57] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *CoNEXT*, pages 481–495, 2016.

[58] Michael T Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*. IEEE, 2011.

[59] Vladimir Braverman and Rafail Ostrovsky. Generalizing the layering method of indyk and woodruff: Recursive sketches for frequency-based vectors on streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 58–70. Springer, 2013.

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

[61] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2), 1990.

[62] Dpdk is the data plane development kit that consists of libraries to accelerate packet processing workloads running on a wide variety of cpu architectures. https://www.dpdk.org/.

[63] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *SIGCOMM*, pages 113–126, 2017.

[64] The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.

[65] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. Heavykeeper: An accurate algorithm for finding top-k elephant flows. *IEEE/ACM Transactions on Networking*, 27(5):1845–1858, 2019.

[66] Amit Goyal, Hal Daumé III, and Graham Cormode. Sketch algorithms for estimating point queries in nlp. In *EMNLP-CoNLL*, pages 1093–1103, 2012.

[67] The source code of MurmurHash. https://github.com/aappleby/smhasher.

[68] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling ecn in multi-service multi-queue data centers. In *NSDI*, pages 537–549, 2016.

[69] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *SIGCOMM*, pages 63–74, 2010.