SketchINT: Empowering INT With TowerSketch for Per-Flow Per-Switch Measurement

Kaicheng Yang, Sheng Long, Qilong Shi, Yuanpeng Li, Zirui Liu, Yuhan Wu, Tong Yang, and Zhengyi Jia

***Abstract*—Network measurement is indispensable to network operations. INT solutions that can provide fine-grained per-switch per-packet information serve as promising solutions for per-flow per-switch measurement. The main shortcoming of INT is its high network overhead incurred by collecting INT information, making INT impractical for production deployment. Sketches that can compactly record per-flow information with small memory footprint, are a promising choice for compressing INT information to reduce INT overhead. An ideal sketch for efficiently compressing INT information in practice should achieve both simplicity and accuracy, but no existing sketch achieves both. Motivated by this, we first design SketchINT to combine INT and sketches, aiming to obtain all per-flow per-switch information with low network overhead. Second, 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. To further ease the configuration and give network operators more confidence on performance of TowerSketch, we propose a method for precise error bound estimation. 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 27.7 times in terms of error; 2) Compared to INT, SketchINT reduces the number of packets belonging to the control plane overhead by $3 \sim 4$ orders of magnitude with an error smaller than 5%; 3) The estimated error bound of TowerSketch can almost match the actual error bound.**

***Index Terms*—In-band Network Telemetry (INT), network measurement, sketch.**

Manuscript received 27 July 2022; revised 5 March 2023; accepted 5 August 2023. Date of publication 28 August 2023; date of current version 12 September 2023. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B0101390001, and in part by the National Natural Science Foundation of China (NSFC) under Grant U20A20179. Recommended for acceptance by B. Veeravalli. The preliminary version of this paper titled "SketchINT: Empowering INT with TowerSketch for Per-flow Per-switch Measurement" was published in 29th IEEE International Conference on Network Protocols, ICNP 2021, Dallas, TX, USA, November 1-5, 2021 [DOI: 10.1109/ICNP52444.2021.9651940]. *(Corresponding author: Tong Yang.)*

Kaicheng Yang, Sheng Long, Qilong Shi, Yuanpeng Li, Zirui Liu, and Yuhan Wu are with the School of Computer Science, National Key Laboratory for Multimedia Information Processing, Peking University, Beijing 100871, China (e-mail: ykc@pku.edu.cn; lgenhogns@pku.edu.cn; stallone@pku.edu.cn; liyuanpeng@pku.edu.cn; zirui.liu@pku.edu.cn; yuhan.wu@pku.edu.cn).

Tong Yang is with the School of Computer Science, National Key Laboratory for Multimedia Information Processing, Peking University, Beijing 100871, China, and also with Peng Cheng Laboratory, Shenzhen, Guangdong 518066, China (e-mail: yangtongemail@gmail.com).

Zhengyi Jia is with Huawei Cloud, Huawei Technologies, Shenzhen, Guangdong 518100, China (e-mail: jiazy1991@outlook.com).

Digital Object Identifier 10.1109/TPDS.2023.3303924

I. Introduction

A. Background and Motivation

NETWORK measurement is essential to various network operations, including traffic engineering [2], [3], anomaly detection [4], [5], [6], failure troubleshooting [7], [8], network accounting and billing [9], flow scheduling [10], [11], and congestion control [12]. Nowadays, measuring per-switch information at flow-level granularity has become the community consensus [13].

In recent years, INT solutions [14], [15], [16], [17] that can provide fine-grained per-switch per-packet information have been widely accepted as promising solutions for per-flow per-switch measurement. INT solutions obtain per-switch information by configuring the switches to insert predefined packet-level information, i.e., *INT information*, into each incoming packet. To perform per-flow per-switch 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/INT-XD [16]), or only the INT-sink switches (passport mode/INT-MD [14]).

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, the network overhead has two dimensions: (1) control plane overhead, i.e., the overhead of sending INT information to the collector from INT-capable switches; (2) data plane overhead, i.e., the increase in packet size due to piggybacking switch internal state on live network traffic. While postcard mode/INT-XD eliminates the data plane overhead, it magnifies the control plane overhead of passport mode/INT-MD several times as it reports at each switch. In order to make INT practical for production deployment, it is strongly desired to reduce the network overhead of INT, and the *first design goal* of this paper is to design a practical INT-based system that supports per-flow per-switch measurement with low network overhead.

Sketches [18], [19], [20], [21], [22], [23], [24], a kind of probabilistic data structures that can compactly record per-flow information with small memory overhead, are a promising choice for compressing INT information. 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) [18], Conservative Update (CU) [9], and Count [25]. These sketches simply consist of counters, and are therefore 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 [9], [26], [27].

In the second phase, typical sketches include ElasticSketch [20], PyramidSketch [28], and ASketch [24]. These sketches improve accuracy at the cost of additional data structures for recording additional information such as flow IDs and flags. Specifically, the state-of-the-art solution, ElasticSketch [20] uses a voting technique to separate large flows from small flows, and ASketch [24] maintains the top-k flows by checking their sizes during each insertion operation.

Compared with simple sketches, sophisticated sketches have two shortcomings: 1) they have additional data structures other than counters and corresponding additional operations other than incrementing counters; 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 [29]. For example, in Tofino switches [30], the workflow of a classic 3-array CM sketch only consists of three addition operations, and therefore can be implemented within just one stage and three stateful ALUs (SALUs), where SALU is a kind of primary processing units. In contrast, the workflow of ElasticSketch[1] consists of nine addition operations, four substitution operations, and many comparisons, and therefore consumes nine SALUs and more than twelve stages.

However, an ideal sketch for efficiently compressing INT information in practice should achieve both simplicity and accuracy. Simplicity allows the sketch for flexible deployment on various hardware and software platforms, dealing with a variety of real-world network architectures. Accuracy allows the sketch to satisfy accuracy requirements within strict memory/bandwidth constraints. As discussed above, existing sketches can hardly achieve both simplicity and accuracy, and the *second design goal* of this paper is to design a simple and accurate sketch.

B. Our Proposed Solution

Towards the first design goal, our first contribution is to design **SketchINT** system to support per-flow per-switch measurement while reducing the control plane overhead of INT through combining INT and sketches. Towards the second design goal, 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. We design SketchINT, which achieves per-flow per-switch measurement

[1]The source P4 code of ElasticSketch. https://github.com/BlockLiu/ElasticSketchCode/blob/master/src/P4_cpu_implementation/elastic.p4

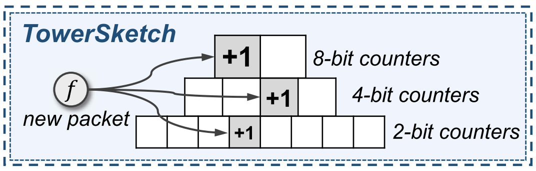


Fig. 1. TowerSketch example.

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 belonging to the control plane overhead 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 Fig. 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.* TowerSketch has three insertion strategies: 1) CM insertion similar to that of the CM sketch, 2) CU insertion similar to that of the CU sketch, and 3) approximate CU insertion similar to that of the SuMax sketch [13] (see Section IV-C). The query operation of TowerSketch is also similar to that of the CM sketch. In this way, TowerSketch can approach a state that every bit is counting. To further ease the configuration of TowerSketch and give network operators more confidence on sketch performance, inspired by SketchError [31], we extend its theory of posterior error bound estimation to be applicable to TowerSketch, so as to provide precise error bound estimation with theoretical guarantees (see Section V-B). Thanks to the simplicity of our TowerSketch, TowerSketch 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 [29], [30], single-core CPU, multi-core CPU, and FPGA.

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 up to 27.7 times lower than ElasticSketch; 2) SketchINT can reduce the number of packets belonging to the control plane overhead by $3 \sim 4$ orders of magnitude compared to INT with error smaller than 5%; 3) The estimated error bound of TowerSketch can match the actual error bound in most cases. To make our results easy to reproduce, we have released all related source codes and datasets at Github.[2]

II. Related Work

In this section, we summarize the in-band telemetry solutions and the sketching solutions for network measurement. For other solutions, please refer to reference [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51].

In-band Telemetry Solutions: These solutions insert packet-level statistics into incoming packets in INT-compatible switches. Typical in-band telemetry solutions include INT [14], [15], [16], its successor PINT [17], DeltaINT [52], and LightGuardian [13]. INT has two collection strategies: passport/INT-MD and postcard/INT-XD. In passport mode/INT-MD [14], 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/INT-XD [16], 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 [17] 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 Section VI-B), and its accuracy is also lower than INT. LightGuardian [13] 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 [18], CU [9], Count [25], CMM [53] and CSM [54]. 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 [20], ASketch [24], SF-sketch [55], and more [21], [22], [28], [56]. Specifically, SF-sketch maintains two separate sketches: a large sketch, the Fat-subsketch, and a small sketch, the Slim-subsketch. The Fat-subsketch is used for updating and periodically producing the Slim-subsketch, which is then transferred to the remote collector for answering queries quickly and accurately. Besides, there are a kind of dedicated sketches designed exclusively for specific measurement systems. Typical dedicated sketches include FlowRadar [23], Cold Filter [57], UnivMon [19], BeauCoup [58], OmniMon [59], and more [60], [61], [62], [63], [64]. Among them, FlowRadar uses a variant of Invertible Bloom filter [65] to record flow-level information. Cold Filter uses two cascaded CU sketches consisting of different-sized counters to filter small flows. UnivMon builds several sketches on the data plane to perform many measurement tasks, and uses a key method called universal streaming [66] 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. To give network operators more confidence on sketch performance, SketchError [31] proposes precise error estimation methods with theoretical guarantees for various sketches, including CM, CU and Count.

Comparison with existing per-flow per-switch measurement solutions: We compare SketchINT with existing measurement solutions that can perform per-flow per-switch measurement from five aspects. As shown in Table I, 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

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

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

TABLE I
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

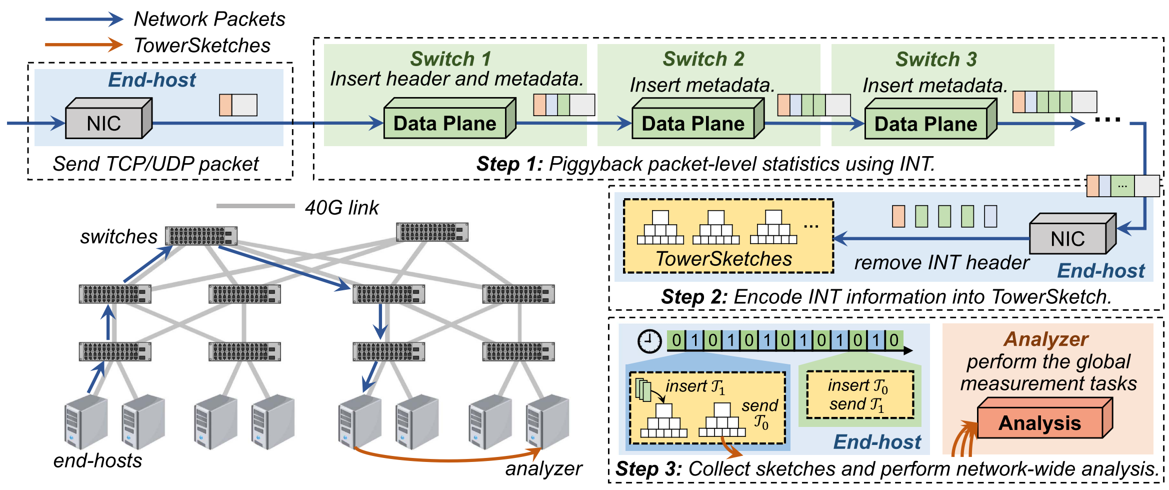


Fig. 2. Architecture and workflow of SketchINT.

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.

We further detail the pros and cons of SketchINT and LightGuardian because they are the most relevant. Compared to LightGuardian, the major advantage of SketchINT is its ease of deployment. LightGuardian requires each switch to be programmable, making it difficult to deploy. In contrast, given the fact that more and more commodity switches are becoming INT-capable [16], SketchINT only requires edge/ToR switches to be programmable at most. Further, considering that many works [67], [68], [69] are designed to optimize storage systems with edge/ToR switches, SketchINT can coexist with these works without additional changes to the network architecture. The major advantage of LightGuardian is its accuracy and bandwidth overhead when SketchINT cannot deploy TowerSketches on end-hosts. It is reasonable as LightGuardian can compress INT information into sketches on programmable switches as soon as possible, and it has much more memory for measurement as it has more programmable switches under control.

III. SKETCHINT DESIGN

As shown in Fig. 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 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. Fig. 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 Section IV) in

TABLE II
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
$F_k(R)$	The ratio of counters whose values are larger than R in array $\mathcal{A}_k$

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. Then, the SketchINT agent inserts/encodes the information carried by INT metadata (e.g., switch ID, internal latency) into TowerSketches. Optionally, the SketchINT agent forwards packets identified as belonging to large flows by TowerSketches to subsequent data structures (e.g., Space Saving [70]) to improve their accuracy.

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 [18]. Then, we show the data structure and operations of our TowerSketch. We list the main symbols in this paper and their meanings in Table II.

A. The Classic CM Sketch

The CM sketch consists of d counter arrays $\mathcal{A}_1, \ldots, \mathcal{A}_d$. 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: $\mathcal{A}_1[h_1(f)], \ldots, \mathcal{A}_d[h_d(f)]$, 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 [9] 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 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 of TowerSketch: The first key idea of our TowerSketch is to use different-sized counters for different arrays, i.e., the array at the higher level has counters larger in size. In this way, 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. Considering that the network traffic is often highly skewed, i.e., most flows are small and a small amount of large flows contribute to most traffic [9], [26], [27], the second key idea of our TowerSketch is to allocate the same amount of memory for each array, i.e., the array at the higher level has fewer counters larger in size. In this way, the array at high level has a few large counters to match a few large flows, and the array at low level has many small counters to match many small flows. Overall, TowerSketch automatically records larger flows in larger counters and smaller flows in smaller counters.

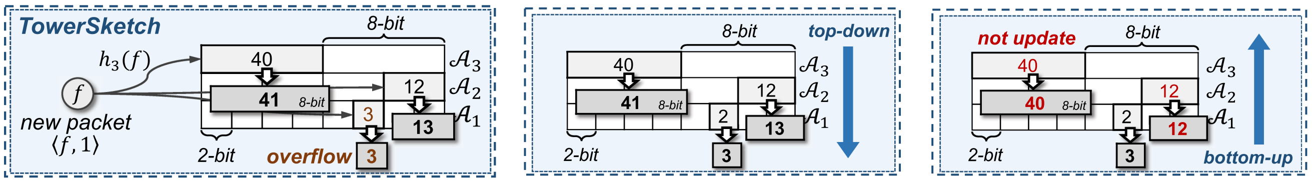
Data Structure: As shown in Fig. 3, TowerSketch consists of d arrays, $\mathcal{A}_1, \ldots, \mathcal{A}_d$. Each array $\mathcal{A}_i$ consists of w_i counters, and is associated with a hash function $h_i(.)$. The size of each counter in array $\mathcal{A}_i$ is δ_i bits. The key difference between 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 Fig. 3(a), 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: Fig. 3(a) 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 adopt the conservative update strategy of CU [9] to greatly improve the accuracy. Instead



(a) Basic Data Structure of TowerSketch. (b) Top-down access of ACU insertion. (c) Bottom-up access of ACU insertion.

Fig. 3. Examples of TowerSketch (ACU refers to Approximate CU).

of incrementing all the d hashed counters, TowerSketch only increments the smallest counter(s) that are not overflowed.

Query: The query process is the same for every insertion strategy. The query 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. Approximate CU Insertion

Although the CU insertion can greatly improve the accuracy, it does not follows a unidirectional workflow, and thus can hardly be implemented in pipeline architecture, which hinders it from being implemented in the prevalent P4-capable switches (e.g., Tofino [30]). To improve the accuracy and support pipeline implementation simultaneously, inspired by the SuMax sketch [13], we design the approximate CU (ACU) insertion specific for TowerSketch. To record a packet with flow ID f, TowerSketch maintains a current minimum value and initializes it to $+\infty$. For the d hashed counters, TowerSketch must access them in a certain order. As shown in Fig. 3(b) & (c), there are mainly two kinds of accessing order: 1) top-down access and 2) bottom-up access. *We choose the latter one: If and only if the currently accessed counter is smaller than the current minimum value, TowerSketch increments it by one, and updates the current minimum value to its value.* The reason behind is as follows. Note that the number of counters decreases as the array approaches the top, and thus the counters in higher array usually suffer more overestimation. Therefore, bottom-up access can usually obtain the real minimum value at the earliest time, and therefore is accurate. In contrast, a completely opposite order of access, i.e., top-down access, will lead to that the maintained current minimum value is usually not the real minimum value, and therefore is inaccurate. The experimental results (see Section VIII-B) also prove our insight: The accuracy of ACU insertion accessing counters in a bottom-up manner is much higher than that accessing counters in a top-down manner, even comparable with the CU insertion. Note that the query process for ACU insertion is the same as that for CM and CU insertion.

Example: Fig. 3(b) and (c) illustrate the top-down access and bottom-up access of ACU insertion, respectively. We let the counters in higher array store larger values, so as to simulate a general scenario. As shown in Fig. 3(b), in the top-down access, the maintained current minimum value is always not the real minimum value, and therefore all the d hashed counters are incremented by one ($d = 3$). In contrast, as shown in Fig. 3(c), in the bottom-up access, the current minimum value is always the real minimum value after the first access, and therefore merely the counter in the bottom array is incremented by one. Compared with the top-down access, obviously the bottom-up access increments less counters, and thus suffers from less overestimation and achieves higher accuracy.

TABLE III
PROS AND CONS OF INSERTION STRATEGIES

Strategies	Accuracy	Complexity	Pipeline implementation
CM	Low	Low	✓
ACU	Moderate	Moderate	✓
CU	High	High	×

D. Discussion

Pros and cons of insertion strategies: We list the pros and cons of each insertion strategy in Table III. For CM insertion, it is the simplest, and also supports pipeline implementation. However, it suffers from the lowest accuracy. For ACU insertion, it achieves moderate accuracy and supports pipeline implementation with moderate complexity. For CU insertion, it achieves the highest accuracy. However, it is the most complex and unfriendly to pipeline implementation. In summary, each insertion strategy has its unique pros and cons, and the network operators could choose the most appropriate one based on the actual situation.

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. A potential optimization for CM/CU sketches is to use different-sized counters for different arrays while the number of counters in each array keeps the same. However, such optimization is not enough because it ignores the skewness of network traffic as we stated in the rationale of TowerSketch, which is also verified by experiments in Section VIII-C.

V. Error Bound Analysis

Error bound estimation is of great importance for sketch configuration, as network operators usually look forward to satisfying the accuracy requirement with minimal memory overhead. In this section, we first derive the prior error bound of TowerSketch, which is independent on the network workload. Then, we derive its posterior error bound, which is dependent on the network workload and much tighter.

A. *Prior Error Bound Analysis*

In this section, we derive the prior error bound without any prior knowledge about the network workload, i.e., the worst-case error bound. Let $\delta_0 = 0$. Note that $\delta_0 < \delta_1 < \cdots < \delta_d$. Given an arbitrary flow f_j, without loss of generality, we assume its real size n_j satisfies $2^{\delta_{t-1}} - 1 \leqslant n_j < 2^{\delta_t} - 1$, 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., $n = \sum_{l=1}^{m} n_l$.

Theorem 1 (Prior Error bound): For TowerSketch using CM insertion, given an arbitrary positive number ϵ, the estimation error of flow f_j is bounded by

$$Pr\left\{\hat{n_j} \leqslant n_j + \epsilon\right\} \geqslant 1 - \prod_{k=t}^{q-1}\left\{\frac{n}{(2^{\delta_k} - n_j - 1)\cdot w_k}\right\}$$
$$\times \prod_{k=q}^{d}\left\{\frac{n}{\epsilon \cdot w_k}\right\}$$

where q satisfies that $2^{\delta_{q-1}} - 1 \leqslant n_j + \epsilon < 2^{\delta_q} - 1$.

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

$$\begin{aligned} & Pr\{\hat{n_j} \geqslant n_j + \epsilon\} \\ = \ & Pr\left\{\forall k \geqslant t, \mathcal{A}_k[h_k(f_j)] \geqslant n_j + \epsilon\right\} \\ = \ & Pr\left\{\forall k, t \leqslant k < q, n_j + X_{j,k} \geqslant 2^{\delta_k} - 1\right\} \cdot \\ & Pr\left\{\forall k \geqslant q, n_j + X_{j,k} \geqslant n_j + \epsilon\right\} \\ = \ & Pr\left\{\forall k, t \leqslant k < q, X_{j,k} \geqslant 2^{\delta_k} - n_j - 1\right\} \cdot \\ & Pr\left\{\forall k \geqslant q, X_{j,k} \geqslant \epsilon\right\} \end{aligned}$$

According to the Markov inequality, we can derive that

$$\begin{aligned} & Pr\left\{\hat{n_j} \geqslant n_j + \epsilon\right\} \\ \leqslant \ & \prod_{k=t}^{q-1}\left\{\frac{E(X_{j,k})}{2^{\delta_k} - n_j - 1}\right\} \prod_{k=q}^{d}\left\{\frac{E(X_{j,k})}{\epsilon}\right\} \\ \leqslant \ & \prod_{k=t}^{q-1}\left\{\frac{n}{(2^{\delta_k} - n_j - 1)\cdot w_k}\right\} \prod_{k=q}^{d}\left\{\frac{n}{\epsilon \cdot w_k}\right\} \end{aligned}$$

Therefore, we have

$$\begin{aligned} & Pr\left\{\hat{n_j} \leqslant n_j + \epsilon\right\} \\ \geqslant \ & 1 - \prod_{k=t}^{q-1}\left\{\frac{n}{(2^{\delta_k} - n_j - 1)\cdot w_k}\right\} \prod_{k=q}^{d}\left\{\frac{n}{\epsilon \cdot w_k}\right\} \end{aligned}$$

□

From Theorem 1, we can see that the smaller flow goes with the smaller n_j, as well as the smaller t, q. Therefore, we can conclude that the prior error bound is dependent on the flow size, and the smaller flows have the smaller error in TowerSketch.

B. *Posterior Error Bound Analysis*

Traditionally, network operators configure the sketch based on the the worst-case error bound derived from traditional workload-independent analysis [18]. However, the actual error bound is strongly dependent on the network workload, and thus could be much tighter than worst-case error bound, especially when the workload is highly skewed [20], [24]. Compared to the tighter actual error bound, the worst-case error bound seriously undermines the confidence of network operators on sketch accuracy, resulting in a large amount of meaningless memory waste. To address this issue, the state-of-the-art work, SketchError [31], has provided posterior, near-optimal, and unified error bound estimation at the query time for the classic CM sketch with theoretical guarantees. We aim at extending its theory and error bound estimation method to be applicable to TowerSketch, so as to ease the configuration and give network operators more confidence on the performance of TowerSketch. Different from SketchError, our insight is that the posterior error bound of TowerSketch should not be unified for all flows, but strongly dependent on the flow size, because different-sized flows are automatically recorded in different arrays consisting of different-sized counters in TowerSketch. In this section, we first derive the posterior error bound of TowerSketch, then propose how to estimate the error bound for flows of arbitrary sizes.

Theorem 2 (Posterior Error bound): Given a TowerSketch using CM insertion after the insertion process, an arbitrary positive number ϵ, the estimation error of flow f_j is bounded by

$$Pr\left\{\hat{n_j} \leqslant n_j + \epsilon\right\} \approx 1 - \prod_{k=t}^{q-1} F_k(2^{\delta_k} - 2 - n_j) \prod_{k=q}^{d} F_k(\epsilon)$$

where q satisfies $2^{\delta_{q-1}} - 1 \leqslant n_j + \epsilon < 2^{\delta_q} - 1$, and $F_k(R)$ denotes the ratio of counters whose value are larger than R in array $\mathcal{A}_k$.

Proof: For array $\mathcal{A}_k$ of TowerSketch, the value of each counter in it could be viewed as a sample of the same random variable Y_k.

$$Y_k = \begin{cases} \sum_{l=1}^{m} n_l M_{k,l}, & \sum_{l=1}^{m} n_l M_{k,l} < 2^{\delta_k} - 1 \\ +\infty, & \text{otherwise} \end{cases}$$

Here, $M_{k,l}$ denotes the probability that flow n_l is hashed to a certain counter in array $\mathcal{A}_k$. Obviously, $M_{k,l}$ is a 0/1 variable, and $Pr\{M_{k,l}=1\}=\frac{1}{w_k}$.

Based on the counter size, We classify each array of TowerSketch into three kinds.

The first kind of arrays are those whose counters are naturally overflowed by n_j. For any array $\mathcal{A}_k$ of the first kind ($\forall k < t, \mathcal{A}_k$), Obviously, We have

$$Pr\{\mathcal{A}_k[h_k(f_j)] > n_j + \epsilon\} = 1$$

The second kind of arrays are those whose counters are not overflowed by n_j, but overflowed by $n_j+\epsilon$. For any array $\mathcal{A}_k$ of the second kind ($\forall k, t \leqslant k < q, \mathcal{A}_k$), considering that each counter in it is a sample of the same random variable Y_k, according to Bernoulli's law of large numbers, we have

$$\begin{aligned} Pr\{\mathcal{A}_k[h_k(f_j)] > n_j + \epsilon\} &= Pr\{Y_k > 2^{\delta_k} - 2 - n_j\} \\ &\approx F_k(2^{\delta_k} - 2 - n_j) \end{aligned}$$

The third kind of arrays are those whose counters are not overflowed by $n_j+\epsilon$. For any array $\mathcal{A}_k$ of the third kind ($\forall k, q \leqslant k \leqslant d, \mathcal{A}_k$), similarly, according to Bernoulli's law of large numbers, we have

$$Pr\{\mathcal{A}_k[h_k(f_j)] > n_j + \epsilon\} = Pr\{Y_k > \epsilon\} \approx F_k(\epsilon)$$

Therefore, we can derive that

$$\begin{aligned} Pr\left\{\hat{n_j} \leqslant n_j + \epsilon\right\} &= 1 - \prod_{k=1}^{d} Pr\{\mathcal{A}_k[h_k(f_j)] > n_j + \epsilon\} \\ &\approx 1 - \prod_{k=t}^{q-1} F_k(2^{\delta_k} - 2 - n_j) \prod_{k=q}^{d} F_k(\epsilon) \end{aligned}$$

□

From Theorem 2, we can see that the smaller flow goes with the smaller n_j, as well as the smaller t, q, and $F_k(2^{\delta_k} - 2 - n_j)$. Therefore, we can conclude that the posterior error bound is dependent on the flow size, and the smaller flows have the smaller error in TowerSketch.

Error bound estimation for TowerSketch using CM insertion: For TowerSketch using CM insertion, we directly use the right-hand side formula of Theorem 2 to estimate its posterior error bound. The correctness of this method is guaranteed by Theorem 2, and we further verify its correctness with experiments (see Section VIII-E).

C. Discussion

Both Theorems 1 and 2 apply to estimating the overestimation error on large flows (also small flows). Indeed, TowerSketch uses less counters to estimate the sizes of true large flows, and as Theorems 1 and 2 demonstrate, it leads to larger overestimation error on large flows than small flows. However, TowerSketch prevents most small flows from being overestimated as fake large flows because of its accurate estimation of small flows. Therefore, TowerSketch achieves great performance on measurement tasks that focus on large flows. As shown in Fig. 7(b)(c)(d)(h), TowerSketch achieves higher accuracy than CM/CU sketches and at least comparable accuracy with the state-of-the-art on measurement tasks that focus on large flows.

VI. Measurement Tasks

In this section, we elaborate on how SketchINT performs seven representative local measurement tasks and four 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., path tracing). SketchINT is also perfectly compatible with other INT-based mechanism (e.g., HPCC [12]).

A. Local Measurement Tasks

This subsection presents seven 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, 6) cardinality estimation, and 7) large flow marking. Although existing network measurement solutions support these tasks, TowerSketch significantly improves the accuracy for 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] Flow sizes characterize the flows and provide the basis for advanced tasks, such as heavy hitter detection and heavy change detection. 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. Heavy hitters can provide guidance for traffic engineering and attack detection [71], [72]. 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. Heavy changes are indications of traffic anomalies [73]. 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. Flow size distribution can provide guidance for traffic engineering and attack detection [74]. The traditional flow size distribution estimation algorithm is MRAC [74], which can estimate the flow size distribution with a single counter array. MRAC uses the Expectation Maximization (EM) algorithm [75] to provide distribution estimation for flows of any size. To

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

estimate the flow size distribution for TowerSketch, a straightforward solution is to apply MRAC to the top array of TowerSketch. However, MRAC is not applicable to the other counter arrays with smaller counters, as counters in them could be overflowed. This leads TowerSketch to lack accuracy in the estimation of the distribution of small flows. To address this issue, we slightly modify the basic MRAC algorithm to enable it to be applicable to the other arrays besides the top array. Specifically, for array $\mathcal{A}_i$, instead of taking flows of any size into consideration as the basic MRAC algorithm, we only divide the flows into 1) flows with size in range $[1, 2^{\delta_i} - 1)$ and 2) flows with size reaching or exceeding $2^{\delta_i} - 1$. In this way, we can apply the modified MRAC algorithm to any array in TowerSketch. The detailed workflow is as follows. We apply the modified MRAC algorithm to all counter arrays of TowerSketch from bottom to top. Each array $\mathcal{A}_i$ could provide the estimated distribution for flows with size in range $[1, 2^{\delta_i} - 1)$, and we use the estimated distribution for flows with size in range $[2^{\delta_{i-1}} - 1, 2^{\delta_i} - 1)$ as the input distribution for the higher arrays. The input distribution will not be recalculated in the process of EM. In other words, array $\mathcal{A}_{i+1}$ will estimate the distribution for flow size in range $[2^{\delta_i} - 1, 2^{\delta_{i+1}} - 1)$ with the input distribution in range $[1, 2^{\delta_i} - 1)$. Here, we fix the flow distribution for small flows as we believe lower arrays can provide more accurate distribution for small flows than higher arrays, because lower arrays have much more counters. After the workflow ends, we can finally obtain the complete flow size distribution.

Entropy estimation: estimating the entropy of flow sizes. Entropy can provide guidance for anomaly detection [76]. After getting the estimation of flow size distribution, we can easily compute the entropy by the following formula: $-\sum(m_i \cdot \frac{i}{M} \log \frac{i}{M})$, 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 most counters to estimate cardinality, and calculate the results using the linear counting algorithm [77].

Large flow Marking: identifying flows whose sizes are larger than a threshold Δ_f, and marking their packets as belonging to large flows. Note that there is a fundamental difference between large flow marking and heavy hitter detection: large flow marking is at packet-level and needs to identify whether a packet belongs to large flows at real time, while heavy hitter detection is at flow level and reports heavy hitters after a measurement window ends. Large flow marking is an important task in data stream processing, as it can greatly improve the accuracy for data structures focusing on large flows [57]. TowerSketch can support large flow marking independently. For every incoming packet, we query its flow size from TowerSketch. If the queried flow size exceeds Δ_f, we mark this packet as belonging to large flows.

B. Global Measurement Tasks

This subsection presents four 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 can reuse the TowerSketch built for local measurement tasks in each host.

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 reuses the TowerSketch built for local tasks (flow-size TowerSketch) and builds an additional TowerSketch to record the latency information (latency TowerSketch). For each incoming packet, we first insert it to flow-size TowerSketch with its flow ID as the key. Then, we 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 counters in the latency TowerSketch. For each hashed counter, we increment it by the internal latency in S_i. 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 flow-size TowerSketch for its flow size with its flow ID as the key, query latency TowerSketch for its total latency with the concatenated key formed by its flow ID and S_i, 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 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 process it as described in latency estimation. 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 reuses the TowerSketch built for local tasks and builds an additional 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.

VII. 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

1) Standard Version: 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 measurement tasks supported by TowerSketch. 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 in Tofino switches unless recirculating most of the packets, which is quite expensive in terms of bandwidth. Therefore, we only implement TowerSketch using CM insertion and ACU insertion on Tofino switches through several registers and Stateful ALUs (SALU). For each counter array $\mathcal{A}_i$ consisting of w_i δ_i-bit 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,[5] we use the register elements that are slightly larger than δ_i-bit 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.

[5]Tofino switch also supports 1-bit register elements. However, the capability of 1-bit elements is limited to being set to 0 or 1, and thus cannot be used as counters.

TABLE IV
Resources Usage of TowerSketch&SketchINT Agent

Resource types	Baseline	TowerSketch	SketchINT agent
Stateful ALU	16	3(18.75%)	18(112.5%)
VLIW Actions	82	5(6.10%)	34(41.46%)
TCAM	145	0(0.00%)	11(7.59%)
SRAM	562	15(2.67%)	73(12.99%)
Hash Bits	1851	93(5.02%)	300(16.21%)
Ternary Crossbar	409	0(0.00%)	44(10.76%)
Exact Crossbar	371	26(7.01%)	342(92.18%)

2) Extension: Extension to 2-bit counters: We can also support 2-bit counters by using multiple register elements. Specifically, we can use three cascaded 1-bit register elements to simulate a 2-bit counter. For insertion, we set the the first zero 1-bit register element to one. For query, we report the sum of the three 1-bit register elements as the value of the 2-bit counter. Obviously, the three cascaded 1-bit register elements are equivalent to a 2-bit counter in terms of function. The price we pay for simulating 2-bit counters is mainly twofold: 1) 50% additional memory overhead; 2) two more SALUs for accessing the three 1-bit register elements compared to 8-bit counter that only accesses one 8-bit register element. The extension to 2-bit counters is of great value, especially when the network traffic is highly skewed: recording flows of size one or two with 2-bit counters is much more memory-efficient than 8-bit counters. This extension is not only applicable for TowerSketch, but also other sketches that rely on small-sized counters, such as FCMSketch [78], PyramidSketch [79], and Cold Filter [57].

Extension to larger counters: We can also support counters larger than 32-bit by using multiple register elements. Taking a 48-bit counter as an example, we simulate it with a pair of 32-bit and 16-bit register elements. From the output of 32-bit element, we can get the flow size and overflow information, then access the 16-bit element. Combining two outputs from the elements, we can simulate a 48-bit counter. Obviously, this basic idea could be extended to support counters of any size that are larger than 32-bit.

3) Resource Usage: We compare the resource usage of our standard-version 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. There is almost no difference in resource usage between the implementations of ACU insertion and CM insertion, except that the SALUs for CM insertion can be placed in one stage, while that for ACU insertion cannot, because there are dependencies between arrays in ACU insertion. Both the implementations of large counters and 2-bit counters will increase the usage of SALUs, as they require more logic as well as SALUs to simulate these counters.

We further compare the resource usage of a P4-based SketchINT agent that supports all global tasks to `switch.p4`. Overall, to perform all global measurement tasks, we require only two TowerSketches (one for storing flow size information and one for storing latency information) and two hash tables (one for storing flows suffering from inflated latency and one for storing per-switch heavy-hitter). We show the resource usage of a P4-based SketchINT agent in Table IV. The agent can support a typical data center network that has at most five hops. The parameters of the TowerSketch for recording flow size information are: $d = 3$, $\delta_i = 2^{i+2}$, and $w_i = 2^{17-i}$. The parameters of the TowerSketch for recording latency information are: $d = 2$, $\delta_i = 2^{i+3}$, and $w_i = 2^{15-i} \times 5$. Both the two hash tables have 1024 table entries to record the entire 104-bit flow ID (5-tuple) and additional required information (e.g., forwarding path). We find the additional usage of SALU and Exact Crossbar is relatively high. This is because that the TowerSketch for recording latency information requires to insert the latency of each packet in its each passing switch (at most five), and therefore involves a great number of register actions and memory I/O, which are just related to these two kinds of resources.

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 [80] 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 Section VIII-C).

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.

VIII. Experimental Results

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

- **Which access order achieves highest accuracy in ACU insertion?** We evaluate the accuracy of TowerSketch using ACU insertion that accesses counters in top-down and bottom-up manners on several well-known datasets.
- **How accurate can TowerSketch perform the 7 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/INT-MD [14].
- **How the estimated error bound of TowerSketch match the actual error bound?** We evaluate our proposed error estimation method of TowerSketch on several well-known datasets.

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) :* $\frac{\sum_{i=1}^{z}|m_i-\hat{m_i}|}{\sum_{i=1}^{z}(\frac{m_i+\hat{m_i}}{2})}$, 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 [81].
- *Throughput:* Million packets per second (Mpps).

Datasets:

- *CAIDA dataset:* The CAIDA dataset is built from the anonymized IP traces collected in 2018 from CAIDA [82], which is widely used in prior work [81], [83]. We use the 32-bit source IP address as the flow ID. Each trace contains about 2.3 M packets of 170 K flows, with a monitoring time interval of 5 s.
- *Webpage dataset:* The Webpage dataset is built from a collection of web pages [84], and so called Webpage dataset. The dataset contains about 2 M packets of 130 K flows.
- *Synthetic datasets:* The synthetic datasets are generated following the widely recognized Zipf distribution [85]. The skewness of the two datasets are 0.5 and 1.0, and so called Zipf_0.5 dataset and Zipf_1.0 dataset, respectively.

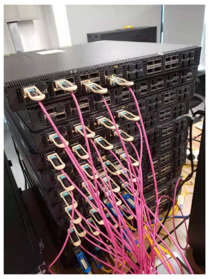


Fig. 4. Testbed.

Zipf_0.5 dataset contains about 2 M packets of 900 K flows, and Zipf_1.0 dataset contains about 2 M packets of 350 K flows.

A. Testbed Setup

As shown in Fig. 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/INT-MD, the mirrored packets at sink switches are considered without payload.

B. Experimental Results on ACU Insertion

Experimental setup: For TowerSketch, we evaluate the accuracy of the two main access orders of ACU insertion, i.e., bottom-up access and top-down access, on the most basic measurement task, flow size estimation, so as to determine the better one. We also compare them with TowerSketch using CM insertion and CU insertion, regarding their accuracy as references. 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. The experiments are conducted on CAIDA, Webpage, Zipf_0.5, and Zipf_1.0 datasets to demonstrate the generality of the results. All experiments are repeated 100 times and the average results are reported.

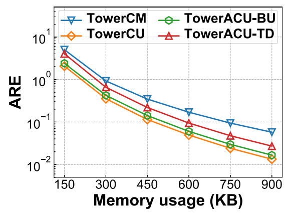
Flow size estimation (Fig. 5(a)–(d)): We find that the average ARE of TowerACU-BU is 1.68 times lower than TowerACU-TD. The results show that when using 900 KB of memory, TowerACU-BU achieves at least 1.56 times lower ARE than TowerACU-TD, at least 2.45 times lower ARE than TowerCM, and up to 1.24 times higher ARE than TowerCU.

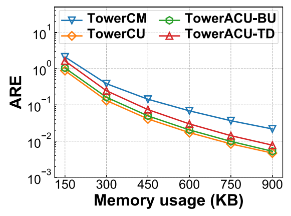
Summary: The experimental results on the four datasets demonstrate that TowerACU-BU outperforms TowerACU-TD and approaches the accuracy of TowerCU. The reason behind is as follows. The number of counters decreases as the array nears the top, so counters in higher arrays generally suffer more overestimation. Thus, bottom-up access can often obtain real minimum value at the earliest time, and thus achieve almost the same accuracy as CU insertion. In contrast, top-down access can hardly obtain real minimum value at the earliest time, and thus cannot approach the accuracy of CU insertion. *For the highest accuracy, in the rest experiments, ACU insertion only accesses counters in a bottom-up manner.*

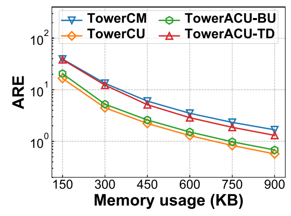
C. Experimental Results on Local Tasks

Experimental setup: We compare TowerSketch with the most widely used CM and CU sketches, NitroSketch, UnivMon, and the state-of-the-art ElasticSketch. In addition, we compare TowerSketch with the variants of CM and CU sketches that use different-sized counters for each array while the number of counters in each array keeps the same, denoted by CM(O) and CU(O), respectively. 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(O) and CU(O), their only difference from TowerSketch is that they allocate the same number of counters for each array. For CM and CU, we use 3 hash functions as recommended in literature [86]. For ElasticSketch and UnivMon, we set its parameters as the original paper [19], [20] recommends. For NitroSketch, we set depth $d = 3$ and geometry sampling rate $p = 1.0$ to achieve highest accuracy [21]. We set the capacity of the hash table used in heavy change detection and heavy hitter detection to 1024. We use the famous BobHash [87] for all sketches. All experiments are repeated 100 times on CAIDA dataset 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.
- *Large flow marking:* We set the large flow threshold $\Delta_f = 500$, about 0.02% of the total packets. Instead of ElasticSketch, we compare TowerSketch with Cold Filter, the state-of-the-art large flow marking algorithm. For Cold Filter, we set it to consist of two cascading 3-layer CU sketches, one of which consists of 4-bit counters, and the other of which consists of 16-bit counters, as recommended in literature [57]. We evaluate the performance of each algorithm with a score which regards the performance of a zero-error hash table as the baseline, i.e., the score is always zero for the hash table. The hash table marks every packet whose current flow size is larger than Δ_f as belonging to large flows. Each algorithm marks a packet as belonging to large flows if its estimated flow size exceeds Δ_f. If the algorithm correctly marks a packet as belonging to large flows, and this packet is not marked by the hash table, we increment the score by one. If the algorithm wrongly marks a packet as belonging to large flows, or misses a packet marked as belonging to large flows by the hash table, we decrement the score by one.


(a) CAIDA.


(b) Webpage.


(c) Zipf_0.5.

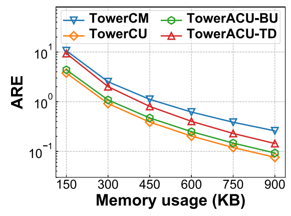
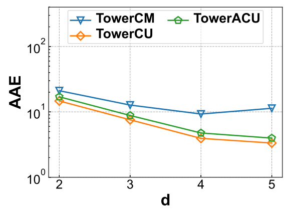

(d) Zipf_1.0.

Fig. 5. Performance (ARE) of TowerSketch using different insertion strategies on flow size estimation, where **TowerCM** represents the TowerSketch using CM insertion, **TowerCU** represents the TowerSketch using CU insertion, **TowerACU-BU** represents the TowerSketch using ACU insertion that accesses counters in a bottom-up manner, and **TowerACU-TD** represents the TowerSketch using ACU insertion that accesses counters in a top-down manner.


(a) AAE *v.s.* d.

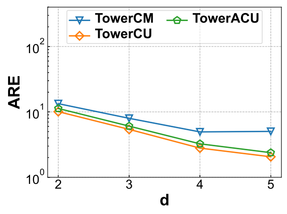

(b) ARE *v.s.* d.

Fig. 6. Accuracy of TowerSketch on flow size estimation versus d under 150 KB of memory. We set $\delta_i = 2^{i+5-d}$ for $\forall i \in [1, d]$.

- *Processing speed evaluation:* We conduct the flow size estimation experiments on single-core CPU and multi-core CPU. We allocate 2 MB of memory to each algorithm. To enlarge the subtle difference between different algorithms, we use 5 hash functions for Tower, CM, CU and NitroSketch. On multi-core CPU, we implement both the lock version and the lock-free (LF) version of our TowerSketch.

Accuracy versus d (Fig. 6): For TowerSketch, as Fig. 6(a)–(b) show, we measure its accuracy with different d on the most fundamental task, i.e., flow size estimation, and we find that $d = 5$ works best. Here, $d = 5$ works best because a great number of flows in CAIDA dataset have less than 3 or 15 packets, making 2-bit and 4-bit counters powerful.

Flow size estimation (Fig. 7(a)): We find that the average AAE of TowerCU is much lower tahan all competitors. When using 900 KB of memory, the AAE of TowerCU is 0.021, while that of TowerCM, TowerACU, CM, CU, CM(O), CU(O), NitroSketch, UnivMon and ElasticSketch are 0.296, 0.026, 2.482, 1.285, 0.558, 0.130, 9.370, 3.837, 0.576, respectively. TowerCU achieves 27.3 times lower AAE than ElasticSketch. Note that the AAE of CM(O) and CU(O) is larger than TowerCM and TowerCU and better than CM and CU, respectively, which demonstrates the rationale of TowerSketch.

Heavy hitter detection (Fig. 7(b)–(c)): We find that TowerCU always achieves better F_1 score and ARE than all competitors. When using 300 KB of memory, the F_1 score of TowerCU is 0.999, while that of TowerCM, TowerACU, CM, CU, CM(O), CU(O), NitroSketch, UnivMon and ElasticSketch are 0.951, 0.998, 0.974, 0.997, 0.979, 0.999, 0.825, 0.947, 0.997, respectively. And the ARE of TowerCU is 0.0003, while that of TowerCM, TowerACU, CM, CU, CM(O), CU(O), NitroSketch, UnivMon and ElasticSketch are 0.028, 0.0005, 0.015, 0.001, 0.012, 0.0006, 0.036, 0.019, 0.001, respectively.

Heavy change detection (Fig. 7(d)): We find that TowerCU achieves better F_1 score than all competitors. When using 300 KB of memory, the F_1 score of TowerCU is 0.997, while that of TowerCM, TowerACU, CM, CU, CM(O), CU(O), NitroSketch, UnivMon and ElasticSketch are 0.961, 0.996, 0.959, 0.980, 0.979, 0.996, 0.528, 0.852, 0.990, respectively.

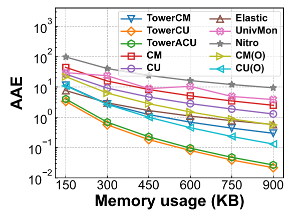
Entropy estimation (Fig. 7(e)): We find that the average RE of TowerCU is much times lower than all competitors. When using 900 KB of memory, the RE of TowerCU is 0.0002, while that of TowerCM, TowerACU, CM, CU, CM(O), CU(O), NitroSketch, UnivMon and ElasticSketch are 0.003, 0.0002, 0.009, 0.007, 0.003, 0.001, 0.025, 0.01, 0.003, respectively.

Cardinality estimation (Fig. 7(f)): We find that Tower achieves comparable accuracy with ElasticSketch and better accuracy than other competitors. When using 900 KB of memory, the RE of TowerCU is 0.0006, while that of TowerCM, TowerACU, CM, CU, CM(O), CU(O), and ElasticSketch are 0.0006, 0.0005, 0.002, 0.001, 0.002, 0.002, 0.0006, respectively. We do not compare with UnivMon and NitroSketch as they do not discuss how to estimate cardinality.

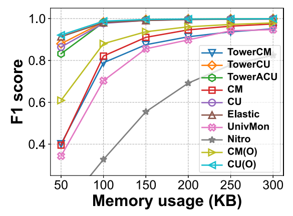
Flow size distribution estimation (Fig. 7(g)): We find that the average WMRE of TowerCM is 5.6 times lower than CM and CU. When using 900 KB of memory, the WMRE of TowerCU is 0.115, while that of TowerCM, TowerACU, CM, CU, CM(O), CU(O), and ElasticSketch are 0.045, 0.098, 0.296, 0.246, 0.197, 0.223, 0.008, respectively. We do not compare with UnivMon and NitroSketch as they do not discuss how to estimate flow size distribution.

Large flow marking (Fig. 7(h)): We find that TowerCU and TowerACU can achieve comparable score with the hash table when using less memory than CF, CM, and CU. The results show that when using 20 KB of memory, the scores of TowerCU and TowerACU are 4418 and -4917, respectively, while those of CF, CM, and CU are -78462, -471484, and -151300, respectively.

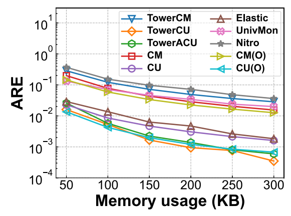
Speed on CPU (Fig. 8(a)–8(e)): We find that Tower achieves comparable processing speed with CM and CU on single-core CPU. The results show that the throughput of TowerCM is 22.6Mpps, while that of CM and CU are 27.1Mpps and 21.7Mpps, respectively. Among the eight algorithms, ElasticSketch achieves the 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



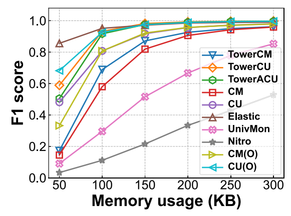
(a) Flow size estimation (AAE).



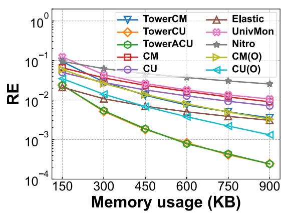
(b) Heavy hitter detection (F_1).



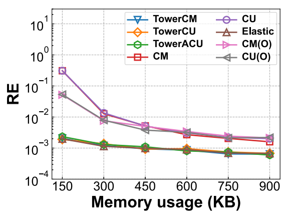
(c) Heavy hitter detection (ARE).



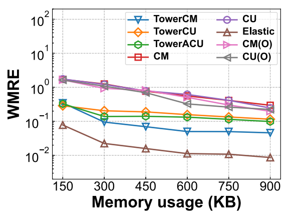
(d) Heavy change detection.



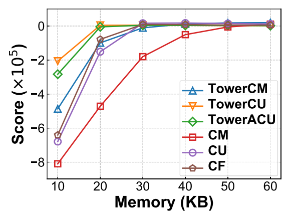
(e) Entropy estimation.



(f) Cardinality estimation.

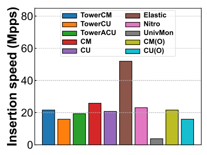


(g) Flow size distribution.

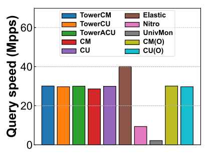


(h) Large flow marking.

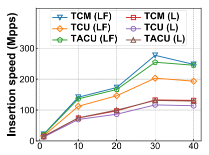
Fig. 7. Performance on local measurement tasks, where **TowerACU** represents the TowerSketch using ACU insertion, **CF** represents Cold Filter.



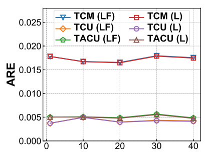
(a) Single-core insertion speed.



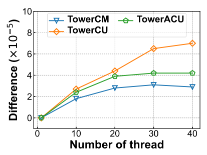
(b) Single-core query speed.



(c) Multi-core insertion speed.



(d) ARE *v.s.* number of thread.

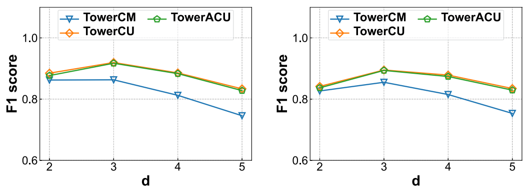


(e) Difference *v.s.* number of thread.

Fig. 8. Processing speed on single-core CPU and multi-core CPU, where **T** represents Tower, **L** represents the lock version, and **LF** represents the lock-free version.

incurred by concurrency is negligible. The results show that when using 30 threads, the lock-free version of TowerCM can reach a throughput of 277Mpps, which is 2.1 times higher than the lock version. The reason behind is that the concurrency issues caused by the lock-free version are actually not significant. We evaluate the relative difference of the sum of all counters of TowerSketch between the lock version and lock-free version, denoted by **difference** in Fig. 8(e). As shown in Fig. 8(e), even with forty threads, the relative difference is only 7×10^{-5}, and therefore has little impact on the accuracy. This is intuitively reasonable as when real NIC works, it will automatically assign packets to different cores/threads for load balancing according to the hash value calculated from their 5-tuples, and therefore the packets of the same flow will be naturally assigned to the same core/thread. As a result, packets of the same flow do not lead to concurrency issues because they are processed by the same core/thread in order, and packets of different flows rarely lead to concurrency issues because they are randomly hashed. Our experiment simulates the workflow of NIC and therefore leads to the similar result.

Summary: In summary, compared with prior arts (including UnivMon, NitroSketch, CM(O), and CU(O)), 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. As for speed on CPU platform, the throughput of TowerCM and TowerCU is slightly below that of CM and CU, respectively. This is because building 2-bit and 4-bit counters require bitwise operations, and thus consume more CPU resources. TowerACU achieves moderate throughput between TowerCM and TowerCU, and this is because its complexity is between TowerCM and TowerCU. UnivMon is the slowest because it needs to visit multiple levels (Count sketches) for each insertion. The insertion speed of NitroSketch is similar to CM because we set its sample rate to 1 that all packets are processed by NitroSketch. But its query speed is slower than CM because it requires an additional sorting operation. The speeds of CM(O) and CU(O) are similar to TowerCM and TowerCU respectively, which is easy to understand because their insert/query operations are the same. While ElasticSketch seems to have the most complicated workflow, it achieves the highest throughput. The reason behind is two-fold. First, due to the skewed network traffic, most packets belong to large flows, and therefore their insertions end in the heavy part and do not go through the whole workflow. Note that ElasticSketch uses heavy part and light part to separately record large flows and small flows. The insertion to the heavy part of ElasticSketch involves the computation of only one hash function to locate the hashed bucket in the heavy part, while the insertion of TowerSketch involves the computation of five hash functions considering that there are five counter arrays. Second, ElasticSketch uses SIMD instructions to further accelerate the computation in the hashed



(a) Per-switch heavy hitter. (b) Per-switch heavy change.

Fig. 9. Accuracy of TowerSketch on per-switch heavy hitter/change detection versus d under 20 KB of memory. We set $\delta_i = 2^{i+5-d}$ for $\forall i \in [1, d]$.

bucket. 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.

D. Experimental Results on Global Tasks

Experimental setup: In our SketchINT system, we configure the eight end-hosts to send and receive traffic at the same time, and they use the Traffic Generator [88] to generate the traffic under DCTCP [89] 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 s. Each trace contains about 16 M packets and 73 K flows, and each packet passes 4.06 switches on average. For each packet, we assume INT-MD mirrors its 64 B packet header and all its carried INT information (16-bit switch ID and 32-bit latency per switch) to the analyzer at INT-sink switches. The overall bandwidth usage and packet number belonging to the control plane overhead of INT-MD is 1.5 GB and 16 M, respectively.

Since ElasticSketch does not naturally support the global tasks, we just compare TowerSketch with CM, CU, CM(O), and CU(O). For TowerSketch, we set $d = 3$, and $\delta_i = 2^{i+2}$ for $\forall i \in [1, 3]$, and allocate the same amount of memory for each array with different counter size. For CM(O) and CU(O), their only difference from TowerSketch is that they allocate the same number of counters for each array. For CM and CU, we still use 3 hash functions. For per-switch heavy hitter detection, we set $\Delta_h = 1500$. For per-switch heavy change detection, we set $\Delta_c = 750$. For per-flow per-switch inflated latency detection, we set $\Delta_l = 5$. To perform latency tasks, for each sketch, we build an additional sketch of the same type with the same number of arrays to record latency information. We allocate $\frac{3}{4}$ of the total memory to the additional sketch considering that each packet could pass multiple switches and have multiple latency. For the additional TowerSketch/CM(O)/CU(O), we set $\delta_i = 20 + 4i$ for $\forall i \in [1, 3]$. For the additional CM/CU, we still use 32-bit counters.

Accuracy versus d (Fig. 9): For TowerSketch, as Fig. 9(a)–(b) show, we measure its accuracy with different d on per-switch heavy hitter/change detection, and we find that $d = 3$ works best. Here, $d = 3$ works better than $d = 5$ because flows in this trace follow DCTCP distribution, and are relatively larger than flows in CAIDA dataset, making 2-bit and 4-bit counters useless.

Per-switch heavy hitter detection (Fig. 10(a)): We find that TowerCU is more accurate than CM, CU, CM(O), and CU(O). When using 10 KB of memory, the F_1 score of TowerCU is 0.726, while that of CM, CU, CM(O), and CU(O) are 0.438, 0.517, 0.607, and 0.680, respectively. The F_1 score of TowerCU reaches 0.99 under 50 KB of memory.

Per-switch heavy change detection (Fig. 10(b)): We find that TowerCU is more accurate than CM, CU, CM(O), and CU(O). Under 10 KB of memory, the F_1 score of TowerCU is 0.708, while that of CM, CU, CM(O), and CU(O) are 0.475, 0.548, 0.610, and 0.660, respectively. Under 50 KB of memory, the F_1 score of TowerCU reaches 0.98.

Latency estimation (Fig. 10(c)): We find that TowerCU is more accurate than CM, CU, CM(O), and CU(O). When using 3 MB of memory, the ARE of TowerCU is 0.1038, while that of CM, CU, CM(O), and CU(O) are 0.1440, 0.1408, 0.1068, and 0.1049, respectively.

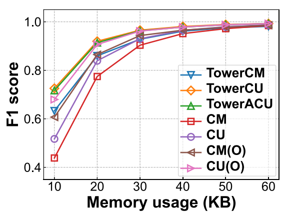
Inflated latency detection (Fig. 10(d)): We find that TowerCU always achieves better F_1 score than CM, CU, CM(O), and CU(O). When using 100 KB of memory, the F_1 score of TowerCU is 0.814, while that of CM, CU, CM(O), and CU(O) are 0.768, 0.769, 0.799, and 0.803, respectively.

Bandwidth overhead and packet number comparison (Fig. 13(a)–(b)): We find that, SketchINT can achieve almost the same accuracy as INT while reducing the bandwidth usage and the number of packets that belong to the control plane overhead to 3% and by $3 \sim 4$ orders of magnitude, respectively. 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 usage belonging to the control plane overhead of SketchINT to that in the standard INT passport mode/INT-MD. We use *Normalized Packet Number (NPN)* to represent the ratio of the number of packets belonging to the control plane overhead of SketchINT to that in standard INT passport mode/INT-MD. The results show that TowerSketch requires only 3% NB and 0.03% NPN to achieve 1.5% ARE on the latency estimation task. In contrast, CM/CU requires 4% NB and 0.04% NPN, indicating a 33% additional overhead.

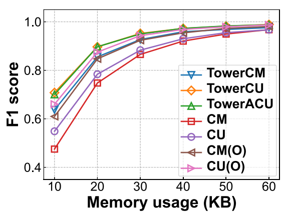
Summary: In summary, SketchINT achieves high accuracy with low network overhead because TowerSketch can efficiently encode per-flow information. While TowerSketch can achieve best performance, normal CM/CU sketches are still realistic choices when sketches are deployed on end-hosts with abundant memory. However, when deployed on hardware platforms with only limited memory, such as programmable switches, normal CM/CU sketches may be not realistic as they may not satisfy accuracy requirements when facing high-volume traffic under strict memory constraints.

E. Experimental Results on Posterior Error Bound

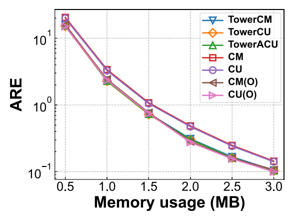
Experimental setup: We evaluate the error estimation method for TowerSketch using CM insertion. For TowerSketch, we set $d = 5$, and $\delta_i = 2^i$ for $\forall i \in [1, 5]$. We allocate the same amount of memory for each array with different counter size. The experiments are conducted on CAIDA, Webpage, Zipf_0.5,



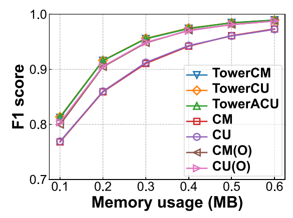
(a) Per-switch heavy hitter.



(b) Per-switch heavy change.

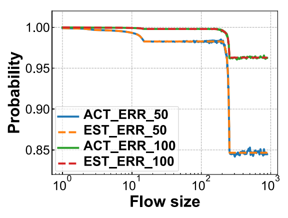


(c) Latency estimation.

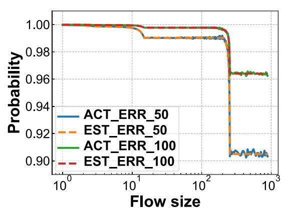


(d) Inflated latency detection.

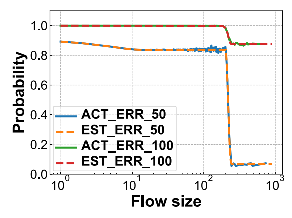
Fig. 10. Performance on global measurement tasks.



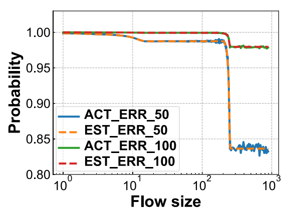
(a) CAIDA.



(b) Webpage.

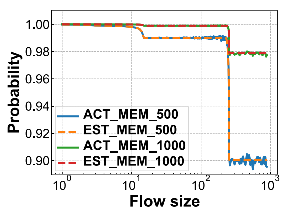


(c) Zipf_0.5.

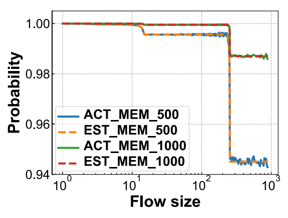


(d) Zipf_1.0.

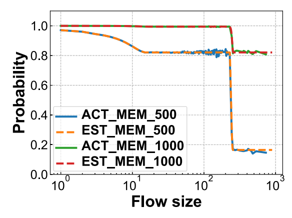
Fig. 11. Estimated error bound versus actual error bound for error of different sizes, where **ACT_ERR_x** and **EST_ERR_x** represent the actual and estimated probability that the error is smaller than x, respectively.



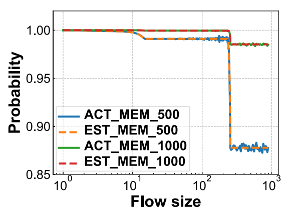
(a) CAIDA.



(b) Webpage.



(c) Zipf_0.5.



(d) Zipf_1.0.

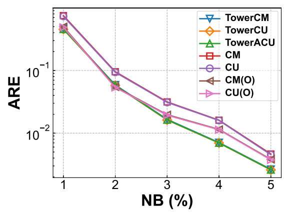
Fig. 12. Estimated error bound versus actual error bound under different memory usage, where **ACT_MEM_x** and **EST_MEM_x** represent the actual and estimated probability that the error is smaller than 25 when using xKB memory, respectively.

and Zipf_1.0 datasets to demonstrate the generality of the results. All experiments are repeated 10 K times to make the actual error bound stable.

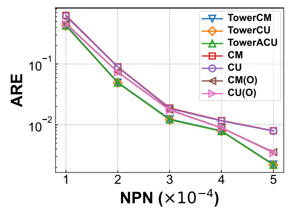
Estimated error bound versus actual error bound for error of different sizes (Fig. 11(a)–(d)): We find that the estimated error bound can match the actual error bound for error of different sizes on all the four datasets, and the larger flow suffers from larger estimated error. We allocate 250 KB for TowerCM in the experiments. On CAIDA dataset, for flows of size 1000, the probabilities that its estimated error is less than 50 and 100 are 0.846 and 0.963, respectively.

Estimated error bound versus actual error bound under different memory usage (Fig. 12(a)–(d)): We find that the estimated error bound can match the actual error bound under different memory usage on all the four datasets, and the larger flow suffers from larger estimated error. We set the error to 25 in the experiments. On CAIDA dataset, for flows of size 1000, when using 500 KB memory, the probability that its estimated error is less than 25 is 0.900; when using 1000 KB memory, the probability is 0.979.

Summary: In summary, experimental results on the four datasets demonstrate that the error bound estimation method proposed in Section V-B can precisely estimate the actual error bound for error of different sizes under different memory usage.



(a) Bandwidth overhead.



(b) Packet number.

Fig. 13. Bandwidth and packet number overhead.

IX. 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.

References

[1] K. Yang et al., "SketchINT: Empowering INT with towersketch for per-flow per-switch measurement," in *Proc. IEEE 29th Int. Conf. Netw. Protoc.*, 2021, pp. 1–12.

[2] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proc. 7th Conf. Emerg. Netw. Experiments Technol.*, 2011, pp. 1–12.

[3] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving traffic demands for operational ip networks: Methodology and experience," *IEEE/ACM Trans. Netw.*, vol. 9, no. 3, pp. 265–279, Jun. 2001.

[4] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," in *Proc. Conf. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2003, pp. 325–336.

[5] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 245–256, 2004.

[6] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang, "Is sampled data sufficient for anomaly detection?," in *Proc. 6th ACM SIGCOMM Conf. Internet Meas.*, 2006, pp. 165–176.

[7] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, 2014, pp. 71–85.

[8] Y. Zhou et al., "Flow event telemetry on programmable data plane," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2020, pp. 76–89.

[9] 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.

[10] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *Proc. 12th USENIX Symp. Networked Syst. Des. Implementation*, 2015, pp. 455–468.

[11] Z. Li et al., "Rate-aware flow scheduling for commodity data center networks," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.

[12] Y. Li et al., "HPCC: High precision congestion control," in *Proc. ACM Special Int. Group Data Commun.*, New York, NY, USA, 2019, pp. 44–58.

[13] Y. Zhao et al., "Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets," in *Proc. 18th USENIX Symp. Networked Syst. Des. Implementation*, 2021, pp. 991–1010.

[14] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2015, pp. 662–680.

[15] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: Using packets for low latency network programming and visibility," in *Proc. ACM Conf. SIGCOMM*, New York, NY, USA, 2014, pp. 3–14.

[16] Cisco Nexus 9000 Series NX-OS Programmability Guide, 2019. [Online]. Available: 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

[17] R. B. Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "PINT: Probabilistic in-band network telemetry," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, pp. 662–680, 2020.

[18] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

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

[20] T. Yang et al., "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2018, pp. 561–575.

[21] Z. Liu et al., "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proc. ACM Special Int. Group Data Commun.*, New York, NY, USA, 2019, pp. 334–350.

[22] Q. Huang et al., "Toward nearly-zero-error sketching via compressive sensing," in *Proc. 18th USENIX Symp. Networked Syst. Des. Implementation*, 2021, pp. 1027–1044.

[23] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better netFlow for data centers," in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, 2016, pp. 311–324.

[24] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1449–1463.

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

[26] Y. Zhang, M. Roughan, W. Willinger, and L. Qiu, "Spatio-temporal compressive sensing and internet traffic matrices," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2009, pp. 267–278.

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

[28] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: A sketch framework for frequency estimation of data streams," in *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1442–1453, 2017.

[29] P4–16 Language Specification, 2020. [Online]. Available: https://p4.org/p4-spec/docs/P4--16-v1.2.1.html#sec-checksums

[30] Barefoot tofino: World's fastest p4-programmable ethernet switch asics, 2016. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

[31] P. Chen, Y. Wu, T. Yang, J. Jiang, and Z. Liu, "Precise error estimation for sketch-based flow measurement," in *Proc. 21st ACM Internet Meas. Conf.*, 2021, pp. 113–121.

[32] Y. Hua, B. Xiao, B. Veeravalli, and D. Feng, "Locality-sensitive bloom filter for approximate membership query," *IEEE Trans. Comput.*, vol. 61, no. 6, pp. 817–830, Jun. 2012.

[33] G. Liu, S. Guo, B. Xiao, and Y. Yang, "SDN-based traffic matrix estimation in data center networks through large size flow identification," *IEEE Trans. Cloud Comput.*, vol. 10, no. 1, pp. 675–690, First Quarter 2019.

[34] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with switchpointer," in *Proc. 15th USENIX Symp. Networked Syst. Des. Implementation*, 2018, pp. 453–456.

[35] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot, "NetDiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data," in *Proc. ACM CoNEXT Conf.*, 2007, pp. 1–12.

[36] Y. Peng, J. Yang, C. Wu, C. Guo, C. Hu, and Z. Li, "deTector: A topology-aware monitoring system for data center networks," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 55–68.

[37] D. Ghita, H. Nguyen, M. Kurant, K. Argyraki, and P. Thiran, "Netscope: Practical network loss tomography," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.

[38] H. H. Song, L. Qiu, and Y. Zhang, "NetQuest: A flexible framework for large-scale network measurement," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, pp. 121–132, 2006.

[39] B. Arzani, S. Ciraci, B, T. Loo, A. Schuster, and G. Outhred, "Taking the blame game out of data centers operations with netpoirot," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 440–453.

[40] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive realtime datacenter fault detection and localization," in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, 2017, pp. 595–612.

[41] M. Yu, "Network telemetry: Towards a top-down approach," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 1, pp. 11–17, 2019.

[42] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 233–248.

[43] A. Khandelwal, R. Agarwal, and I. Stoica, "Confluo: Distributed monitoring and diagnosis stack for high-speed networks," in *Proc. 16th USENIX Symp. Networked Syst. Des. Implementation*, 2019, pp. 421–436.

[44] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever, "Stroboscope: Declarative network monitoring on a budget," in *Proc. 15th USENIX Symp. Networked Syst. Des. Implementation*, 2018, pp. 467–482.

[45] T. Buddhika, S. L. Pallickara, and S. Pallickara, "Pebbles: Leveraging sketches for processing voluminous, high velocity data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 8, pp. 2005–2020, Aug. 2021.

[46] Y. Xiang, W. Zhou, and M. Guo, "Flexible deterministic packet marking: An IP traceback system to find the real source of attacks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 4, pp. 567–580, Apr. 2009.

[47] A. C. Viana, M D. de Amorim, Y. Viniotis, S. Fdida, and J. F. de Rezende, "Twins: A dual addressing space representation for self-organizing networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 12, pp. 1468–1481, Dec. 2006.

[48] C. Baquero, P. S. Almeida, R. Menezes, and P. Jesus, "Extrema propagation: Fast distributed estimation of sums and network sizes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 4, pp. 668–675, Apr. 2012.

[49] B. Xiao and Y. Hua, "Using parallel bloom filters for multiattribute representation on network services," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 1, pp. 20–32, Jan. 2009.

[50] Y. Qiao, T. Li, and S. Chen, "Fast bloom filters and their generalization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 93–103, Jan. 2014.

[51] M. D. D. Moreira, R. P. Laufer, P. B. Velloso, and O. C. M. B. Duarte, "Capacity and robustness tradeoffs in bloom filters for distributed applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2219–2230, Dec. 2012.

[52] S. Sheng, Q. Huang, and P. P. C Lee, "DeltaINT: Toward general in-band network telemetry with extremely low bandwidth overhead," in *Proc. IEEE 29th Int. Conf. Netw. Protoc.*, 2021, pp. 1–11.

[53] F. Deng and D. Rafiei, "New estimation algorithms for streaming data: Count-min can do more," *Webdocs. Cs. Ualberta. Ca*, 2007.

[54] T. Li, S. Chen, and Y. Ling, "Per-flow traffic measurement through randomized counter sharing," *IEEE/ACM Trans. Netw.*, vol. 20, no. 5, Oct. 2012.

[55] L. Liu et al., "SF-sketch: A two-stage sketch for data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2263–2276, Oct. 2020.

[56] T. Yang, S. Gao, Z. Sun, Y. Wang, Y. Shen, and X. Li, "Diamond sketch: Accurate per-flow measurement for big streaming data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 12, pp. 2650–2662, Dec. 2019.

[57] Y. Zhou et al., "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 741–756.

[58] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "BeauCoup: Answering many network traffic queries, one memory update at a time," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2020, pp. 226–239.

[59] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao, "OmniMon: Re-architecting network telemetry with resource efficiency and full accuracy," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2020, pp. 404–421.

[60] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: Dynamic resource allocation for software-defined measurement," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 419–430, 2015.

[61] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "SCREAM: Sketch resource allocation for software-defined measurement," in *Proc. ACM CoNEXT Conf.*, 2015, pp. 1–13.

[62] Y. Li, R. Miao, C. Kim, and M. Yu, "LossRadar: Fast detection of lost packets in data center networks," in *Proc. ACM CoNEXT Conf.*, 2016, pp. 481–495.

[63] Q. Huang, P. P. C. Lee, and Y. Bao, "SketchLearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2018, pp. 576–590.

[64] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, 2013, pp. 29–42.

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

[66] V. Braverman and R. Ostrovsky, "Generalizing the layering method of Indyk and Woodruff: Recursive sketches for frequency-based vectors on streams," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Berlin, Germany: Springer, 2013, pp. 58–70.

[67] X. Jin et al., "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 121–136.

[68] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin, "NetLock: Fast, centralized lock management using programmable switches," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2020, pp. 126–138.

[69] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. Ports, "Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories," in *Proc. 14th USENIX Conf. Operating Syst. Des. Implementation*, 2020, pp. 387–406.

[70] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. Int. Conf. on Database Theory*, 2005, pp. 398–412.

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

[72] A. Lakhina, M. Crovella, and C. Diot, "Characterization of network-wide anomalies in traffic flows," in *Proc. 4th ACM SIGCOMM Conf. Internet Meas.*, 2004, pp. 201–206.

[73] D. Lu, P. Mausel, E. Brondizio, and E. Moran, "Change detection techniques," *Int. J. Remote Sens.*, vol. 25, no. 12, pp. 2365–2401, 2004.

[74] A. Kumar, M. Sung, J. J. Xu, and J. Wang, "Data streaming algorithms for efficient and accurate estimation of flow size distribution," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 32, pp. 177–188, 2004.

[75] T. K. Moon, "The expectation-maximization algorithm," *IEEE Signal Process. Mag.*, vol. 13, no. 6, pp. 47–60, Nov. 1996.

[76] Y. Gu, A. McCallum, and D. Towsley, "Detecting anomalies in network traffic using maximum entropy estimation," in *Proc. 5th ACM SIGCOMM Conf. Internet Meas.*, 2005, pp. 32–32.

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

[78] C. H. Song, P. G. Kannan, B. K. H. Low, and M. C. Chan, "FCM-sketch: Generic network measurements with data plane support," in *Proc. 16th Int. Conf. Emerg. Netw. EXperiments Technol.*, 2020, pp. 78–92.

[79] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: A sketch framework for frequency estimation of data streams," in *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1442–1453, 2017.

[80] "DPDK is the data plane development kit that consists of libraries to accelerate packet processing workloads running on a wide variety of CPU architectures," 2023. [Online]. Available: https://www.dpdk.org/

[81] Q. Huang et al., "SketchVisor: Robust network measurement for software packet processing," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2017, pp. 113–126.

[82] The CAIDA UCSD Anonymized Internet Traces 2018, 2017. [Online]. Available: https://www.caida.org/catalog/datasets/passive_dataset/

[83] T. Yang et al., "HeavyKeeper: An accurate algorithm for finding top-k elephant flows," *IEEE/ACM Trans. Netw.*, vol. 27, no. 5, pp. 1845–1858, Oct. 2019.

[84] Real-life transactional dataset, 2004. [Online]. Available: http://fimi.ua.ac.be/data/

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

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

[87] The source code of Bob Hash, 1997. [Online]. Available: http://burtleburtle.net/bob/hash/evahash.html

[88] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ECN in multi-service multi-queue data centers," in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, 2016, pp. 537–549.

[89] M. Alizadeh et al., "Data center TCP (DCTCP)," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2010, pp. 63–74.

Kaicheng Yang received the BS degree in computer science from Peking University, in 2021. He is currently working toward the PhD degree with the School of Computer Science, Peking University, advised by Tong Yang. His research interests include network measurement, and programmable data plane.

Sheng Long received the BS degree in computer science from Peking University, in 2022. His research interests include network measurements, sketches, and KV stores.

Qilong Shi is currently working toward the senior graduate degree with the Department of Computer Science and Technology, School of Electronics Engineering and Computer Science, Peking University. He is currently conducting research under the guidance of Prof. Tong Yang, at the Institute of Network Computing and Information Systems, Peking University. His research interests include sketch, data stream processing, and network measurement.

Yuanpeng Li received the BS degree in data science from Peking University, in 2022. He is currently working toward the PhD degree with the School of Computer Science, Peking University, advised by Tong Yang. His research interests include network measurements, sketches, and programmable data plane.

Zirui Liu received the BS degree in computer science from Peking University, in 2021. He is currently working toward the PhD degree with the School of Computer Science of Peking University, advised by Prof. Bin Cui and Prof. Tong Yang. His research interests include algorithms in data streams and network measurements.

Yuhan Wu received the bachelor's degree from the Department of Electrical Engineering and Computer Science, Peking University, in 2021. He is currently working toward the PhD degree in computer science with the School of Computer Science, Peking University, advised by Tong Yang. His research interests include the fields of computer network and database, including key-value stores, network measurement, and sketches.

Tong Yang received the PhD degree in computer science from Tsinghua University, in 2013. He visited Institute of Computing Technology, Chinese Academy of Sciences (CAS), China from 2013.7 to 2014.7. Now he is an Associate Professor with the Computer Science Department, Peking University. His research interests include network Big Data, sketches, network measurement, Bloom filters, IP lookups, KV stores, hash tables, *etc.* He published papers in SIGCOMM, SIGMOD, SIGKDD, SIGCOMM CCR, VLDB, ATC, WWW, *IEEE Transactions on Parallel and Distributed Systems*, ToN, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Transactions on Computers*, ICDE, INFOCOM, *etc.*

Zhengyi Jia received the BS degree in electronic information science and technology from the University of Science and Technology of China, in 2014, and the PhD degree in signal and information processing from the National Network New Media Engineering Technology Research Center, Institute of Acoustics, Chinese Academy of Sciences in 2019. His research interests include programmable data plane, network telemetry, and high-performance network.