Rethinking Fine-Grained Measurement From Software-Defined Perspective: A Survey

Hao Zheng, Yanan Jiang, Chen Tian, Long Cheng, Qun Huang, Weichao Li, *Member, IEEE*, Yi Wang, Qianyi Huang, *Member, IEEE*, Jiaqi Zheng, Rui Xia, Yi Wang, Wanchun Dou, and Guihai Chen

Abstract—Network measurement provides operators an efficient tool for many network management tasks such as performance diagnosis, traffic engineering and intrusion prevention. However, with the rapid and continuous growth of traffic speed, it needs more computing and memory resources to monitor traffic in per-flow or per-packet granularity. Sample-based measurement systems (e.g., NetFlow, sFlow) have been developed to perform coarse-grained measurement, but they may miss part of records, especially for mice flows, which are important for some network management tasks (e.g., anomaly detection, performance diagnosis). To address these issues, data streaming algorithms such as hash tables and sketches have been introduced to balance the trade-off among accuracy, speed, and memory usage. In this article, we present a systematic survey of various data structures, algorithms and systems which have been proposed in recent years to perform fine-grained measurement for high-speed networks. We organize these methods and systems from a software-defined perspective. In particular, we abstract fine-grained network measurement into three-layer architecture. We introduce the responsibility of each layer and categorize existing state-of-the-art works into this architecture. Finally, we conclude the article and discuss the future directions of fine-grained network measurement.

Index Terms—Network measurement, sketch

1 Introduction

With the explosion of network usage and traffic volume, congestion and anomalies become more likely to happen because of sub-optimal network configurations and malicious attacks, which then will compromise network stability and security, and lead to economic loss. To guarantee the network performance, network measurement provides an avenue for efficiently collecting probing messages from the data plane, which paves the way to detect bottlenecks and anomalous behaviors. According to a recent study [1], the market revenue of network traffic monitoring and analyzing is expected to achieve 2.32 billion USD by 2020, at a Compound Annual Growth Rate of 24.7 percent. Lately, mainstream cloud providers, such as Azure [2] and Google Cloud [3], have developed more efficient and fine-grained tools for network measurement, which improve the transparency into networks and lower the barrier in performance analysis and network optimization.

Network measurement has been investigated since 1980s. Network administrators define the monitoring scope and granularity according to specific measurement tasks, and some measurement tasks (e.g., anomaly detection, performance diagnosis) require fine-grained measurement of network traffic. However, the hardware limitation of computing devices and switches brings great challenges to handle all the traffic packets. Therefore, network administrators have to sample these packets and only store the information of this sampled part. These sample-based measurement methods have been widely used due to their low storage and computational overhead, such as NetFlow [4], sFlow [5], sticky sampling [6], sample and hold [7], OpenWatch [8] and [9], [10]. These algorithms drop too much traffic information, and thus the accuracy of measurement results can not be guaranteed. As a result, it is hard to detect instantaneous anomalies and to count flow distribution.

In modern data center environment, a large volume of network traffic needs well-formed management [11], [12], [13], which fundamentally depends on high-resolution and fine-grained network measurement methods. Works that fully utilize these resources to achieve per-packet monitoring have been proposed [14], [15], [16], [17], [18]. These methods leverage well-designed data structures or advanced hardware features to resolve a set of queries related to network states and flow conditions, and then capture a complete picture which network administrators are interested in.

- *Hao Zheng, Yanan Jiang, Chen Tian, Jiaqi Zheng, Rui Xia, Wanchun Dou, and Guihai Chen are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210093, China.*
  E-mail: hzheng.cs@outlook.com, jiangyanan52239088@qq.com, {tianchen, jzheng, douwc, gchen}@nju.edu.cn, xiarui0428@163.com.
- *Long Cheng is with the Computer Science Department at Clemson University, Clemson, SC 29634 USA. E-mail: lcheng2@clemson.edu.*
- *Qun Huang is with the Department of Computer Science and Technology, Peking University, Beijing 100871, China. E-mail: huangqun@pku.edu.cn.*
- *Weichao Li, Yi Wang, and Qianyi Huang are with the Institute of Future Networks, Southern University of Science and Technology, Shenzhen, Guangdong 518055, China.*
  E-mail: liwc@sustc.edu.cn, wy@ieee.org, huangqy@sustech.edu.cn.
- *Yi Wang is with the School of Modern Posts, Nanjing University of Posts and Telecommunications, Nanjing, Jiangsu 210049, China.*
  E-mail: yiwang@njupt.edu.cn.

Manuscript received 27 October 2020; revised 5 July 2021; accepted 8 August 2021. Date of publication 11 August 2021; date of current version 9 December 2022.
(Corresponding author: Chen Tian.)
Digital Object Identifier no. 10.1109/TSC.2021.3103968

Recently, researchers seek to boost the performance of network measurement through dedicated network devices with specific requirements and improved algorithms (e.g. [15], [17], [19]). However, vendor-specific ASICs can not support customized packet processing pipelines, which hinders the deployment of advanced data structures designated for fine-grained network measurement. The advent of programmable packet processor (P4 [20]) and virtual switches (e.g., Open vSwitch [21], BESS [22]) brings more options to the design of new measurement methods. For example, P4 language together with a recent programmable Ethernet switch [23] can achieve such flexibility at a speed of 12.8 Tbps. In addition, virtual/software switches are widely deployed in VM and container environments to support packet forwarding for VMs or containers. Due to the flexibility of software, more complicated procedures can be implemented in network measurement, and seamlessly merged into virtual switches. Both P4 and virtual switches provision extra computing power to switches, and launch concerted efforts towards fine-grained network measurement.

Many novel methods and systems have been proposed in research community to conduct network measurement for high-speed networks. We broadly divide these methods and systems into two categories according to their measurement granularity:

- *Coarse-grained measurement methods*. Some measurement methods like NetFlow [4] and sFlow [5] adopt sampling strategy to adapt to the traffic speed. They record one packet from every N packets, which can be customized by network administrators. The sampled packets are collected by remote collectors, and further analyzed by upper-layer applications [24], [25], [26]. When a traffic burst happens, switches handle the increased traffic scale by reducing the sampling rate [27], in the meantime, with higher possibility missing many mice flows. Although many works have been explored to infer original traffic from the sampled parts [28], [29], [30], the accuracy is still limited by the low sampling rate required to make the measurement operation affordable [31]. We define this kind of methods as coarse-grained methods since they drop too much traffic information, especially for mice flows, which is important for many measurement tasks.
- *Fine-grained measurement methods*. In contrast to the sample-based methods, some data streaming algorithms perform a single pass over all the packets and then produce an estimate of the traffic characteristics. For example, the sketch-based algorithms (e.g., Count Sketch [32], CU Sketch [7], Count-Min Sketch [14]) summarize traffic statistics of all observed packets with sketches, which is a form of lossy compression on the original traffic data. Although the summarization induces a loss of resolution for analysis (due to hash collision), the degree of error is theoretically related to the amount of available resources. Besides sketch-based methods, many other methods can be used in fine-grained network measurement too (e.g., Frequent [33], Lossy Counting [34]). We define these methods as fine-grained measurement methods since they count and summarize network traffic without missing any flow packets.

In this survey, we explicitly focus on the fine-grained network measurement methods, and thus the coarse-grained methods are out of our scope since they miss too much information while facing heavy traffic scenarios. Note that some algorithms (e.g., Count-Min Sketch [14]) are not unique for network measurement. However, they act as an essential block in current fine-grained measurement systems [35], [36].

We hold the view that an ideal network measurement system need to be general, flexible and user-friendly. Although existing fine-grained methods meet the efficiency and accuracy requirements of network measurement, there are still some challenges. The first challenge is that most methods just support specific measurement targets. For example, Frequent [33] can just find all majority items without counting accurate frequency. Sketch-based algorithms like Count-Min Sketch [14] can not get flow cardinality [37]. More importantly, most measurement methods do not take usability into account. For example, sketch-based algorithms demand intensive manual efforts to configure the parameters in real deployment [38]. Furthermore, some measurement targets (e.g., heavy hitter [6]) are threshold-based. However, existing heavy hitter detection methods take the threshold as input and measurement accuracy is tightly coupled with the threshold choice [38]. In this survey, we abstract fine-grained network measurement into software-defined architecture. In particular, we divide a measurement system into three layers: an operation layer, a control layer and an application layer. We discuss the responsibility of each layer and categorize existing fine-grained measurement methods and systems into this architecture. Data streaming methods like sketches belong to the operation layer, and the control layer orchestrates existing measurement data structures and algorithms to conduct specific measurement tasks. In the application layer, network administrators can define measurement applications by exploiting user-friendly APIs without considering which concrete methods to use.

Note that we do not give a comprehensive design or implementation of the software-defined measurement architecture in this paper. Our contributions are that we give a systematic survey of various methods and systems in fine-grained measurement, and organize these methods and systems from a software-defined perspective. We hope our survey and the perspective can offer an all-around understanding of this area and motivate more novel ideas in fine-grained network measurement.

Most of the related surveys in the literature focus on data analysis. Alconzo *et al.* [39] focus on reviewing the big data analytics of high-volume measurement data. Survey [40] also focuses on big data analytic, but it mainly discusses network intrusion detection. Our survey is also different from [41] which introduces integrated network management tools where administrators can directly operate on a GUI to check traffic information. To the best of our knowledge, this is the first survey for the fine-grained measurement of high-speed networks.

We begin with the background knowledge of fine-grained measurement in Section 2, which includes several terms, some fundamental measurement targets, three basic algorithms and our principles for relevant work selection. We then introduce the three-layer architecture at the beginning of Section 3 and categorize existing fine-grained measurement methods and systems into the operation layer (see Section 3.1), the control layer (see Section 3.2) and the application layer (see Section 3.3) respectively. Finally, we conclude the paper and discuss the future directions of fine-grained network measurement in Section 4.

2 Background

In this section, we will briefly introduce the background knowledge of fine-grained measurement. We first give the definitions of some fundamental terms and measurement targets, which are widely used in this paper. Then we introduce three basic algorithms, which have a profound impact on current network measurement systems.

During network measurement, an *epoch* is defined as a time period of collecting. Considering the limitation of the storage capability, switches usually require recording information of traffic for a short period of time, and then send the information to a server, which allows switches to clear old records to make room for next small time slot. This small time slot is called an epoch, or a measurement epoch. What's more, we call a flow is an *elephant flow* or a *heavy flow* when the flow's volume significantly larger than most of the others, with at least C packets. Similarly, a *mouse flow* is a flow with fewer than C packets. The constant C is a varying parameter according to different traffic characteristics [42].

Measurement targets are specific metrics that we want to figure out in a measurement task. Here, we list several most fundamental measurement targets.

Heavy hitter [6]: A flow whose total size exceeds a specific threshold in a measurement epoch. Heavy hitters detection can help administrators to understand which flows occupy the most network bandwidth in an epoch.

Top-k [43]: The k flows with the largest size in a measurement epoch. It can be used to analyze which web pages are most popular or which flows occupy most bandwidth in an epoch.

Hierarchical heavy hitter (HHH) [44]: The flow which is defined by the longest prefixes (such as the prefixes of source IP) exceeds a specific threshold size after excluding any hierarchical heavy hitter descendants. For example, we use 32 bits source IP to define the flow, then we can build a 32-level binary tree, in which the leaf nodes are real IP addresses and other nodes are prefixes like $192.168.0.X$. The size of this prefix flow is the sum of all the flows whose prefix is $192.168.0.X$. If all the descendant flows of $192.168.0.X$ are small but the size of $192.168.0.X$ exceeds a threshold, then prefix flow $192.168.0.X$ is a hierarchical heavy hitter. The prefix $192.168.X.X$ is a hierarchical heavy hitter if its size exceeds threshold after decreasing all the hierarchical heavy hitter descendants like $192.168.0.X$. The hierarchical heavy hitter can be considered as a heavy hitter in the perspective of clusters.

Heavy changer (HC) [45]: A flow whose difference in size between two consecutive epochs exceeds a specific threshold. A heavy changer can be considered as a signal of congestion or malicious attacks.

Superspreader (SS) [46]: A source host that sends data to more than a specific threshold number of destination hosts in a measure epoch. It can be a network scanner which detects possible vulnerabilities of hosts.

DDoS [47]: A destination host which is under DDoS attack and receives data from more than a specific threshold number of source hosts in a measurement epoch.

Cardinality [37]: The number of distinct flows in a measurement epoch. It can be used to calculate the accessing frequency of a service or a server.

Flow size [48]: The size of any flow which can be identified by a user-defined flow key. It is useful for network service providers to calculate the costs of traffic which are generated by users.

Flow size distribution [31]: The size distribution of measured flows in an epoch. This target aims at analyzing whole bandwidth occupancy instead of focusing on a single flow, which is helpful for network performance improvement.

All the above measurement targets need fine-grained measurement of the time-varying network traffic. However, the rapid and continuous growth of traffic speed brings great challenges to measure these targets. Fortunately, data streaming techniques have been introduced to perform an approximate measurement, which balance the trade-off among accuracy, speed, and memory usage. Here we introduce three representative algorithms, which have a profound impact on current network measurement systems.

Lossy Counting [6] aims to find all items that exceed a user defined frequency. Given a very small number ε to find all items whose frequency exceed εn, the algorithm divides an incoming stream into $buckets$ with width $w = \lceil\frac{1}{\varepsilon}\rceil$ conceptually, where the $bucket\ id$ varies from 1 to $\lceil\frac{n}{w}\rceil$. The algorithm maintains a data structure in the form of (e, f, Δ), where e is the input item in the stream, f is the estimated frequency, and Δ is the maximum error of f. When processing the current input item i with total processed packet number n', if (i, f, Δ) has already existed, we add f; otherwise we create a new tuple $(i, 1, current\_bucket\_id - 1)$. Periodically, the tuples with $f + \Delta \leq current\_bucket\_id$ will be eliminated because their frequency upper bounds are less than the $current\_bucket\_id = \lceil\frac{n'}{w}\rceil = \varepsilon n'$. In [34], each time a new $bucket$ begins to process, we make all tuples decreasing f by one, then we remove all the tuples whose estimated frequency $f = 0$. In this algorithm, the space utilization is $O(\frac{1}{\varepsilon}\log(\varepsilon n))$. Lossy Counting can find all the heavy hitters which exceed a proportion of the whole traffic in network measurement and provides a bounded estimation.

Bloom Filters [49] are used to check whether a packet belongs to an exited flow, or check whether a flow is a member of already known heavy hitters. The most classic bloom filter [49] was proposed in 1970, and then it has been widely used to dynamically test whether an item is already existing. In this bloom filter, a binary array B with initially all zeros and several hash functions $h_1, h_2, \ldots h_k$ are required. To add an item i as a member, the classic bloom filter needs to hash item i by these k hash functions to find k bits in array B and then set these bits to 1. To test whether an item is a member, it also uses the k hash functions to find the mapping bits and check them. If all the k bits are 1, the bloom filter considers it as an existing member, if any bit in these k bits is 0, the bloom filter determines that the item is

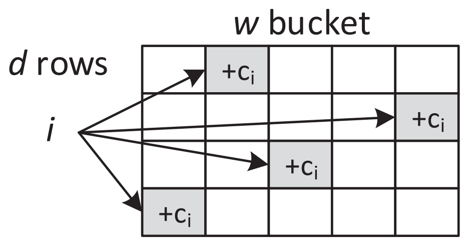


Fig. 1. Overview of CM Sketch [14].

not a member. This algorithm may cause false positives due to hash collisions. In practice, bloom filters are always used in collaboration with other traffic measurement algorithms to help improve the efficiency and accuracy of measurement tasks [50].

Count-Min Sketch (CM Sketch) [14] is one of the most popular sketch-based algorithms to find heavy hitters nowadays. It is an over-estimation algorithm but only requires one memory access for each counter. Count-Min Sketch is represented by a $d \times w$ array of counts with d hash functions $h_1, h_2, \ldots, h_d$ from $[n]$ to $[w]$. As shown in Fig. 1, for each input item i with quantity c_i, CM Sketch adds c_i to the counters $C[j, h_j(i)]$ for all rows j. The frequency estimation $\tilde{f}_i$ is also the minimum count among $C[j, h_j(i)]$. Setting $d = \lceil \ln \frac{1}{\delta} \rceil$ and $w = \lceil \frac{e}{\varepsilon} \rceil$ can ensure that $\tilde{f}_i \leq f_i + \varepsilon n$ with probability at least $1 - \delta$. The update time of CM Sketch is $O(\log \frac{1}{\delta})$ and the space bound is $O(\frac{1}{\varepsilon} \log \frac{1}{\delta})$ [51]. Simple, efficient operation and small memory usage with a high accuracy guarantee make CM sketch becoming a popular traffic counting algorithm in network measurement.

Recall that we divide network measurement methods into two categories. The above three algorithms can be seen as the fine-grained measurement methods because they can check or record all the packets passing through. Although they are not unique in network measurement, they build an essential block so that a lot of fine-grained measurement methods and systems are based on them (e.g., [16], [17], [35], [36], [52], [53]). However, the function of Bloom Filter is limited, and simply using counters like Lossy Counting is not general enough because it still drops too much information about traffic. For example, Lossy Counting [6] cannot get flow cardinality [37] because it only maintains the flows whose frequency exceeds a specific threshold. In contrast to Lossy Counting, sketches are more general for different kinds of measurement targets since they record the volume of all monitored flows.

Principles for Relevant Work Selection. The research community has accumulated a lot of research work on network measurement. A selection is necessary due to the considerations on subject clarity and space constraints. Our principles for the selection of papers can be summarized as follows :

- We prioritize papers that are published in prestigious conferences and journals and highly relevant work cited in these papers.
- Only fine-grained measurement methods are discussed in this paper. Coarse-grained methods are excluded due to their inability to adapt to the rapidly evolving traffic rates.
- The scope of our discussion is not limited to measurement algorithms. Other research on fine-grained measurement systems such as measurement language [54], multitasking management [55], and measurement data collection [56] are also considered in this survey.
- For reasons of timeliness, we tend to select related papers on network measurement in the past five years.

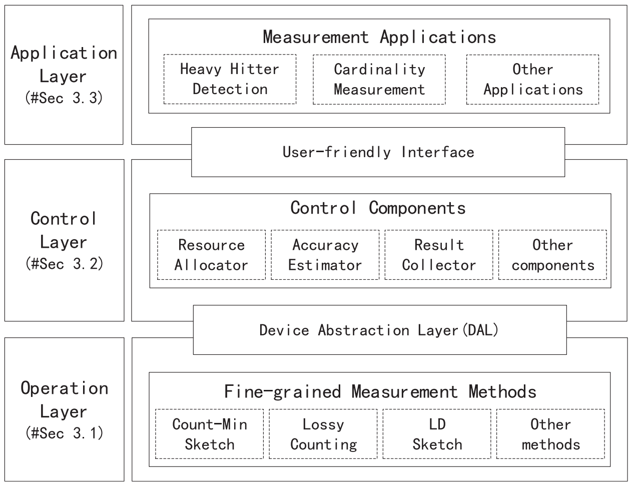


Fig. 2. Overview of software-defined fine-grained measurement architecture.

We present a comprehensive summary of the selected papers and discuss them from a software-defined perspective in the next section.

3 Survey Fine-Grained Measurement From Software-Defined Perspective

In this section, we will discuss some advanced data structures, algorithms and systems, which have been proposed recently to perform fine-grained measurement. First, we need to introduce the perspective of our survey. We hold the view that a mature measurement system need to be general, flexible and user-friendly. Referring to the idea of software-defined networking (SDN), we abstract a fine-grained measurement system into three-layer architecture, including an operation layer, a control layer and an application layer. As shown in Fig. 2, we organize this section by categorizing existing fine-grained measurement methods or systems into this three-layer architecture.

Operation Layer: In the operation layer, specific fine-grained measurement methods are installed in different network devices' data plane. They collect the records of each passing packet so that we can have a clear understanding of the current traffic characteristics by querying corresponding data structures. In Section 3.1, we perform a comprehensive discussion of recent advanced data structures and algorithms which have been proposed to perform fine-grained network measurement.

Control Layer: In the control layer, a measurement controller selects proper data structures and algorithms to meet the demands of different measurement applications. Then these applications' demands can be seen as many measurement tasks scheduled by the controller. In particular, the controller is responsible for installing concrete fine-grained measurement methods in hardware or software devices,

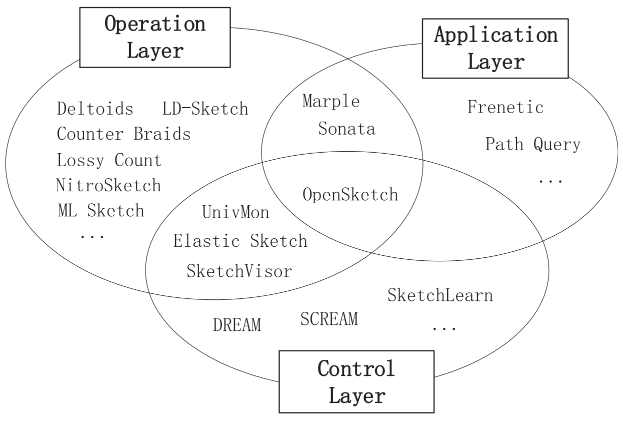


Fig. 3. Layer division of measurement systems.

and then balances the trade-off between resources occupancy and measurement accuracy. A mature measurement system also needs a Device Abstraction Layer (DAL) to transparently install measurement methods without considering which type of the device is used. What's more, measurement data are gathered in the control layer since most measurement systems use distributed measurement architecture (e.g., [48], [57]). In multitasking scenarios, the controller also needs to schedule and manage multiple measurement tasks. Therefore, there are several control components in the control layer (e.g., Resource Allocator, Result Collector). In Section 3.2, we discuss several fine-grained measurement systems, which propose novel mechanisms to manage measurement methods and schedule measurement tasks.

Application Layer: Traditionally, network administrators need to perform network measurement manually according to specific management tasks, which is time-consuming and error-prone. From the software-defined perspective, this process can be abstracted as a measurement application running on the application layer. Network administrators define several measurement applications by user-friendly interfaces to automatically perform measurement and collect results. In Section 3.3, we introduce several measurement interfaces, and we discuss the requirements of interfaces for fine-grained network measurement. Note that concrete measurement applications are not within the scope of our discussion.

For each layer, we will discuss several state-of-the-art works in detail. However, some measurement systems do not only focus on one single layer. We show this phenomena in Fig. 3. For example, OpenSketch [36] not only gives a novel design of data plane to implement different sketches, and then automatically manages the sketches with a sketch manager and a resource allocator, but also provides user-friendly APIs for administrators to define specific measurement tasks. That means OpenSketch covers all the three layers. In this survey, we use OpenSketch(C) to represent the control layer of OpenSketch, while OpenSketch(O) and OpenSketch(A) are defined as the operation layer and the application layer of OpenSketch respectively. Other multi-layer measurement systems are discussed in the same way. We carefully discuss the different layer's features of these multi-layer systems. Most of them can not be discussed separately because the layers are closely related to each other [53], [54], [58]. In the next three subsections, we will discuss recent fine-grained measurement methods and systems in detail from bottom to top.

3.1 The Operation Layer

In the operation layer, concrete data structures and algorithms are running in different devices to perform various measurement tasks. Among them, sketch-based measurement methods are widely studied for their versatility, high performance and ease of deployment. To carry out in-depth discussions, we broadly divide fine-grained measurement methods into two categories: sketch-based methods and other methods (i.e., non-sketch methods).

3.1.1 Sketch-Based Methods

Recall that we briefly introduce the Count-Min Sketch in Section 2, which is one of the most popular sketch algorithms to perform fine-grained measurement nowadays. A lot of measurement methods and systems are based on it (e.g., [15], [18], [48]). However, there are some limitations of Count-Min Sketch. We list four mainly limitations here:

- *Limitation I: Lack of flow key records*. In real production environment, there are numerous flows, which brings great challenge to record all the flow keys [15], [16], [52]. Count-Min sketch itself does not record any flow keys. Users need to provide candidate flow keys to check frequency, which limits the generality of sketches.
- *Limitation II: Accuracy loss caused by resource conflicts*. When tracking massive network traffic with limited resources, frequent hash collisions between elephant flows and mice flows cause serious accuracy loss [38].
- *Limitation III: Fixed counter size*. All the counters in Count-Min Sketch are fixed length with fixed counting capacity. While a counter is used to count heavy hitters, it may overflow. But if a counter is used to count mice flows, then the high bits of this counter will be wasted [59], [60].
- *Limitation IV: Significant computation overhead*. Sketches need to perform hash computation many times for each packet, which introduces significant computation overhead especially for software switches [53], [61], [62].

Here, we discuss several representative measurement algorithms or systems in detail, each of which addresses one or more of the above limitations. Because of the page limitation, other methods which use similar techniques have to be discussed briefly.

Deltoids [15] is designed to find heavy changers between different epochs or switches, and can recover flow keys of heavy changers, which solves the Limitation I indirectly. Deltoids is based on Group Testing technology which is divided into two parts: identification and verification. The identification finds candidate flows which may be heavy changers. The verification removes the items from the candidate set which are not true heavy changers. The workflow of Deltoids is shown in Fig. 4. Identification structure is an extended CM Sketch which requires $a \times g$ buckets and a hash functions $h_1, h_2, \ldots, h_a$ which map $[n]$ to $[g]$. Each

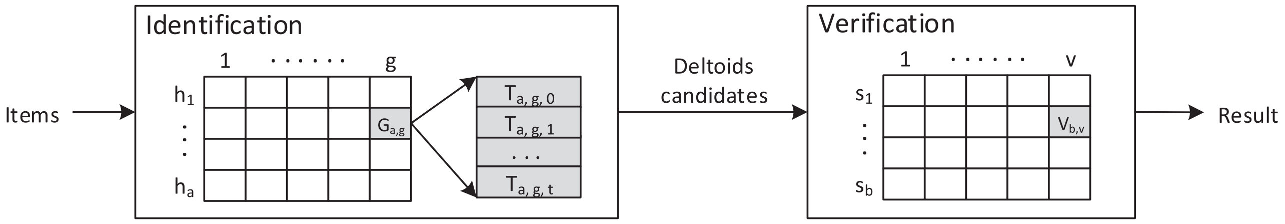


Fig. 4. The finding procedure of Deltoids [15].

bucket represents an item group $G_{a,g}$ which contains all the items mapped in it. In practice, each bucket keeps $1 + \log n$ counter structures $T_{a,g,t}$ which count item i in $G_{a,g}$ whose binary presentation in the tth position is 1. Especially, $T_{a,g,0}$ counts all the items in $G_{a,g}$. Verification structure needs $b \times v$ buckets and b hash functions $s_1, s_2, \ldots, s_b$ which map $[n]$ to $[v]$, each bucket also represents a group $V_{b,v}$ with a single counter. In the update procedure, for each mapped bucket (a, g) of item i, if the binary presentation of i in the tth position is 1, Deltoids updates all the counter $T_{a,g,t}$ with its quantity. Then, Deltoids updates the quantity of i directly to the sketch of verification. The heavy changers are estimated by the T structures and verified with V. The space required to find absolute and relative deltoids is $O(\frac{1}{\varepsilon}\log(n)\log\frac{1}{\delta})$ and the update time is $O(\log(n)\log\frac{1}{\delta})$. Deltoids addresses Limitation I by recovering flow-key candidates from identification part. The sketch in the identification extends each bucket of CM Sketch from one single counter to multiple, which requires more memory overhead. In addition, the counter number in each bucket is related to packet total number n, which means users need to know the scale of flow keys to monitor in advance.

Augmented Sketch (ASketch) [17] is a stream processing framework which uses a separating counting structure to improve the estimation accuracy. It solves the Limitation II that mice flows may collide with elephant flows and cause false positives in Count-Min sketches. ASketch contains two data structures: filter and sketch. As shown in Fig. 5, the filter stores frequent items with a new_count and an old_count. For each item i with value c_i in the incoming stream, ASketch looks for i in the filter at first. If the item i already exists in the filter, then it only adds the new_count. If i does not exist in the filter and filter is not full, it inserts i with its old_count setting as 0 and new_count as c_i. While the filter is full, then the item i will be sent to the sketch and do normal update procedure. After inserting the item to the sketch, ASketch compares the estimated frequency $\tilde{f}$ of current item cur with the smallest new_count of the filter. If $\tilde{f}$ is larger than the smallest new_count, ASketch exchanges the items. The cur item moves to the filter with both old_count and new_count setting as $\tilde{f}$. The removed item $remove$ which has the smallest new_count is inserted to the sketch with update value (new_-count - old_count) since the old_count value is already contained in the sketch since the item was last removed from the sketch. Notice that the exchange is triggered only once between filter and sketch. Which means, after sketch insertion, if the estimated frequency of $remove$ is larger than the new smallest new_count of the filter, it won't trigger one more item exchange. ASketch uses tuples to record potential heavy hitters, all tuples are in the filter. Under real-world datasets, ASketch can achieve $30\% - 40\%$ more data stream processing throughput than single sketches and improve estimation accuracy by 20 percent due to the skew characteristic of network traffic: 20 percent elephant flows occupy 80 percent whole traffic. Mainly heavy flows only access the filter and then the update is done, only 20 percent traffic will access the second sketch. This separation decreases the hash collision between heavy flows and mice flows, and the filter records flow keys of heavy flows as well, which means ASketch can also address Limitation I.

Counter Braids [59] is a structure using layered counters to solve Limitation III: all the counters in the sketch are fixed length with fixed counting capacity. The main goal of this structure is to save counters bits for mice flows and use more layered counters to record elephant flows. Counter Braids uses a low-complexity algorithm based on a graph to reconstruct flow sizes at the end of a measurement epoch with essentially zero error. Counter Braids has a layered structure in which the lth layer has m_l counters with the depth of d_l bits, as shown in Fig. 6. Let L be the total number of layers. In each layer, the number of counters can be decreasing, because a higher layer means larger flow size and the number of large flow is limited in networks. Counters in the low layer contain an additional status bit with initial value 0, which is set to 1 after the counter overflows.

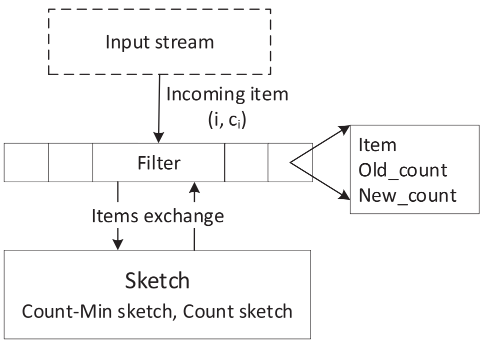


Fig. 5. ASketch architecture [17].

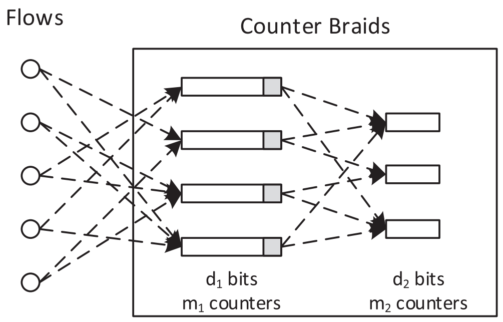


Fig. 6. The structure of Counter Braids [59].

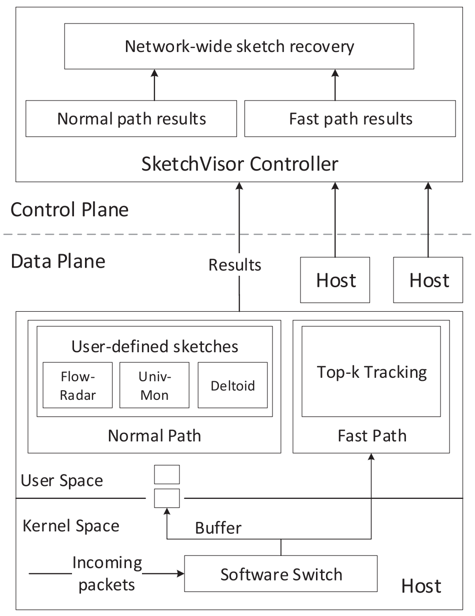


Fig. 7. SketchVisor architecture [53].

Counter Braids uses only one set of hash functions to map flows to the first-layer counters and then map between the two consecutive layers of counters. For each input item i, all the first-layer counters it hashed into are incremented. If a counter c overflows, all the counters in the next layer that c hashed into are incremented, and the value of counter c is reset to 0 with setting its status bit to 1. The reconstruction algorithm decodes these counters from the highest layer to the lowest and calculates the size of each flow at the end of each measurement epoch. While decoding, messages (calculated counter values) pass through the mapping relationships of consecutive layers iteratively and then the flow sizes can be calculated [59]. Notice that the decoding procedure is only executed at the end of each epoch, and it decodes all the flow sizes not only the heavy hitters. Thus, Counter Braids is not an instantaneous function, but it can recover all flow sizes it has recorded which is useful to measure flow sizes and flow size distribution in real network measurement.

SketchVisor(O) is the operation layer of SketchVisor [53], which is a robust network measurement framework for high-speed software packet processing. As shown in Fig. 7, SketchVisor(O) is divided into two components: a normal path and a fast path. The normal path is deployed with one or multiple sketch-based solutions, and processes packets from a bounded FIFO buffer. When the traffic load exceeds the processing capacity of the normal path and the FIFO buffer becomes full, SketchVisor(O) redirects overflowed packets to the fast path where a new top-k algorithm with low computational overhead is deployed. In the fast path, each flow f is associated with three counters (e_f, r_f, d_f) in a hash table. e_f counts the maximum possible bytes that can be missed before flow f is inserted. r_f counts the residual bytes and d_f counts the decremented bytes after f is inserted. The fast path maintains a variable E to count the sum of all decremented bytes. Same with normal counter-based measurement algorithms, a first appeared flow f with size v will be inserted with $(E, v, 0)$ to the hash table when the table is not full. If f is already in it, SketchVisor (O) only increases the corresponding counter r_f. Otherwise, the algorithm will calculate a decremented value $\widehat{e}$, each flow counter r_f in the table decreases by $\widehat{e}$ and each d_f increases accordingly. After counter decreasing, the flows whose r_f counter is no larger than 0 will be kicked out from the hash table. Flow f will be added with $(E, v-\widehat{e}, \widehat{e})$ if $v-\widehat{e}$ is larger than 0. The fast path drops some information during packet processing so that it is less accurate than the normal path. To achieve a more accurate measurement, the normal path should process packets as many as possible, and the fast path is activated only when it is necessary. SketchVisor(O) can address Limitation IV by using the double path pattern to deal with different traffic speed (i.e., up to 10Gbps [53]). Also, the double path pattern leads to a question that a part of packets will be processed completely but the others will not. Then we think SketchVisor(O) will cause information loss which means not all the packets execute the same counting procedure. Fortunately, the control layer of SketchVisor alleviates this problem by adopting a recovery algorithm that uses matrix interpolation and compressive sensing to eliminate the extra errors due to the fast path processing.

NitroSketch [61] is an outer framework for existing sketch-based algorithms, which aims to reduce the computational overhead of measurement algorithms in software switches (Limitation IV). NitroSketch is based on the key property that software switches have more storage resources, and then the computation resources become bottlenecks, which is opposite to hardware switches. The key bottlenecks of sketches in the software switch include several expensive hash computations for each packet, multiple random memory accessing and updating for per-packet processing, and additional overheads to track top-k flow keys. NitroSketch has three key ideas. First, samples on the independent counter arrays of multi-array sketch structure, rather than sampling on packets. For each packet, NitroSketch "flip a coin" with a sample rate p for each array while processing to decide if the counter needs to be updated in this array, which can reduce the times of hashing and counter updating. Second, improves coin flips and adopts a geometric sample way to further reduce computation. When sampling the counter arrays, flipping coins row by row will introduce a large computational overhead. The second idea draws samples from a geometric distribution and can directly decide which counter array will be updated next and how many packets will be skipped until this update. Third, to achieve proper convergence time, the sampling rate is adaptive based on the packet arrival rate. Since more packets will be skipped, the long waiting time is caused to have guaranteed accuracy. Thus, with a low packet arrival rate, the sample rate p can be enlarged to record more packets. There are two modes of adaptive sampling: 1) AlwaysLineRate mode, which dynamically sets p to be inversely proportional to the current packet arrival rate; 2) AlwaysCorrect mode, which starts with $p = 1.0$ and switch to AlwaysLineRate mode once it can guarantee the convergence. We give the total view of the NitroSketch framework in Fig. 8. Once packets arrive at the software switch, NitroSketch selects the packet and the counter array to be updated. As a result, it skips the majority of packets. The sampled packets can update the selected counter array(s) based on the updating functions of

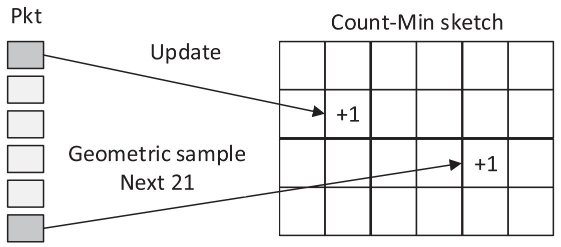


Fig. 8. Sketch with the NitroSketch framework [61].

different sketch algorithms (e.g., in Fig. 8, the first packet updates a counter in Array 2. Then, NitroSketch selects the next updated array which is 21 after the current array. Then, the 4 packets in the middle are skipped and the last packet will update the counter in Array 3). Before packet processing, administrators can choose a sampling mode according to their requirements. This framework is general and can apply to any existing sketch structure to boost packet processing performance without sacrificing their theoretical accuracy guarantee. Thus, here we think NitroSketch still provides fine-grained measurement even if it uses sampling. NitroSketch can be considered as a supplement of existing systems to reach a high line rate.

These sketch-based methods use several representative techniques to solve the above limitations. There are many other sketch-based methods which use similar techniques, but they have some novel points to solve other specific problems. For example, SketchLearn [38] not only addresses Limitation II and Limitation III in its data plane (SketchLearn(O)) by adopting similar techniques with the above methods, it also relieves the burdens of network administrators and supports extended queries on collected information. We briefly introduce other sketch-based methods here because of the page limitation. Count-Min-Heap [63] uses a heap to track all candidate heavy flows (i.e., can address Limitation I) and their estimated sums. MV-Sketch [64] is an invertible Sketch (i.e., can address Limitation I) that supports both heavy hitter and heavy changer detection and can be generalized for distributed detection. FlowRadar [65] proposes encoded flowsets, which is an extension of Invertible Bloom filter Lookup Table (IBLT). For each bucket, FlowRadar keeps the packet count, the number of flows, and the XOR of all different flow keys mapped in this bucket. Further, FlowRadar can get all flow keys by performing a decoding algorithm on the encoded flowsets. SpreadSketch [66] is also an invertible sketch, but it aims to perform network-wide superspreader detection. Reversible Sketch [16] addresses Limitation I by using reversible hash functions to recover the flow keys. SeqHash [67], which is similar to the reversible sketch, hashes the words of keys to the sketch and reverses heavy keys from heavy buckets. SeqSkech and EmbedSketch introduced in [68] can reach nearly zero errors by properly incorporating compressive sensing theory with sketch algorithms. Cold Filter [18] is a separating counting algorithm like Augmented Sketch [17] but only needs one-direction communication between the filter and the sketch. It also aims to improve the measurement accuracy (i.e., can address Limitation II). Pyramid Sketch [60] solves Limitation II and III by using pyramid-shaped data structure to automatically enlarge the size of the corresponding counters according to the frequency of current incoming items, which is similar to Count Braids [59] but is more efficient. Lossy Conservative Update (LCU) sketch [69] combines the idea of lossy counting (see Section 2) on top of CU sketch [7], which also divides the stream into windows and decrements the sketch counters at window boundaries to reduce the error of over-estimation. Elastic Sketch(O) [48] also separates elephant flows from traffic like Augmented Sketch and can be adaptive to bandwidth, packet rate and flow size distribution. LD-Sketch [52] is an arrayed sketch structure that combines the counter-based techniques to solve Limitation I that Count-Min Sketch does not record any flow keys. It is designed to accurately detect heavy hitters and heavy changers using distributed architecture. UnivMon(O) [57] uses universal streaming [70] where a single universal sketch is provable accurate for estimating a large class of functions. Defeat [71] combines sketches with the subspace method [72] to detect anomalies, and it can identify the IP flows(s) that are responsible for the anomaly. Fast sketch [73] aggregates packets into a few flows with higher efficiency and reliability, and can be used to find heavy changers. TCM [74] uses a set of graph sketches which are graphs with far less size than original graphs to summarize streams and monitor flows, then it can support more types of graph analytics (e.g., flow paths). Bias-Aware Sketches [75] proposes bias-aware linear sketches which strictly generalizes standard sketches in the error guarantees under biased data streams. ML Sketch [76] improves the estimation accuracy (Limitation II) by introducing machine learning theory into network measurement. Odd sketch [77], which is a compact binary sketch, estimates the Jaccard similarity of two sets.

Conclusion of sketch-based methods: As a lossy compression of all monitored flows, Sketch can support a wide range of measurement targets. So we can find that most of the sketch-based methods can get a wide range of measurement targets by leveraging the generality of Sketch. These sketch-based algorithms aim to make Sketch more general, more accurate and more efficient. We make a conclusion of these methods and their novel points in Table 1.

3.1.2 Other Methods

The sketch-based methods play an important role in fine-grained network measurement. Besides sketch-based methods, many non-sketch algorithms use different techniques to achieve the same goal.

The first class is the counter-based methods like Lossy Counting (see Section 2). Space Saving [43] aims to avoid the dynamic memory allocation of Lossy Counting (see Section 2) and add frequency estimation to Frequent [33]. HashPipe [78] is based on the SpaceSaving algorithm to track heavy hitter. Probabilistic Counting of Flajolet and Martin [79], Linear Counting [80], LOGLOG [81] and HyperLogLog [82] are used to measure flow cardinality, especially, HyperLogLog was extended as HyperLogLog sketch [83] in 2017. Lall *et al.* in [84] provides two streaming algorithms to estimate the entropy of network traffic. Zhang *et al.* in [85] provides high-resolution measurement to detect traffic bursts that last tens of microseconds. Mitzenmacher *et al.* in [86] and Ben-Basat *et al.* in [44] proposes methods to deal with the hierarchical heavy hitter problem. CSE [87] is designed to detect superspreaders and DDoS based on

TABLE 1
Summary of Sketch-Based Methods in the Operation Layer

Measurement methods	Addressed limitations	Novel points	Measurement targets
Deltoids [15]	I	Recover candidate flow keys	Heavy hitter, Heavy changer, Flow size
Augmented Sketch [17]	I, II	Record part of flow keys and improve estimation accuracy	Heavy hitter, Heavy changer, Flow size
FlowRadar [65]	I	Decode all flow keys from encoded flowsets to provide generality	General purpose
Counter Braids [59]	III	Provide layered counters and minimize memory usage in flow counting	Heavy hitter, Heavy changer, Flow size
SketchVisor(O) [53]	IV	Introduce fast path to accelerate processing speed	General purpose
NitroSketch [61]	IV	Geometric distribution sampling on sketches	General purpose
Count-Min-Heap [63]	I	Record part of flow keys	Heavy hitter, Heavy changer, Top-k, Flow size
MV-Sketch [64]	I	Return candidate flow keys	Heavy Flow Detection
SeqSkech and EmbedSketch [68]	I, II	Reach to nearly zero errors by introducing compress sensing	General purpose
SpreadSketch [66]	I	Return candidate flow keys	Superspreader
Reversible Sketch [16]	I	Recover candidate flow keys by reversible hashing	Heavy hitter, Heavy changer, Flow size
SeqHash [67]	I	Recover candidate flow keys by a novel sequential hashing scheme	Heavy hitter, Heavy changer, Flow size
Cold Filter [18]	I, II	Record part of flow keys and improve estimation accuracy	Heavy hitter, Heavy changer, Top-k, Flow size
Pyramid Sketch [60]	III	Use pyramid-shaped data structure and minimize memory usage	Heavy hitter, Heavy changer, Top-k, Flow size
LCU sketch [69]	II	Combine the idea of lossy counting to further reduce the over-estimation error incurred	Heavy hitter, Heavy changer, Flow size
Elastic Sketch(O) [48]	I, II	Adapt to bandwidth, packet rate, and flow size distribution	General purpose
LD-Sketch [52]	I	Record part of flow keys and use distributed architecture	Heavy hitter, Heavy changer, Top-k, Flow size
UnivMon(O) [57]	-	Leverage universal streaming to achieve general and accurate measurement	General purpose
Defeat [71]	-	Combine sketches with the subspace method	Anomalies detection
Fast sketch [73]	IV	Aggregate packets into a few flows	Heavy changer
TCM [74]	-	Use a set of graph sketches to summarize streams and monitor flows	Graph analytics
Bias-Aware Sketches [75]	II	Provide error guarantees under biased data streams	Heavy hitter, Heavy changer, Flow size
ML Sketch [76]	II	Introduce machine learning theory in sketches to improve accuracy	General purpose
Odd sketch [77]	-	Propose a compact binary sketch	Jaccard similarity

We list the addressed limitations and supported measurement targets of each measurement method. The 'General purpose' means the sketch is general for any given measurement targets.

virtual vectors. Fast hash table [88] improves the performance of packet processing.

Since the standard bloom filter (see Section 2) only supports membership queries, many works have been proposed to improve it. Spectral Bloom Filter (SBF) [89] extends standard to multiple-sets supporting that allows filtering the item whose multiplicities are smaller than a threshold. Invertible Bloom Lookup Tables (IBLT) [90] is designed for key-value pairs and supports lookups in $O(k)$ time where k is the number of hash functions. IBLT can give accurate frequencies of flow packets in traffic monitoring. Bloom Tree [91] is also designed for multiple-set membership testing which uses a complete search tree, which can provide higher accuracy in frequency estimation. Counting Quotient Filter (CQF) [50] provides a general-purpose approximate membership query which can handle skewed input efficiently with supporting merges and resizes, thus it is proper to deal with the skewed traffic flows to give more efficiently membership testing. Persistent Bloom Filter (PBF) [92] provides membership testing for the entire recording history, which is useful in flow tracing.

Some measurement methods use classic mathematical and statistical theories to achieve fine-grained measurement. BeauCoup [62] is a query-driven measurement system which focus on count-distinct tasks(e.g., Superspreader [46]). It is based on the coupon collector problem and can achieve the same accuracy as sketch-based algorithms but uses 4x fewer memory access. Zhang *et al.* in [30] develops a novel spatio-temporal compressive sensing framework which can handle the missing values of traffic matrices and helps to measure networks more accurately. [31] uses a lossy data structure: Multi-Resolution Array of Counters (MRAC) to estimate distribution.

Some measurement systems try to explore the possibilities of different entities (e.g., end hosts, switches, controllers) in networks to achieve finer measurement or address unresolved issues in network measurement. OmniMon [93] coordinates different entities in data center to achieve the targets of full accuracy and resource efficiency. LightGuardian [56] is a lightweight in-band telemetry system. When each packet passes by a switch, the switch will probabilistically embed one column of a sketch called a sketchlet into the packet header. Then end-hosts collect the packets and incrementally reconstruct the original sketch from sketchlets. cSamp [94] coordinates different routers to achieve network-wide flow monitoring. NetSeer [95] is a flow event telemetry system, which is based on programmable switches and aims to discover and record all performance-critical data plane events (e.g., packet drops, packet pause, congestion). Measurement model in [96] and iSTAMP[97] are based on OpenFlow [98] and TCAM to build a network measurement framework. NetSight [99] records packet histories for further traffic analysis. Pingmesh [100] is designed to measure and analyze network latency. The works proposed by Kandula *et al.* in [101] and SNAP [102] collect socket-level logs to analysis network. FlowCover [103], OpenSample [104] and OpenNetMon [105] provide flow statistics monitoring schemes in SDN. Planck [106] uses port mirroring to provide millisecond-scale network monitoring with switch sampling. EverFlow [107] uses "match and mirror" rules to shuffle and filter packets and then reduces overhead.

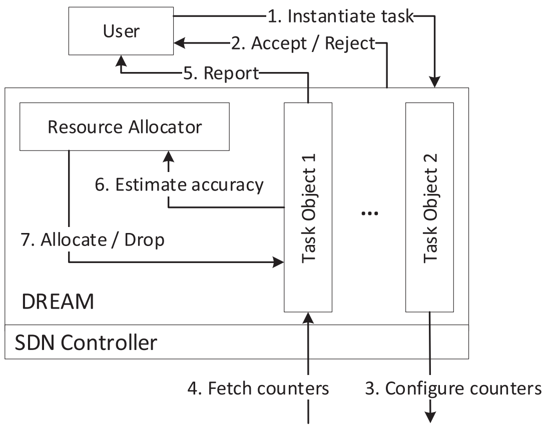


Fig. 9. DREAM system architecture [35].

Conclusion of non-sketch methods: These non-sketch methods leverage various theories and techniques to perform fine-grained measurement. Although these algorithms are not as general as sketch-based methods, most of them can achieve higher performance (i.e., more efficiency and more accurate) for a specific measurement target.

3.2 The Control Layer

With the help of existing novel measurement data structures and algorithms, we can perform fine-grained measurement with various targets in the operation layer. However, it's a hard work for network administrators to understand all features of these methods. Therefore, we need the control layer to automatically manage these algorithms and schedule multiple measurement tasks. In this subsection, we will discuss several measurement systems which focus on the management of measurement methods and measurement tasks in different aspects.

DREAM [35] targets at scheduling and allocating current available resources with different task requirements and different measurement epochs. It is a TCAM-based software-defined measurement system which is designed according to existing TCAM hardware on switches. Thus, it can be deployed at networks immediately. In DREAM, users can submit measurement tasks to the system. The submitted task should contain four parameters: 1) a flow filter, which specifying the traffic aggregate; 2) a packet header field, on which the task event is defined; 3) a threshold, specifying the threshold volume; and 4) an accuracy bound, which is specified by the user. For example, users submit a heavy hitter detection task with the flow filter $<10/8, 12/8, *, *, *>$, a threshold of 1M, the accuracy of 80 percent, and setting source IP as the packet header field. The details of the DREAM workflow are shown in Fig. 9. Step 1, the user instantiates a measurement task and specifies its four parameters. Step 2, DREAM decides to accept the task or not according to the available resources. Step 3, DREAM configures a default number of counters at one or more switches and creates a task object for the accepted task. Step 4, DREAM fetches counters from the switches and passes them to the task objects periodically. Step 5, task objects compute measurement results and report to the user based on the counters. Step 6, task objects measure the accuracy of the current task by the accuracy estimator and then send it to the resource allocator. Step 7, the resource allocator determines the number of TCAM counters to allocate to each task object. Task objects use the allocated counters to measure traffic and can reconfigure the switches (step 3). If a task is dropped due to the lack of resources, DREAM removes its task object and de-allocates the counters of this task. Dynamic resource allocation is the key component of DREAM. It computes both global accuracy and local accuracy to determine whether a task should be allocated with more resources. For example, if the global accuracy of a task exceeds the specified bound, there is no need to allocate more resources even if one of the local accuracy is below. But if the global accuracy dose not satisfy the bound, the switches with low local accuracy should allocate more counters for this task. While allocating resources, DREAM changes the number of counters by a factor of 2 due to the diminishing returns. If the task needs more counters, DREAM allocates it with a double number. Or if the task is rich and can free some counters, DREAM halves its counter. Also, operators can specify a drop priority for each task. Thus, poor tasks with low drop priority are permitted to steal resources from the tasks with high drop priority. As a result, tasks with high drop priority may be dropped. For large switches, DREAM can keep almost all tasks satisfying without rejecting or dropping. For small switches, it also can keep high satisfaction but with high rejection. The DREAM controller can handle many tasks for its high parallelization. Each task can run on a core and each allocator of one switch can run separately. The delay of the control loop includes saving the incremental rules, fetching counters, allocating resources, creating reports, estimating accuracy and configuring counters, which is less than 20ms with a 512 TCAM capacity switch.

OpenSketch [36] is a software-defined network measurement architecture with a simple, efficient data plane and a customized analysis controller. Leveraging flexible data plane design, OpenSketch(C) installs and manages different sketch algorithms with guaranteed accuracy, then collects and analyzes measurement data to meet the requirements of application layer. OpenSketch(C) builds a measurement library at the controller which can automatically configure the data plane with different sketches. Operators are free from understanding the complex implementation of switches, which makes the measurement programming much easier. The powerful functions of OpenSketch(C) are supported by the novel design of data plane. As it is shown in Fig. 10, the data plane contains three stages: 1) hashing, 2) classification, and 3) counting. The hashing stage chooses which set of packets should be measured, and then can reduce the volume of total measurement data. The classification stage can focus on specific flows that are specified by wildcard rules. The counting stage uses a table of counters to store and accumulate traffic statistics without flow keys recording to save memory. Thus, OpenSketch requires complex indexing while it uses hashing and classification modules. OpenSketch(C) can maintain the mappings between storage counters and specific flows by classification-based indexing, or reverse the flows from hash values by hash-based indexing. By leveraging smart combinations of the three stages, the data plane can support a wide variety of sketches to meet different measurement requirements. OpenSketch(C) provides a sketch library with a sketch

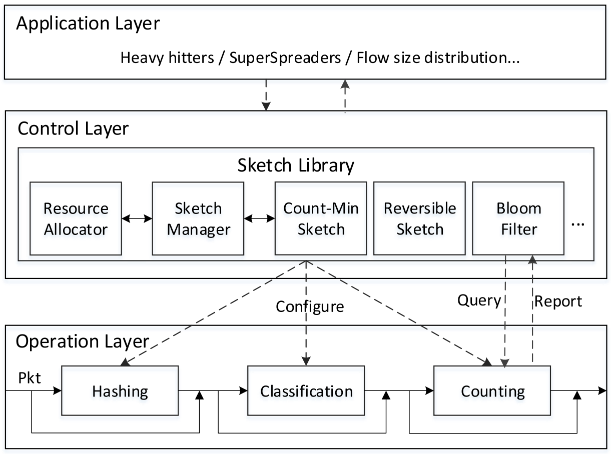


Fig. 10. OpenSketch architecture [36].

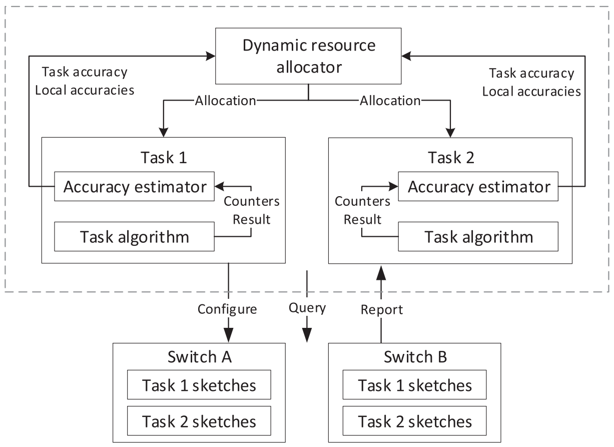


Fig. 11. SCREAM architecture [55].

manager and a resource allocator. According to the requirements of given measurement task, the sketch manager can automatically pick proper sketches, configure the sketches for the best accuracy and allocates resources across sketches with giving provable memory-accuracy trade-offs. The sketch manager can also automatically install new sketches in the data plane to learn traffic statistics for better configurations. The resources allocator can automatically allocate the limited resources across multiple measurement tasks without considering the implementation details of each task, it only cares about the relative importance of these tasks to the operator. With the resource allocator and the sketch manager, OpenSketch(C) can automatically perform the management of sketches such as sketches choosing, sketch size setting and so on. OpenSketch(C) has implemented seven sketches in C++ including bitmap, hash table, bloom filter, count-min sketch, reversible sketch and so on, and the data plane of OpenSketch has been implemented on NetFPGA. The experiments show that the OpenSketch prototype switch achieves full throughput without packet loss on 1GE port, the delay of each measurement component procedure is smaller than the packet incoming rate even for the 64B packet. However, the data plane of OpenSketch is a little complex and causes limitation of packet processing. Under up-to-date 10Gbps, 40Gbps, or even 100Gbps network links, OpenSketch can not handle the real traffic.

SCREAM [55] is a sketch-based resource allocation measurement system which can be regarded as an improved version of DREAM [35] and overcome some drawbacks: DREAM uses TCAM memory which is expensive and power-hungry. Thus, it can not provide numerous counters to support measurement tasks but relies on the solution of prefix-based summarization. SCREAM uses sketches to summarize traffic which can be easily implemented with cheap and power-efficient SRAM memory. Besides, SCREAM can capture the right set of flow properties without iterative reconfiguration. SCREAM implements sketch-based tasks across multiple switches, and also enables dynamic resource allocation with an accuracy estimator. It is inspired by the DREAM and deliberately reuses the dynamic resource allocator. The differences between SCREAM and DREAM are mainly in three points. 1) SCREAM can support sketch-based tasks that can not be supported by TCAM counters. 2) SCREAM can assign different sized sketches to different switches for they may see different amounts of traffic and then merge them. And 3) SCREAM improves the accuracy estimator of DREAM to dynamically allocate resources. SCREAM can run multiple concurrent instances of different task types. It distributes resources to each task on each switch. Each task queries counters from switches periodically. According to the traffic observations, tasks update the sketch parameters based on allocated resources and then reconfigure their counters. Thus, SCREAM contains two components: tasks and dynamic resource allocation. The total architecture of SCREAM is shown in Fig. 11. Tasks at the controller need to configure sketch counters at switches, fetch counters and prepare reports. For different tasks, SCREAM uses different algorithms (e.g., sum, min, max) to merge sketches with different sizes at multiple switches. Each task also contains an accuracy estimator which uses Markov inequality to calculate the precision and estimate its accuracy, and then shares its results and counters periodically. Dynamic resource allocation is the heart of the SCREAM system, it uses the instantaneous accuracy of tasks as the feedback for an iterative allocation algorithm. If the estimated accuracy is smaller than the specified accuracy bound, the poor task can receive more resources that are taken away from rich tasks whose estimated accuracy is much higher than the bound. The resource allocation order for poor tasks is based on assigned priorities. If a poor task can not be allocated, it may be dropped. While traffic skew keeps changing, SCREAM can support more accurate tasks than OpenSketch [36] for its dynamic resource allocation. SCREAM has a higher satisfaction rate with a lower rejection rate both in single switch and across multiple switches. The error of accuracy estimator varies with different task types, but it goes down with larger capacity switches. The experiments show that the error of SCREAM accuracy estimation is controlled within 5 percent, recall is above 80 percent for all switch sizes on average.

SketchLearn [38] aims at relieving the burdens of network administrators and supporting extended queries on collected information. It characterizes the inherent statistical properties of resource conflicts, and builds on a multi-level sketch which will infer and extract large flows iteratively to guarantee the remaining flows are small. The iterative inference is self-adaptive, and then the configured parameters will have little impact on the final results, in which way the binding between configurations and accuracy becomes loose. The architecture

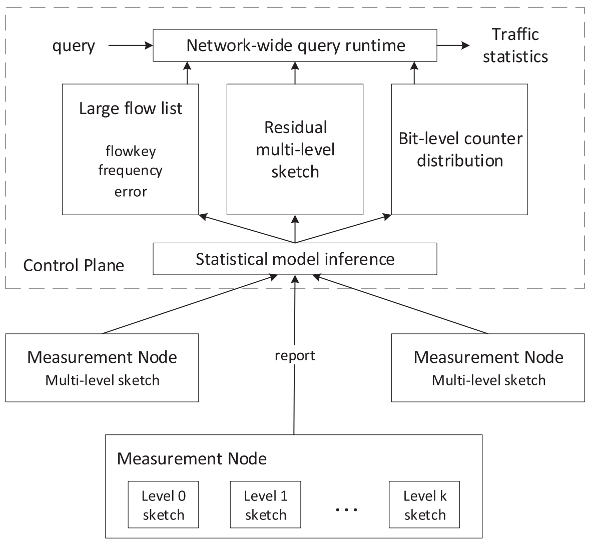


Fig. 12. SketchLearn architecture [38].

of SketchLearn (Fig. 12) contains a distributed data plane and a centralized control plane. The data plane contains multiple measurement nodes each deploying a multi-level sketch and processes the incoming packet. The control plane needs to analyze and decompose the multi-level sketch into three components: 1) the large flow list, which identifies large flows and records the estimated frequency and corresponding error; 2) the residual multi-level sketch, which stores the traffic statistics of the remaining small flows; and 3) the bit-level counter distributions, each of which models the counter value distribution of each level sketch in the residual multi-level sketch. SketchLearn uses a multi-level sketch structure that contains multiple small sketches. Each small sketch tracks the traffic statistics of a specific bit of the flow key (e.g., 5-tuple). Suppose l is the bit number of the flow key, then the multi-level sketch contains $l+1$ levels (level-0 to level-l), each of which corresponding to a small sketch with r rows and c columns counters. All the $l+1$ sketches use the same r hash functions where h_i hash function $(1 \leq i \leq r)$ maps a flow key to the jth columns $(1 \leq j \leq c)$ in the ith row. Notice level-0 sketch records all the packets, and other level-k $(1 \leq k \leq l)$ only records the packets whose kth bit of flow key is 1. Let $p[k]$ describes the probability that the kth bit equals to one and let $R_{i,j}[k] = \frac{C_{i,j}[k]}{C_{i,j}[0]}$. The multi-level sketch provides a key property that if there is no large flow, $R_{i,j}[k]$ follows a Gaussian Distribution with the mean $p[k]$. According to this key property, SketchLearn builds a conflict model which contains a large flow list and residual multi-level sketch. The model infers and extracts large flows by examining $R_{i,j}[k]$ and its difference from $p[k]$ iteratively, and then removes large flows from the sketches and adds to the large flow list until the observed $R_{i,j}[k]$ fits a Gaussian Distribution well. SketchLearn supports general traffic statistics and achieves high accuracy for various measurement tasks, and remains stable across different configurations. The network-wide coordination of SketchLearn also performs well. The experiment shows that the measurement accuracy significantly improves as the number of measurement points increasing. However, the multiple sketches make SketchLearn requiring more hashing and more counter updating in the data plane, which may cause performance degradation.

Besides the above systems, there are many other measurement systems which aims to manage the operation layer. UnivMon(C) [57] generates sketching manifests to specify the monitoring responsibility of each switch. When a packet arrives at a switch, the switch uses the manifest to determine the set of sketching actions to apply. UnivMon (C) collects the sketch information from switches, and runs estimation algorithms for every management application. Adaptive Sketches (Ada-Sketches) [108] provides time adaptive sketches that dynamically adjust sketch sizes per-interval and reallocate the limited memory resources as time passes. Elastic Sketch(C) [48] can globally collect measurement data in an efficient and adaptive way, and it is the first measurement system that proposes sketch compression algorithms and merging strategies. SketchVisor(C) [53] aims at merging the measurement results from both normal path and fast path. It designs a recovery algorithm that uses matrix interpolation and compressive sensing to eliminate the extra errors due to the fast path processing. gSketch [109] uses the network characteristic which calls *Local Similarity* and partitions a virtual global sketch (e.g., CM sketch) of the entire graph streams to a set of localized sketches to achieve better performance.

Conclusion of the control layer: In this section, we introduce several measurement systems which present novel mechanisms to manage existing methods in the operation layer. We summarize the main features of them in Table 2. However, none of the above methods are sufficient to properly manage complex and diverse measurement methods. For example, most of them need users to specify which measurement methods to use. In addition, they don't take Device Abstraction Layer(DAL) into account, which means they are not flexible and adaptive enough for practical networks, which may consist of different kinds of network devices. Here, we pose four requirements of the control layer:

- Free network administrators from the burden of selecting and configuring different (and also difficult) measurement methods. There are two ways to achieve this goal. The first is automatically selecting proper fine-grained measurement methods according to specific measurement tasks and underlying data plane, and the other is using one simple data structure to perform all kinds of measurement tasks (like UnivMon [57]).
- Adaptively adjust the trade-off between measurement accuracy and resource occupancy according to time-varying flow characteristics. Parameter configuration is also a complicated job since there is a tight bind between measurement accuracy and resource provision in most approximate measurement methods.
- Support concurrent execution of multiple measurement tasks. When diagnosing network problems, single measurement metrics may be not enough. Network administrators need to perform multiple measurement tasks concurrently to locate the root cause.
- Globally merge measurement data and report analysis results to the application layer without occupying too much bandwidth and memory resources. Measurement tasks may run on different locations of

TABLE 2
Summary of Measurement Systems in the Control Layer

Measurement systems	Novel points
DREAM [35]	Design an accuracy estimator and a resource allocator to schedule multiple measurement tasks and manage the operation layer's measurement algorithms
SCREAM [55]	Manage sketch-based methods with cheap and power-efficient SRAM memory (compared with TCAM) and improve the accuracy estimator of DREAM to dynamically allocate resources
OpenSketch(C) [36]	Introduce sketch library and provide a sketch manager and a resource allocator to automatically manage sketches
SketchLearn [38]	Resolve resource conflicts automatically by learning sketches' statistical properties iteratively
UnivMon(C) [57]	Formulate tasks, and then assign sketching responsibilities to network elements and present the big switch abstraction
Ada-Sketches [108]	Provide time adaptive sketches that dynamically adjust sketch sizes and reallocate the limited memory resources
ElasticSketch(C) [48]	Collect measurement data by introducing sketch compression and merging strategy
SketchVisor(C) [53]	Use matrix interpolation and compressive sensing to eliminate the extra errors due to the fast path processing
gSketch [109]	Partition a virtual global sketch to a set of localized sketches to achieve better performance

We list the novel points of these measurement systems.

networks. When the control layer gathers measurement data from multiple switches, it may cause bandwidth overhead and interfere the normal traffic, especially for heavy traffic load scenarios.

The control layer is much more important than the operation layer and the application layer in a network measurement system since it acts as a manager role and is "stuck in the middle". We believe that there is still much room for improvement in the control layer of fine-grained network measurement. In addition to managing measurement methods in the operation layer, the controller also needs to support many applications running on the application layer. In the next subsection, we will introduce several measurement interfaces, which can describe measurement demands in different ways.

3.3 The Application Layer

When facing network management problems, it is difficult for network administrators to locate the root cause directly. Network administrators need to perform specific measurement tasks then gather some useful information to solve the problems. In traditional way, network administrators need to decide what to measure by their experiences, and perform measurement tasks manually, all of which cost a lot of time. From the software-defined perspective, this process can be abstracted as a measurement application programmed by a domain-specific language. In this section, we introduce several application layer's languages (or interfaces) proposed in recent years, which can be used to perform network queries or program measurement applications, then make network measurement easier and reusable.

Frenetic [110] is a domain-specific language for programming OpenFlow networks, which comprises two integrated sublanguages: one is a limited but high-level and declarative network query language, while the other is a general-purpose, functional and reactive network policy management library. Frenetic is embedded in Python which makes programming more convenient. Frenetic query language allows filtering a set of packets, subdividing the set by grouping on the header fields, splitting these sets by arrival time or whenever a header field value changes, limiting the number of returned values, and aggregating packets. Then, there are several main syntactic elements. As shown in Fig. 13, $Select(a)$ clause aggregates the results using the method a, where a can be one of the packets, counts or bytes. $Where(fp)$ clause filters the results and retains the packets which satisfy the filter pattern *fp*. $GroupBy([h_1, \ldots, h_n])$ clause subdivides the set of packets into subsets based on header fields h_1 through h_n. $SplitWhen([h_1, \ldots, h_n])$ clause also subdivides the packets into subsets. The difference is that it generates a new subset each time the value of one of the given fields changes. For example, if a query splits on the source IP address, the packet stream with source IP A, B, A will be split into three subsets because that their IP addresses differ from the preceding. If the stream sequence is A, A, B, then only two subsets will be generated. $Every(n)$ clause partitions packets, and the packets which arrive within the same n-second window will be grouped. $Limit(n)$ clause limits the number of packets in each subset. The result of a query is an event stream which represents a stream of values. The library helps to manage packet forwarding policy with functional and reactive programming. One of the basic operations performed by a Frenetic program is to construct packet-forwarding rules for installation on switches. Thus, Frenetic programs create network policies and control the installation of policies in networks with the library. The query language of Frenetic defines what kind of packets need to be monitored. This language allows to measure the packets with some specific field values and only focus on a necessary part of the traffic while measuring a specific task. This query

```
Aggregates  a ::= packets | sizes | counts
Headers     h ::= inport | srcMac | dstMac | ethtype | vlan | srcIP
                  | dstIP | protocol | srcPort | dstPort | switch
Queries     q ::= Select (a) *
                    Where (fp) *
                    GroupBy ([h1, ..., hn]) *
                    SplitWhen ([h1, ..., hn]) *
                    Every (n) *
                    Limit (n)
```

Fig. 13. Syntax of frenetic query [110].

```
header   ::= srcMac | srcIP | dstMac | dstIP | ...
location ::= switch | inport | outport
field    ::= location | header
pred     ::= true | false | field = value | ingress() | egress()
           | pred & pred | (pred | pred) | ~pred
atom     ::= in_atom(pred) | out_atom(pred) | in_out_atom(pred, pred)
           | in_group(pred, [header]) | out_group(pred, [header])
           | in_out_group(pred, [header], pred, [header])
path     ::= atom | path & path | (path | path) | ~path | path ^ path
           | path*
```

Fig. 14. Syntax of path query language [111].

language gives a general filter that can significantly decrease the scale of packets which then will be processed by sketches or counter tuples.

Path Query [111] measurement language is proposed not only for traffic measurement, but also for whole network monitoring. Narayana *et al.* in [112] develops a path query language where operators specify regular expressions over boolean conditions on packet location and header contents. The switches in a network are programmed to record path information in each packet as it flows through. The queries will be compiled into a deterministic finite automaton (DFA) whose implementation is distributed across the switches. The state of DFA is stored in each packet and is updated when the packet traverses networks. Switches read the current state of DFA while receiving a packet, then check the queries, and write a new state to the packet. A path query identifies the set of packets with particular header values and traversing particular locations. The boolean predicate is a basic part of path query which matches a packet at a single location and may match on standard header fields such as $srcIP$. The predicates can be related to topology. $Ingress()$ can match all packets that enter the network at a specific ingress interface, $egress()$ then matches all packets that exit the network at a specific egress interface. Since the set of packets matching a given predicate may be different at the switch entry and exit due to headers rewriting or packets dropping, $atoms$ is used to refine the meaning of predicates. In_atom tests packets as they enter the switch, and out_atom tests them when they leave. $In\_out\_atom(pred, pred)$ tests one predicate when packets enter, and tests another when packets leave. $Atoms$ also supports group operation, for example, $in\_group(pred, [h_1, \ldots, h_n])$ collects packets that match the $pred$ at switch ingress, and then divides them into different groups by the header values $[h_1, \ldots, h_n]$. Out_group and In_out_group are similar. Path queries can be described by combining atoms using regular combinators: &, |, $\sim$ are standard which means both satisfy, either satisfy, not satisfy, respectively; $\wedge$ is a concatenation that the query $p1 \wedge p2$ specifies a path that satisfies $p1$ in current switch, the next-hop switch satisfies $p2$; $*$ is repetition that $p1*$ specifies paths that are zero or more repetitions of paths satisfying $p1$. The syntax summary is shown in Fig. 14. The packets matching a query can be counted, sent to a specific port or the SDN controller and so on. Operators also can specify where along a path to capture the packet that satisfies a query, either upstream, downstream or somewhere in between.

Marple [54] provides a language that extends Path Query and can express a large variety of performance monitoring use cases. With dedicated hardware, Marple language can get more detailed information of switches like queue length, packet timestamps of in and out switches. The design goal of Marple language is not for simple traffic measurement, it aims to conduct performance analysis across flows it monitored. Marple provides an abstraction of performance information of network streams. The streams contain tuples that record performance metadata such as queue lengths and timestamps when a packet entered queues. Then, these streams are recorded as $pktstream$ which are base input streams. Marple provides one tuple for each packet at each queue with seven fields: $(switch, qid, hdrs, uid, tin, tout, qsize)$. $switch$ and qid denote the switch and queue at which the packet was observed. $hdrs$ can record regular packet headers, and uid uniquely determines a packet. tin and $tout$ denote the enqueue and dequeue timestamps of this packet, while $qsize$ denotes the queue length when the packet is enqueued. Based on these metadata, Marple provides four functional constructs: $filter, map, groupby, zip$, as shown in Table 3. A filter has the form $filter(R, pred)$ where R is some $pktstream$, the predicate $pred$ can restrict users' attention to the interesting packets which satisfy the predicate. Map has the form $map(R, [expression], [field])$ that can compute fields to express new quantities of interest. For example, $map(pktstream, [tin/epoch\_size], [epoch])$ can round the packet timestamps to a new field 'epoch'. $groupby$ has the form $groupby(R, [fields], fun)$ where input stream R will be partitioned by $fields$ and an aggregation function fun will operate over each group. For example, $groupby(pktstream, [5 - tuple], count)$ can count packets belonging to the same 5-tuple flow. zip with the form $zip(R, S)$ can merge fields. The

TABLE 3
Summary of Marple Language Constructs [54]

Construct	Description
$filter(R, pred)$	Output tuples in R which satisfy predicate $pred$
$map(R, [expression], [field])$	Evaluate $[expression]$ over fields of R, emitting tuples with new field $[field]$
$groupby(R, [fields], fun)$	Partition R by $[fields]$ and evaluate function fun over the partitioned group
$zip(R, S)$	Merge fields in tuples of R and S

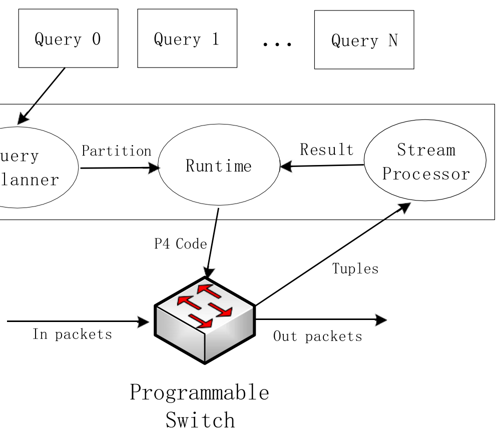


Fig. 15. Overview of Sonata [111].

TABLE 4
Comparison of Measurement Languages

Language	Usage	Requirements
Frenetic [110]	Filter packets with specific field values	Python language
Path Query [112]	Path tracing of packets	Program Switches with the Path Query functions
Marple [54]	Network performance monitoring	Switches with programmable KV-store
Sonata [111]	Network performance monitoring	Programmable switches and stream processors
ProgME [113]	Group packets to arbitrary set of flows	A binary decision diagram (BDD)-based data structure and Flowset-based Query Answering Engine (FQAE)
TPP [114]	Collect packet history	NetFPGA with TPP
Stroboscope [115]	Mirror and analyze traffic	Switches which support mirroring

We summarize the usage and the requirements of these languages.

output of *zip* operation over two input streams is a single stream containing tuples that are a concatenation of all the fields in the two streams. Marple language allows users to filter the packets which wait for a long time by *tout* and *tin* or which wait in a specific queue. It also provides the usual filtering rules of header fields. Thus, Marple language is mainly used to monitor network performance, which is an important usage in traffic measurement.

Sonata [111] is a system that allows operators to express queries using familiar data flow operators(e.g., map, filter, reduce). It indicates that Maple [54] and OpenSketch [36] are limited because they execute queries solely in the data plane, with insufficient memory and poor programmability. Sonata allows operators to view each packet as a tuple and expresses queries as operations over tuple streams. It partitions the workload between switches and stream processors, and then refines the configurations iteratively for the data plane (switches) and the processors to inspect traffic at a finer granularity. As it is shown in Fig. 15, Sonata compiles user queries to a set of rules installed in the switches, and installs processing pipelines at the stream processor. The packets are treated as tuples that each is a collection of field values including *srcIP*, *srcMac*, *srcPort*, *dstIP*, *dstMac*, *dstPort* and so on. Operators can specify whether a particular operation should execute in the switches or at the stream processors. For example, operators can apply a filter A at the switch and apply another filter B at the processor. Sonata also allows operators to express the logic for refining queries. The results from ongoing queries can drive refinements to existing queries and then iteratively query refinement occurrences.

Besides the above interfaces, OpenSketch(A) [36] provides *configure* and *query* APIs, which let users perform network measurement without considering concrete implementation of sketches. ProgME [113] proposes flowset composition language (FCL) to collects traffic statistics based on flowsets. TPP [114] uses programmatic interfaces and embeds tiny programs into packets to get network states. Stroboscope [115] introduces a SQL-like language to extract a traffic sample set by activating and deactivating traffic mirror for any destination prefixes.

Conclusion of the application layer: All above measurement languages or interfaces can describe measurement tasks in different ways. We give a summary of the application layer in Table 4. However, all of these languages are not general enough for all kinds of measurement tasks. More importantly, they will introduce significant performance loss when process stateful query operations such as *groupby* [110] and *reduce* [58]. Here, we pose four requirements of measurement languages or interfaces:

- General interfaces to define different kinds of measurement tasks: diverse measurement tasks are expected to support various management operations, thereby a general interface framework is needed to support the definition of all kinds of the measurement targets(e.g., Heavy hitter, DDoS).
- Can be compiled to a set of fine-grained measurement methods: it is important because we need to leverage existing fine-grained methods to perform network measurement.
- Explicit performance costs before running: it is also important because some measurement tasks will introduce significant performance loss inevitably [54]. A notice is needed before network administrator conducting the measurement tasks.
- Near real-time response time: the response time includes two parts. The first is introduced by transforming and installing measurement methods, and the second is how much time the measurement data can be collected and analyzed. Both of two parts need to be efficient.

The design of measurement languages or interfaces is an interesting but challenging topic. For fine-grained measurement systems, more universal and flexible languages or interfaces are needed.

4 Conclusion

In this paper, we present a comprehensive survey of recent fine-grained measurement methods and systems. We discuss these methods and systems from the software-defined perspective. In particular, we categorize existing works into three layers and discuss current limitations or future requirements of each layer.

For future directions of fine-grained network measurement, we believe that the control layer have much more room to improve. Compared with other network operations, the conducting process of network measurement is still original, especially for measuring high-speed networks in a fine-grained and dynamic way. The absence of a powerful control plane for fine-grained measurement is the major reason. The measurement interface can be regarded as a follow-up production since it is compiled and conducted by the corresponding controller. Last but not least, more efficient and accurate measurement algorithms are always needed to satisfy both old and new measurement demands in the operation layer.

Acknowledgments

The authors would like to thank anonymous reviewers for their valuable comments. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B0101390001, in part by the National Natural Science Foundation of China under Grants 61772265, 61802172, and 62072228, in part by the Fundamental Research Funds for the Central Universities, in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization, and in part by the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

References

[1] marketsandmarkets, “Network analytics market by type, by end user - Global forecast to 2020,” 2019. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/network-analytics-market-1244.html

[2] Y. Khalidi, “Azure networking fall 2018 update,” 2018. [Online]. Available: https://azure.microsoft.com/en-us/blog/azure-networking-fall-2018-update/

[3] I. Envid, “Introducing VPC flow logs network transparency in near real-time,” 2018. [Online]. Available: https://cloud.google.com/blog/products/gcp/introducing-vpc-flow-logs-network-transparency-in-near-real-time

[4] B. Claise, “Cisco systems netflow services export version 9,” *RFC*, vol. 3954, pp. 1–33, 2004.

[5] M. Wang, B. Li, and Z. Li, “sFlow: Towards resource-efficient and agile service federation in service overlay networks,” in *Proc. 24th Int. Conf. Distrib. Comput. Syst.*, 2004, pp. 628–635.

[6] G. S. Manku and R. Motwani, “Approximate frequency counts over data streams,” in *Proc. 28th Int. Conf. Very Large Data Bases*, 2002, pp. 346–357.

[7] C. Estan and G. Varghese, “New directions in traffic measurement and accounting,” in *Proc. Conf. Appl., Technol., Architect., Protocols Comput. Commun.*, 2002, pp. 323–336.

[8] Y. Zhang, “An adaptive flow counting method for anomaly detection in SDN,” in *Proc. 9th ACM Conf. Emerging Netw. Experiments Technol.*, 2013, pp. 25–30.

[9] M. Canini, D. Fay, D. J. Miller, A. W. Moore, and R. Bolla, “Per flow packet sampling for high-speed network monitoring,” in *Proc. 1st Int. Commun. Syst. Netw. Workshops*, 2009, pp. 1–10.

[10] D. Stutzbach, R. Rejaie, N. G. Duffield, S. Sen, and W. Willinger, “On unbiased sampling for unstructured peer-to-peer networks,” *IEEE/ACM Trans. Netw.*, vol. 17, no. 2, pp. 377–390, Apr. 2009.

[11] A. Mendiola, J. Astorga, E. Jacob, and M. Higuero, “A survey on the contributions of software-defined networking to traffic engineering,” *IEEE Commun. Surv. Tut.*, vol. 19, no. 2, pp. 918–953, Apr.-Jun. 2017.

[12] S. Agarwal, M. Kodialam, and T. Lakshman, “Traffic engineering in software defined networks,” in *Proc. IEEE INFOCOM*, 2013, pp. 2211–2219.

[13] M. Shafiee and J. Ghaderi, “A simple congestion-aware algorithm for load balancing in datacenter networks,” *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3670–3682, Dec. 2017.

[14] G. Cormode and S. Muthukrishnan, “An improved data stream summary: The count-min sketch and its applications,” *J. Algorithms*, vol. 55, pp. 58–75, 2005.

[15] G. Cormode and S. Muthukrishnan, “What’s new: Finding significant differences in network data streams,” *IEEE/ACM Trans. Netw.*, vol. 13, no. 6, pp. 1219–1232, Dec. 2005.

[16] R. T. Schweller *et al.*, “Reversible sketches: Enabling monitoring and analysis over high-speed data streams,” *IEEE/ACM Trans. Netw.*, vol. 15, no. 5, pp. 1059–1072, Oct. 2007.

[17] 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.

[18] 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.

[19] D. Ting, “Data sketches for disaggregated subset sum and frequent item estimation,” in *Proc. Int. Conf. Manage. Data*, 2018, pp. 1129–1140.

[20] P4 language. [Online]. Available: https://p4.org/

[21] B. Pfaf *et al.*, “The design and implementation of open vSwitch,” in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 117–130.

[22] Bess: Berkeley extensible software switch. [Online]. Available: https://github.com/NetSys/bess

[23] Tofino 2. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino-2/

[24] Scrutinizer. [Online]. Available: https://www.plixer.com/products/scrutinizer/

[25] Netflow analyzer PRTG. [Online]. Available: https://www.paessler.com/netflow_monitoring

[26] R. Hofstede *et al.*, “Flow monitoring explained: From packet capture to data analysis with netFlow and IPFIX,” *IEEE Commun. Surv. Tut.*, vol. 16, no. 4, pp. 2037–2064, Oct.-Dec. 2014.

[27] C. Estan, K. Keys, D. Moore, and G. Varghese, “Building a better netFlow,” *ACM SIGCOMM Comput. Commun. Rev.*, 2004, pp. 245–256.

[28] N. G. Duffield, C. Lund, and M. Thorup, “Estimating flow distributions from sampled flow statistics,” in *Proc. Conf. Appl., Technol., Architect., Protocols Comput. Commun.*, 2003, pp. 325–336.

[29] N. Hohn and D. Veitch, “Inverting sampled traffic,” *IEEE/ACM Trans. Netw.*, vol. 14, no. 1, pp. 68–80, Feb. 2006.

[30] Y. Zhang, M. Roughan, W. Willinger, and L. Qiu, “Spatio-temporal compressive sensing and internet traffic matrices (extended version),” *IEEE/ACM Trans. Netw.*, vol. 20, no. 3, pp. 662–676, Jun. 2012.

[31] 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.

[32] M. Charikar, K. C. Chen, and M. Farach-Colton , “Finding frequent items in data streams,” in *Proc. 29th Int. Colloquium Automata, Lang. Program.*, 2002, pp. 693–703.

[33] E. D. Demaine, A. López-Ortiz, and J. I. Munro, “Frequency estimation of internet packet streams with limited space,” in *Proc. 10th Annu. Eur. Symp. Algorithms*, 2002, pp. 348–360.

[34] G. S. Manku and R. Motwani, “Approximate frequency counts over data streams,” in *Proc. 28th Int. Conf. Very Large Data Bases*, pp. 346–357.

[35] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, “DREAM: Dynamic resource allocation for software-defined measurement,” in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 419–430.

[36] M. Yu, L. Jose, and R. Miao, “Software defined traffic measurement with opensketch,” in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 29–42.

[37] J. Shan, Y. Fu, G. Ni, J. Luo, and Z. Wu, “Fast counting the cardinality of flows for big traffic over sliding windows,” *Front. Comput. Sci.*, vol. 11, no. 1, pp. 119–129, 2017.

[38] Q. Huang, P. P. C. Lee, and Y. Bao, “Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference,” in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2018, pp. 576–590.

[39] A. D’Alconzo, I. Drago, A. Morichetta, M. Mellia, and P. Casas, “A survey on big data for network traffic monitoring and analysis,” *IEEE Trans. Netw. Serv. Manage.*, vol. 16, no. 3, pp. 800–813, Sep. 2019.

[40] R. J. L. Wang, “Big data analytics for network intrusion detection: A survey,” *Int. J. Netw. Commun.*, vol. 7, no. 1, pp. 24–31, 2017.

[41] C. So-in, “A survey of network traffic monitoring and analysis tools,” *Cse 576m Computer System Analysis Project*, Washington University, St. Louis, USA, 2009.

[42] Y. Azzana, Y. Chabchoub, C. Fricker, F. Guillemin, and P. Robert, “Adaptive algorithms for the identification of large flows in IP traffic,” 2009, *arXiv:0901.4846*.

[43] A. Metwally, D. Agrawal, and A. El Abbadi , “Efficient computation of frequent and top-K elements in data streams,” in *Proc. 10th Int. Conf. Database Theory*, 2005, pp. 398–412.

[44] R. Ben-Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, “Constant time updates in hierarchical heavy hitters,” in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 127–140.

[45] G. Cormode and S. Muthukrishnan, "What's new: Finding significant differences in network data streams," *IEEE/ACM Trans. Netw.*, vol. 13, no. 6, pp. 1219–1232, Dec. 2005.

[46] N. Kamiyama, T. Mori, and R. Kawahara, "Simple and adaptive identification of superspreaders by flow sampling," in *IEEE INFOCOM - 26th IEEE Int. Conf. Comput. Commun*, 2007, pp. 2481–2485.

[47] Y. Xu and Y. Liu, "DDoS attack detection under SDN context," in *IEEE INFOCOM - 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.

[48] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2018, pp. 561–575.

[49] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[50] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *Proc. Int. Conf. Manage. Data*, 2017, pp. 775–787.

[51] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *Int. J. Very Large Data Bases*, vol. 19, no. 1, pp. 3–20, 2010.

[52] Q. Huang and P. P. C. Lee, "A hybrid local and distributed sketching design for accurate and scalable heavy key detection in network data streams," *Comput. Netw.*, vol. 91, pp. 298–315, 2015.

[53] Q. Huang *et al.*, "SketchVisor: Robust network measurement for software packet processing," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 113–126.

[54] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 85–98.

[55] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "SCREAM: Sketch resource allocation for software-defined measurement," in *Proc. 11th ACM Conf. Emerging Netw. Experiments Technol.*, 2015, pp. 14:1–14:13.

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

[57] 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.

[58] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2018, pp. 357–371.

[59] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: A novel counter architecture for per-flow measurement," in *ACM SIGMETRICS Perform. Eval. Rev.*, 2008, pp. 121–132.

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

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

[62] 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 Interest Group Data Commun. Appl., Technol., Architect., Protocols Comput. Commun.*, 2020, pp. 226–239.

[63] 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.

[64] L. Tang, Q. Huang, and P. P. C. Lee, "MV-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams," in *Proc. IEEE INFOCOM - IEEE Conf. Comput. Commun.*, 2019, pp. 2026–2034.

[65] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better netFlow for data centers," in *Proc. 13th USENIX Conf. Netw. Syst. Des. Implementation*, 2016, pp. 311–324.

[66] L. Tang, Q. Huang, and P. P. C. Lee, "SpreadSketch: Toward invertible and network-wide detection of superspreaders," in *Proc. IEEE INFOCOM - IEEE Conf. Comput. Commun.*, 2020, pp. 1608–1617.

[67] T. Bu, J. Cao, A. Chen, and P. P. C. Lee, "Sequential hashing: A flexible approach for unveiling significant patterns in high speed networks," *Comput. Netw.*, vol. 54, no. 18, pp. 3309–3326, 2010.

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

[69] A. Goyal and H. Daumé, "Lossy conservative update (LCU) sketch: Succinct approximate count storage," *Proc. 25th AAAI Conf. Artif. Intell.*, vol. 25, no. 1, pp. 878–883, 2011.

[70] Z. Liu, G. Vorsanger, V. Braverman, and V. Sekar, "Enabling a 'RISC' approach for software-defined monitoring using universal streaming," in *Proc. 14th ACM Workshop Hot Top. Netw.*, 2015, pp. 21:1–21:7.

[71] X. Li *et al.*, "Detection and identification of network anomalies using sketch subspaces," in *Proc. 6th ACM SIGCOMM Conf. Internet Measure.*, 2006, pp. 147–152.

[72] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, pp. 217–228, 2005.

[73] Y. Liu, W. Chen, and Y. Guan, "A fast sketch for aggregate queries over high-speed network traffic," in *Proc. IEEE INFOCOM*, 2012, pp. 2741–2745.

[74] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1481–1496.

[75] J. Chen and Q. Zhang, "Bias-aware sketches," in *Proc. VLDB Endowment*, vol. 10, no. 9, pp. 961–972, 2017.

[76] T. Yang *et al.*, "Empowering sketches with machine learning for network measurements," in *Proc. Workshop Netw. Meets AI ML*, 2018, pp. 15–20.

[77] M. Mitzenmacher, R. Pagh, and N. Pham, "Efficient estimation for high similarities using odd sketches," in *Proc. Int. World Wide Web Conf.*, 2014, pp. 109–118.

[78] 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.

[79] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 182–209, 1985.

[80] K. Whang, B. T. V. 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.

[81] M. Durand and P. Flajolet, "Loglog counting of large cardinalities (extended abstract)," in *Proc. Eur. Symp. Algorithms*, 2003, pp. 605–617.

[82] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," *Proc. Conf. Anal. Algorithms Discrete Math. Theor. Comput. Sci.*, vol. AH, no. 1, pp. 137–156, 2007.

[83] O. Ertl, "New cardinality estimation algorithms for hyperLogLog sketches," 2017, *arXiv:1702.01284*.

[84] A. Lall, V. Sekar, M. Ogihara, J. J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," in *ACM SIGMETRICS Perform. Eval. Rev.*, 2006, pp. 145–156.

[85] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proc. Internet Measure. Conf.*, 2017, pp. 78–85.

[86] M. Mitzenmacher, T. Steinke, and J. Thaler, "Hierarchical heavy hitters with the space saving algorithm," 2011, *arXiv:1102.5540*.

[87] M. Yoon, T. Li, S. Chen, and J. Peir, "Fit a spread estimator in small memory," in *Proc. IEEE INFOCOM*, 2009, pp. 504–512.

[88] H. Song, S. Dharmapurikar, J. S. Turner, and J. W. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2005, pp. 181–192.

[89] S. Cohen and Y. Matias, "Spectral bloom filters," in *ACM Proc. Int. Conf. Manage. Data*, 2003, pp. 241–252.

[90] M. T. Goodrich and M. Mitzenmacher, "Invertible bloom lookup tables," 2011, *arXiv:1101.2245*.

[91] M. Yoon, J. Son, and S. Shin, "Bloom tree: A search tree based on bloom filters for multiple-set membership testing," in *Proc. IEEE INFOCOM - IEEE Conf. Comput. Commun.*, 2014, pp. 1429–1437.

[92] Y. Peng, J. Guo, F. Li, W. Qian, and A. Zhou, "Persistent bloom filter: Membership testing for the entire history," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 1037–1052.

[93] 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 Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, 2020, pp. 404–421.

[94] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "CSAMP: A system for network-wide flow monitoring," in *Proc. 5th USENIX Symp. Netw. Syst. Des. Implementation*, 2008, pp. 233–246.

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

[96] L. Jose and M. Yu, "Online measurement of large traffic aggregates on commodity switches," in *Proc. 11th USENIX Conf. Hot Top. Manage. Internet, Cloud, Enterprise Netw. Serv.*, 2011, 13 p.

[97] M. Malboubi, L. Wang, C. Chuah, and P. Sharma, "Intelligent SDN based traffic (de)aggregation and measurement paradigm (iSTAMP)," in *Proc. IEEE INFOCOM - IEEE Conf. Comput. Commun.*, 2014, pp. 934–942.

[98] N. McKeown *et al.*, "OpenFlow: enabling innovation in campus networks," *Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[99] 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. 11th USENIX Conf. Netw. Syst. Des. Implementation*, 2014, pp. 71–85.

[100] C. Guo *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, pp. 139–152, 2015.

[101] S. Kandula, S. Sengupta, A. G. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. 9th ACM SIGCOMM Conf. Internet Measure.*, 2009, pp. 202–208.

[102] M. Yu *et al.*, "Profiling network performance for multi-tier data center applications," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 57–70.

[103] Z. Su, T. Wang, Y. Xia, and M. Hamdi, "FlowCover: Low-cost flow monitoring scheme in software defined networks," in *Proc. IEEE Global Commun. Conf*, 2014, pp. 1956–1961.

[104] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. B. Carter, "OpenSample: A low-latency, sampling-based measurement platform for commodity SDN," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 228–237.

[105] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in opeFflow software-defined networks," in *Proc. IEEE Netw. Oper. Manage. Symp.*, 2014, pp. 1–8.

[106] J. Rasley *et al.*, "Planck: Millisecond-scale monitoring and control for commodity networks," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 407–418.

[107] Y. Zhu *et al.*, "Packet-level telemetry in large datacenter networks," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 479–491.

[108] A. Shrivastava, A. C. König, and M. Bilenko, "Time adaptive sketches (ada-sketches) for summarizing data streams," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1417–1432.

[109] P. Zhao, C. C. Aggarwal, and M. Wang, "gSketch: On query estimation in graph streams," *Proc. VLDB Endowment*, vol. 5, no. 3, pp. 193–204, 2011.

[110] N. Foster *et al.*, "Frenetic: a network programming language," in *ACM SIGPLAN Notices*, vol. 46, pp. 279–291, 2011.

[111] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker , and W. Willinger, "Network monitoring as a streaming analytics problem," in *Proc. 15th ACM Workshop Hot Top. Netw.*, 2016, pp. 106–112.

[112] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker, "Compiling path queries," in *Proc. 13th USENIX Conf. Netw. Syst. Des. Implementation*, 2016, pp. 207–222.

[113] L. Yuan, C. Chuah, and P. Mohapatra, "ProgME: Towards programmable network measurement," in *IEEE/ACM Trans. Netw.*, vol. 19, pp. 115–128, 2011.

[114] 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*, 2014, pp. 3–14.

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

Hao Zheng received the BS degree from the Department of Software Engineering, Southeast University, China, in 2020. He is currently working toward the PhD degree with the Department of Computer Science and Technology, Nanjing University, China. His research interests include software-defined network and network measurement.

Yanan Jiang received the BS degree from the Department of Software Engineering, Jilin University, China, in 2017. She is currently working toward the MS degree with the Department of Computer Science and Technology, Nanjing University, China. Her research interests include datacenter networks and network measurement.

Chen Tian received the BS, MS, and PhD degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2008, respectively. He is currently an associate professor with State Key Laboratory of Novel Software Technology, Nanjing University, China. He was previously an associate professor with the School of Electronics Information and Communications, Huazhong University of Science and Technology, China. From 2012 to 2013, he was a postdoctoral researcher with the Department of Computer Science, Yale University. His research interests include data center networks, distributed systems, Internet streaming, and urban computing.

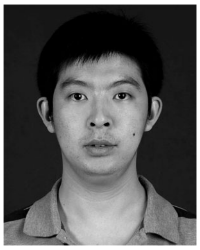
Long Cheng received the first PhD degree from the Beijing University of Posts and Telecommunications, China, in 2012, and the second PhD degree in computer science from Virginia Polytechnic Institute and State University, USA, in 2018. He is currently an assistant professor of computer science with Clemson University, USA. He was a research scientist with the Institute for Infocomm Research, Singapore, from 2014 to 2015, and a research fellow with the Singapore University of Technology and Design from 2012 to 2014. His research interests include system and network security, cyber-physical systems (CPS) security, Internet of Things, mobile computing, and wireless networks.

Qun Huang received the bachelor's degree from Peking University in 2011 and the PhD degree from The Chinese University of Hong Kong in 2015. He is currently an assistant professor with Peking University. Before joining Peking University, he was with the Institute of Computing Technology, Chinese Academy of Sciences, and Huawei. His research interests include network measurement and distributed systems.

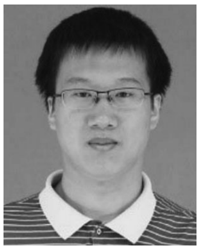
Weichao Li (Member, IEEE) received the BE and ME degrees from the South China University of Technology, Guangzhou, China, in 2002 and 2009, respectively, and the PhD degree from the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, in 2017. He is currently a research assistant professor with the Institute of Future Networks, Southern University of Science and Technology, China. His research focuses on network measurement, especially on Internet measurement technologies, large-scale network performance monitoring and diagnosis, and mobile network performance monitoring. He is also interested in areas including SDN, industrial internet, cloud computing, quality of experience (QoE) measurement, and machine learning.

Yi Wang received the PhD degree in computer science and technology from Tsinghua University in July 2013. He is currently a research associate professor with the Sustech Institute of Future Networks, Southern University of Science and Technology. Before joining Sustech, he was a senior researcher with the Huawei Future Network Theory Lab, Hong Kong. His research interests include Future network architectures, information centric networking, software-defined networks, and the design and implementation of high-performance network devices.

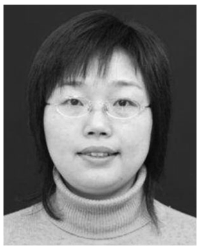
Qianyi Huang (Member, IEEE) received the bachelor's degree in computer science from Shanghai Jiao Tong University and the PhD degree from the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology. She is currrently a research assistant professor with the Southern University of Science and Technology, Shenzhen, China. She has authored or coauthored a number of papers in top-ranking journals and conferences, including the *IEEE/ACM Transactions on Networking* (TON), *IEEE Transactions on Mobile Computing* (TMC), MobiCom, UbiComp and INFOCOM. Her research interests include mobile computing, Internet-of-Things, and security.



Jiaqi Zheng received the PhD degree from Nanjing University, China, in 2017. He was a research assistant with the City University of Hong Kong in 2015, and a visiting scholar with Temple University in 2016. He is currently a research assistant professor with the Department of Computer Science and Technology, Nanjing University. His research interests include computer networking, particularly, data center networks, SDN/NFV, and machine learning systems. He is a member of the ACM. He was the recipient of the Best Paper Award from the IEEE ICNP 2015, the Doctoral Dissertation Award from ACM SIGCOMM China 2018, and the First Prize of the Jiangsu Science and Technology Award in 2018.



Rui Xia received the BS degree from the Department of Software Engineering, Nankai University, China, in 2017. He is currently working toward the PhD degree with the Department of Computer Science, Nanjing University, China. His research interests include software-defined network and network function virtualization.



Yi Wang received the BS, MS, and PhD degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2009 respectively. She is currently a lecturer with the School of Modern Posts, Nanjing University of Posts and Telecommunications, China. Her research interest focuses on cloud computing.

Wanchun Dou received the PhD degree in mechanical and electronic engineering from the Nanjing University of Science and Technology, China, in 2001. He is currently a full professor with the State Key Laboratory of Novel Software Technology, Nanjing University. He visited the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, as a visiting scholar from April 2005 to June 2005 and from November 2008 to February 2009. He has authored or coauthored more than 60 research papers in international journals and international conferences. His research interests include workflow, cloud computing, and service computing. He has chaired three National Natural Science Foundation of China projects.

Guihai Chen received the BS degree in computer software from Nanjing University in 1984, the ME degree in computer applications from Southeast University in 1987, and the PhD degree in computer science from The University of Hong Kong in 1997. He is currently a distinguished professor with Nanjing University. He had been invited as a visiting professor by the Kyushu Institute of Technology, Japan; University of Queensland, Australia, and Wayne State University, USA. He has a wide range of research interests, which include parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture, and data engineering. He has authored or coauthored more than 350 peer-reviewed papers, and more than 200 of them are in well-archived international journals, such as the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE/ACM Transactions on Networking*, and *ACM Transactions on Sensor Networks*, and also in well-known conference proceedings, such as HPCA, MOBIHOC, INFOCOM, ICNP, ICDCS, CoNext, and AAAI. He was the recipient of nine paper awards including ICNP 2015 Best Paper Award and DASFAA 2017 Best Paper Award.