Continuous Flow Measurement with SuperFlow

Zongyi Zhao*, Xingang Shi*‖§, Arpit Gupta¶, Qing Li†‡, Zhiliang Wang*‖, Bin Xiong*, Xia Yin*‖
zhaozong16@mails.tsinghua.edu.cn shixg@cernet.edu.cn arpitgupta@cs.ucsb.edu
liq8@sustc.edu.cn wzl@cernet.edu.cn xb19@mails.tsinghua.edu.cn yxia@tsinghua.edu.cn
*Tsinghua University, †Southern University of Science and Technology, ¶UC Santa Barbara
‡PCL Research Center of Networks and Communications, Peng Cheng Laboratory, Shenzhen
‖Beijing National Research Center for Information Science and Technology (BNRist), §Corresponding Author

***Abstract*—Flow-based network measurement enables operators to perform a wide range of network management tasks in a scalable manner. Recently, various algorithms have been proposed for flow record collection at very high speed. However, they all focus on processing traffic in a short time window, but overlook the fact that flow measurements are typically needed continuously for unlimited time. To this end, we propose a new algorithm named SuperFlow to support continuous and accurate flow record collection at very high speed by monitoring the flow activeness and exporting the inactive records from the data plane automatically. Our data structures and the corresponding algorithms are carefully designed and analyzed, so the above goal is achieved with limited memory and bandwidth consumption. We implement SuperFlow on both x86 CPU and state-of-the-art PISA target. Comprehensive experiments show that SuperFlow consistently outperforms its competitors significantly. Especially, compared with the best competitor, it records around 136.7% more flows, reduces the error in flow size estimation by 51.5%, and reduces the memory or bandwidth consumption by up to 71.0%, while bringing only negligible throughput degradation.**

I. Introduction

Statistics produced by network measurement tools enable network operators to perform a wide range of network management tasks, e.g., traffic engineering, error diagnosis and attack detection [1]. Flow-level measurement tools such as NetFlow [2] and IPFIX [3] can aggregate packets with the same flow ID[1] into flows, and report certain properties of each flow. Such tools are believed to make a good balance between scalability and informativeness, which is what the packet sniffing tools [4] and the highly summarized SNMP counters [5] fail to achieve. To meet the ultra-high link speed (e.g., 100 Gbps), it is also well recognized that the flow measurement tools have to be augmented with sampling. However, even the most elaborate sampling and estimation algorithms [6–8] under typical settings, e.g., with a sampling rate of 1/1000 for 100 Gbps traffic, severely reduces the accuracy of the reported results.

In another direction, sketch [9–11] (i.e., some succinct data structure) based algorithms have been proposed for efficient tracking and accurate estimation of specific traffic statistics such as the total number of flows and the flow size distribution, but their functional specificity hinders their wide adoption in practice. Recently, advanced sketches [12–19] have been proposed to track flow-level records at ultra-high speed by leveraging the hardware pipelines and a very small amount of on-chip memory. Unfortunately, they all focus on processing traffic in a short time epoch, but overlook the fact that flow measurements are typically needed continuously for unlimited time. Without a proper method to make room for new flow records by exporting the existing ones, they definitely cannot be used for actual deployment. A possible solution [15] is to use two sketches alternately, such that at the end of a pre-configured time epoch, the idle sketch takes over to track flow records in the next epoch, while the first sketch exports flow records it has tracked, gets reset, and waits for its turn, and so on. As this method makes continuous flow measurement with those sketches possible, it is far from perfect for three reasons: (1) the memory utilization is low because the early completed flows, instead of being evicted immediately, have to stay in the memory until the end of the current epoch; (2) the epoch, once fixed, cannot adapt to traffic fluctuations such as abrupt changes of flow concurrency; (3) it's challenging to pick a proper epoch length which strikes a good balance between memory and bandwidth consumption due to the existence of great number of long-lived flows.

To this end, we have designed and implemented SuperFlow, an algorithm to support continuous and accurate flow record collection at ultra-high speed for unlimited time. Our data structures, as well as the flow record maintenance and export procedures, are carefully designed and analyzed, so scalability and accuracy, as well as informativeness and deployability, are achieved with limited memory and bandwidth. Specifically, we make the following contributions:

- We propose a framework that continuously tracks flow records and automatically exports them.
- We design the sketch structures and the corresponding algorithms for flow record maintenance. In particular, we design a new sketch for flow size estimation, and provide the corresponding theoretical analyses.
- We design a flow record export scheme which detects and exports inactive flow records automatically to accommodate active flows. Only with this scheme can SuperFlow work continuously for unlimited time.
- We propose an optimization for large flow ID spaces (e.g., that of IPv6 traffic), which makes the consumption of high-speed memory independent of the flow ID length

[1]Flow ID can be a set of packet fields like network prefix, address, transport layer port, or even a keyword. By default, we will use <srcAddr, destAddr, srcPort, dstPort, protocol> as flow ID if not specified otherwise.

with little bandwidth overhead.

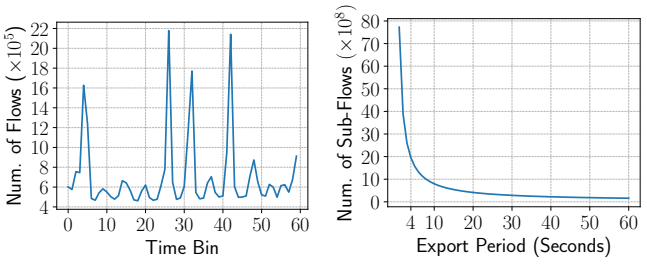
- We implement SuperFlow on both x86 CPU and a state-of-the-art Barefoot PISA switch [20], and evaluate its performance against five latest flow measurement algorithms [12–16] under fair settings.

Under various traffic patterns, SuperFlow consistently outperforms its competitors significantly. For example, compared with the best competitor, it records around 136.7% more flows, reduces the error in flow size estimation by 51.5%, and reduces the memory or bandwidth consumption by up to 71.0%, while bringing only negligible throughput degradation.

The remainder of the paper is organized as follows. We first introduce the motivation and basic ideas of SuperFlow in Section II, then present the design details in Section III. In Section IV, the performance of SuperFlow is evaluated and compared with that of five state-of-the-art competitors. Finally, we conclude the paper in Section V.

II. Motivation and Basic Ideas

The widely used NetFlow [2] provides continuous tracking of traffic flows, but providing such a functionality at ultra-high traffic speed is very difficult, if not impossible, due to the conflict between the required memory space, the processing overhead, and the resulting accuracy. For example, sampling methods [6, 7, 21, 22] can reduce the processing overhead but at the cost of much worse accuracy. Since sketch based methods can take advantage of the ultra-high processing capacity of modern hardware and work under very tight memory restrictions (e.g., a few megabytes), achieving reasonably good accuracy meanwhile, we believe they are more promising to keep up with the ever-increasing traffic speed. However, to implement *continuous tracking* in sketches is not easy, because the explicit "termination" flags usually don't exist in various operator-defined flows, *waiting a flow until it has been inactive for a certain time period* usually incurs significant processing and memory overhead, and exporting a single flow record is impossible for some sketches. Therefore, the existing sketch based flow measurement algorithms [12–16] have all overlooked this problem, and focus only on processing traffic in a small time epoch. A natural remedy to this, as suggested in [15], is to adopt the *periodical export* model, by which the data plane is exported periodically.



(a) Number of Concurrent Flows (b) Flow Lifetimes

Fig. 1. The number of flows in each one-minute time bin and the number of sub-flows corresponding to various export periods.

Unfortunately, there are several limitations in this model which heavily undermines its practicality. First, to avoid disrupting the flow record collection, usually two independent copies of the data structures are required, so that one copy can track flows while the other is being exported. This doubles the memory consumption, which is undesirable because of the scarcity of memory in commodity switches. Second, many flows may go inactive far before the end of the export period, but cannot be exported until this period ends, thus preventing the other active flows from being recorded, which degrades the memory utilization. Third, the number of concurrent flows often fluctuates greatly. For example, Fig. 1(a) shows that, in a CAIDA trace [23] spanning one hour, the maximum number of flows within a one-minute time bin may be $4.7\times$ as large as the minimum one. Therefore, with a fixed export period, traffic spikes may easily overwhelm the sketch. Most importantly, it's challenging to pick an export period which strikes a good balance between the memory and bandwidth consumption due to the existence of the great number of long flows. As an example, there are 0.4×10^8 distinct flows in the CAIDA trace. As shown in Fig. 1(b), if we adopt a relatively small export period of 4 seconds, as the flows in a single export period are small enough in number to fit into a fair amount of memory, up to 19.5×10^8 sub-flows will be exported, increasing the bandwidth consumption by $54.1\times$ compared with the case where each flow is exported exactly once. On the contrary, no more than 1.6×10^8 sub-flows will be exported if we set the export period to 60 seconds, but up to 2.2×10^6 flows may exist in a single export period (as shown in Fig. 1(a)), and, supposing that each flow record occupies 15 bytes (i.e., 13 bytes for the typical 5-tuple, and 2 bytes for the counter), 31.2 MB of memory is required to accommodate the flows, which are far greater than that the commodity switches can provide for the flow measurement algorithms [2].

In this paper, we argue that a simple scheme which we call *Inactive Flow Eviction (IFE)*, when combined with the specially designed data structures, may overcome these defects effectively. Specifically, *IFE* should try its best to sense the flow activeness, keep active flows in memory, and evict the inactive flows to save space for newly started ones. The policy of evicting inactive flows, we believe, is the key to allow a sketch to function well for unlimited time without performance degradation. With *IFE*, a long flow can stay in the memory as long as it is active, avoiding the bandwidth overhead caused by the frequent exports of its segments. But once it gets stale, it will be evicted to make room for the new flows, thus improving the memory utilization. On the other hand, *IFE* should be able to adapt to traffic spikes by exploiting the transient activeness of flows and evicting aged records. For example, the same memory space can be used to accommodate f_1, then used to accommodate f_2 when f_2 grows more active, and then used to accommodate f_3 when f_3's activeness exceeds that of f_2.

Another technique we adopt in designing SuperFlow is to

[2]Our P4 switch has only a few hundreds of Mb of SRAM, and they have to be shared between many different functions.

decouple memory consumption from flow ID size, by storing in memory the shorter flow ID fingerprints and recovering the full ID in the control plane later. This mechanism can effectively reduce the memory consumption when coping with large flow IDs, such as the typical five-tuple flow IDs which are of 37 bytes in IPv6.

III. The Design of SuperFlow

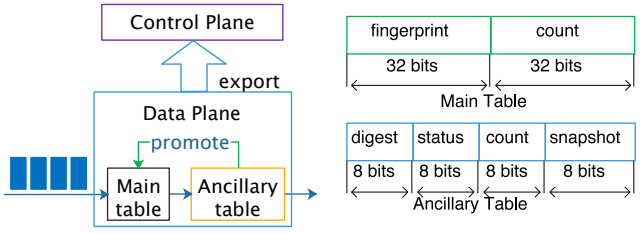
A. Overall Framework and Data Structures

As shown in Fig. 2(a), SuperFlow consists of two parts. The data plane part is responsible for flow record maintenance and export, and can be implemented in the pipelines of a PISA switch. It contains two tables, i.e., a main table $\mathbf{M}$ and an ancillary table $\mathbf{A}$, where the former is further split into d ($d = 3$ by defult) sub-tables $\mathbf{M}_1, \cdots, \mathbf{M}_d$. The control plane part receives records from the data plane, then decodes and saves them. The structure of each bucket in $\mathbf{M}_i$ and $\mathbf{A}$ is illustrated in Fig. 2(b). In a bucket of $\mathbf{M}_i$, there are two 32-bit fields, namely fingerprint and count. In a bucket of $\mathbf{A}$, there are four 8-bit fields, which are digest, status, count and snapshot. Besides, we have a set of independent hash functions $h_1, \cdots, h_d$ and g. The main workflow is as follows.

When a packet arrives, SuperFlow extracts its flow ID f as the key[3], and in a pipeline fashion, uses h_i as an index function to map it into a bucket in each $\mathbf{M}_i$. A 32-bit fingerprint is computed for f, and by comparing it with the fingerprint fields of the buckets, we can tell (approximately) whether all of the d buckets are already occupied by flows other than f. If this happens, the packet will be passed to $\mathbf{A}$ which uses g for indexing, as will be described a little later. Otherwise, we arrive at an empty bucket or a bucket with an existing record for f in $\mathbf{M}$, so we update it accordingly.

If a packet of f is passed to the ancillary table $\mathbf{A}$, some information about f can be estimated there, based on which we can tell whether f is larger or more active than some flow stored in $\mathbf{M}$. If this is true, we export the latter one and use its bucket space to accommodate the former. With this scheme, flow records are continuously maintained and exported.

Next, we explain the detailed algorithm, and present the pseudo code in Algorithm 1.



(a) Framework of SuperFlow (b) Bucket Structures of $\mathbf{M}$ and $\mathbf{A}$

Fig. 2. The framework of SuperFlow and bucket structures of $\mathbf{M}$ and $\mathbf{A}$

[3]We also use f to represent the flow itself when it is clear from the context.

B. Update the Main Table $\mathbf{M}$

When a packet p with flow ID f arrives, a 32-bit fingerprint F is generated from its flow ID, then it is hashed into a bucket $\mathbf{M}_1[idx]$ in $\mathbf{M}_1$ where $idx = h_1(f)$. If $\mathbf{M}_1[idx]$ is empty, we save F in the fingerprint field, set the count field to 1, and export the ID f to the control plane. On the other hand, if the bucket is already occupied by a flow record with the same fingerprint F, the count field is simply increased by 1 [4]. In either case, a proper bucket has been found for p, and the processing of this packet is finished. Otherwise, we encounter a collision, where two flows with different fingerprints are hashed into the same bucket, so we repeat the above process in each sub-table $\mathbf{M}_i$ one by one, using h_i as the corresponding indexing function. If no proper bucket can be found in any of the d sub-tables, we further process this packet in $\mathbf{A}$. In this case, p must have passed through all the d sub-tables of $\mathbf{M}$, and the corresponding d buckets that p has been hashed into constitute a *matching path* of p. We use max and min to represent the maximum and minimum values of the count fields on p's *matching path*, respectively.

C. Update the Ancillary Table $\mathbf{A}$

Each bucket in $\mathbf{A}$ records the most active flow mapped into it recently. Specifically, the digest field records an 8-bit digest generated from its flow ID, the status field accumulates a resistance count that keeps this flow from being evicted by another flow mapped into the same bucket, the count field tries to remember how many packets of the flow have been recently hashed into this bucket, and the snapshot field records the max value (i.e., the maximum count on the aforementioned *matching path*) collected when a flow seizes the bucket. In particular, the status and count fields will be used for the estimation of flow activeness and size. The packet processing in $\mathbf{A}$ contains the following three schemes.

Elastic Collision Resolution. When a packet p of flow f comes in, an 8-bit digest D is generated from its flow ID f, then we use the hash function g to map it to a bucket $\mathbf{A}[idx]$ in $\mathbf{A}$, where $idx = g(f)$. If $\mathbf{A}[idx]$ is an empty bucket, we set both status and count of this bucket to 1, and record D and max in digest and snapshot. If the bucket is already occupied by a flow with the same digest D, with a high probability this existing flow is f, so its status and count are both incremented by 1. In this case,we also promote f back to the main table $\mathbf{M}$ under certain conditions, as will be explained in the *elephant flow promotion* and *inactive flow eviction* procedures below. At last, if we encounter a different digest D, $\mathbf{A}[idx]$ will be updated in a more subtle way, depending on the value of the status field. Specifically, if status > 0, we decrease it by 1, reducing the existing flow's resistance to be evicted. Otherwise, status has reached 0, indicating that the existing flow has exhausted its resistant power and should be evicted by the incoming f, so we update the digest field with D, set status to 1, increment count by 1, and set snapshot to

[4]We neglect the small probability that different flows hashed into the same bucket happen to have the same fingerprint or digest.

max. This scheme, while reducing the impact caused by flow collisions in $\mathbf{A}$, is specifically designed for tracking elephant and active flows.

Elephant Flow Promotion. If we find that an existing flow f in $\mathbf{A}$ is larger than some flow in $\mathbf{M}$, we will promote f back to $\mathbf{M}$. This is accomplished by comparing the count value in $\mathbf{A}[idx]$ with min, which is the minimal count on the *matching path* of p. If the count value of $\mathbf{A}[idx]$ is larger than min, we export the flow record (in $\mathbf{M}$) corresponding to min as well as the flow ID f to the control plane, and substitute that record with f's fingerprint and the count value in $\mathbf{A}[idx]$. In this way, SuperFlow prefers to save room for elephant flows.

Inactive Flow Eviction. If the *elephant flow promotion* scheme fails to be carried out, we check whether a stale elephant flow in the main table $\mathbf{M}$ should be exported. Remember that the snapshot field records the old max value a packet of f saw in $\mathbf{M}$ when f seized $\mathbf{A}[idx]$. When a subsequent packet of f passes through $\mathbf{M}$, sees a new max value, and arrives at $\mathbf{A}[idx]$, very probably the old and the new max values are from the same elephant flow record f' in $\mathbf{M}$. So we know $max-\text{snapshot}$ packets of f' have arrived during this time period. On the other hand, the number of packets in flow f that have arrived during the same period is at least status. So we treat f as a more active one than f' if the status value is greater than $max-\text{snapshot}$.[5] In this case, we export the flow record of f' (i.e., the one corresponding to max in $\mathbf{M}$), as well as the flow ID f, to the control plane, and update the bucket of f' with f's fingerprint and the count value of $\mathbf{A}[idx]$. In this way, SuperFlow prevents stale elephant flows from staying in $\mathbf{M}$ and wasting the precious space.

Since the *elephant flow promotion* and *inactive flow eviction* schemes evict small and elephant flows from $\mathbf{M}$ respectively, there's little chance that a flow will stay in the memory forever but never get exported. Together with the *elastic collision resolution* scheme, they enable SuperFlow to efficiently collect accurate flow records in a continuous manner.

D. Recover Full Flow Records in the Control Plane

The control plane receives raw flow records and flow IDs exported from the data plane. [6]. Since a raw flow record maintained in the main table $\mathbf{M}$ contains a fingerprint instead of a flow ID, we have to recover the flow ID to get a full flow record. To be specific, the control plane maintains a reverse mapping from fingerprints to flow IDs. When receiving an exported flow ID, it computes the corresponding fingerprint and updates the mapping dictionary. In this way, when a raw flow record is received, the corresponding flow ID can be looked up easily, and the full flow record (i.e., flow ID and flow size) can be recovered and saved for other tasks. It should be noted that, after this lookup, the corresponding entry in the dictionary has to be deleted, so that entries cannot accumulate infinitely in the dictionary, and the probability that two flows happen to have the same fingerprints will always be negligible.

[5]The status value is a conservative estimation on f's size, so it helps to prevent radical evictions of elephant flows in $\mathbf{M}$. Besides, in our actual code, we use $\text{status} > max - \text{snapshot} + 1$ to avoid some exceptional conditions.

[6]Although a raw flow record and a flow ID may be exported at the same time, they are actually unrelated, i.e., not of the same flow.

Algorithm 1: SuperFlow

Input: packet p
$min \leftarrow \infty, max \leftarrow 0, f \leftarrow p.\text{flowID}$
generate a 32-bit fingerprint F for f
for $i = 1$ *to* d **do**
  $idx \leftarrow h_i(f)$
  if $\mathbf{M}_i[idx] == (0, 0)$ **then**
    $\mathbf{M}_i[idx] \leftarrow (F, 1)$, export f
    return
  else if $\mathbf{M}_i[idx].\text{fingerprint} == F$ **then**
    $\mathbf{M}_i[idx].\text{count}++$
    return
  else
    if $\mathbf{M}_i[idx].\text{count} < min$ **then**
      $min \leftarrow \mathbf{M}_i[idx].\text{count}$
      $t_{min} \leftarrow i, idx_{min} \leftarrow idx$
    if $\mathbf{M}_i[idx].\text{count} > max$ **then**
      $max \leftarrow \mathbf{M}_i[idx].\text{count}$
      $t_{max} \leftarrow i, idx_{max} \leftarrow idx$
generate an 8-bit digest D for f
$idx \leftarrow g(f)$, $max \leftarrow max\%2^8$
if $\mathbf{A}[idx] == (0, 0, 0, 0)$ **then**
  $\mathbf{A}[idx] \leftarrow (D, 1, 1, max)$ $\Leftarrow p_1$
else if $\mathbf{A}[idx].\text{digest} == D$ **then**
  $\mathbf{A}[idx].\text{status}++$
  $\mathbf{A}[idx].\text{count}++$ $\Leftarrow p_4$
  if $\mathbf{A}[idx].\text{count} > min$ **then**
    $\text{promote\_record}(t_{min}, idx_{min}, f, F, \mathbf{A}[idx].\text{count})$
    $\mathbf{A}[idx] \leftarrow (0, 0, 0, 0)$ $\Leftarrow p_9$
  else if
   $\mathbf{A}[idx].\text{status} > max - \mathbf{A}[idx].\text{snapshot} + 1$
   then
    $\text{promote\_record}(t_{max}, idx_{max}, f, F, \mathbf{A}[idx].\text{count})$
    $\mathbf{A}[idx] \leftarrow (0, 0, 0, 0)$ $\Leftarrow p_7$
else if $\mathbf{A}[idx].\text{status} > 0$ **then**
  $\mathbf{A}[idx].\text{status}--$ $\Leftarrow p_6$
else if $\mathbf{A}[idx].\text{status} == 0$ **then**
  $\mathbf{A}[idx].\{\text{digest}, \text{status}, \text{snapshot}\} \leftarrow \{D, 1, max\}$
  $\mathbf{A}[idx].\text{count}++$ $\Leftarrow p_2$
Function `promote_record` (t, idx, f, F, c) **:**
  export $\mathbf{M}_t[idx]$ and f
  $\mathbf{M}_t[idx] \leftarrow (F, c)$

IV. Evaluation

In this section, we evaluate SuperFlow against five state-of-the-art flow measurement algorithms, i.e., HashFlow [12], HashPipe [13], FlowRadar [15], Elastic [16] and PRECISION [14], and we denote the algorithms by SF, HF, HP, FR, EL, and PR respectively when necessary.

A. Methodology

All algorithms are implemented in C++ on x86 CPU platform for general performance comparison, based on codes publicly available or provided by the authors. To make the comparisons fair, we augment the competitors by using two copies of the sketches that operate in the *periodical export* model alternately. We evaluate the performance of the algorithms using a 40 Gbps backbone link trace from CAIDA [23], a 10 Gbps link trace from a campus network, and another two traces from different ISP access networks. The basic notations and evaluation metrics we use are listed in TABLE I and TABLE II respectively. By default, we allocate 1 MB of memory to the algorithms, and set the export periods of the competitors in such a way that they consume the same export bandwidth as SuperFlow. We will state it explicitly when different settings are adopted.

TABLE I
BASIC NOTATIONS

$\mathcal{F}, \hat{\mathcal{F}}$	The flow set in the traffic and that reported by an algorithm
$n, \hat{n}$	The number of flows in $\mathcal{F}$ and $\hat{\mathcal{F}}$
$s_f, \hat{s}_f$	The actual and estimated size of flow f
w_f	The weight of flow f in $\mathcal{F}$, defined as $w_f = \frac{s_f}{\sum_{f\in\mathcal{F}} s_f}$
T	The threshold for a flow to be classified as a heavy hitter
$\mathcal{H}, \hat{\mathcal{H}}$	The actual and reported set of heavy hitters with respect to T
$\tilde{\mathcal{H}}$	The intersection of $\mathcal{H}$ and $\hat{\mathcal{H}}$ (i.e., $\mathcal{H} \cap \hat{\mathcal{H}}$)
$c, \hat{c}, \tilde{c}$	The number of heavy hitters in $\mathcal{H}, \hat{\mathcal{H}}$ and $\tilde{\mathcal{H}}$ respectively

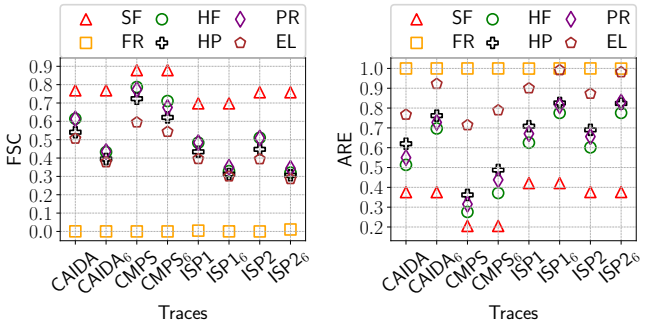
TABLE II
METRICS AND DEFINITIONS

Flow Record Report	
Flow Set Coverage (FSC)	$\frac{\hat{n}}{n}$
Weighted Flow Set Coverage (WFSC)	$\sum_{f\in\hat{\mathcal{F}}} w_f$
Average Relative Error (ARE)	$\frac{1}{n}\sum_{f\in\mathcal{F}} \left\lvert\frac{s_f-\hat{s}_f}{s_f}\right\rvert$
Weighted Relative Error (WRE)	$\frac{1}{n}\sum_{f\in\mathcal{F}} w_f \left\lvert\frac{s_f-\hat{s}_f}{s_f}\right\rvert$
Heavy Hitter Detection	
Recall Rate (RR)	$\tilde{c}/c$
Precision Rate (PR)	$\tilde{c}/\hat{c}$
F1 Score	$\frac{2\times PR\times RR}{PR+RR}$
Average Relative Error (ARE)	$\frac{1}{c}\sum_{f\in\mathcal{H}} \left\lvert\frac{s_f-\hat{s}_f}{s_f}\right\rvert$

B. Application Performance

We first evaluate the performance of SuperFlow and its competitors in flow record report and heavy hitter detection.

Flow Record Report. As shown in Fig. 3, SuperFlow can record more than 69.8% of the total flows, which is up to 136.7% better than those of the best competitors, and the average relative error (ARE) of SuperFlow in flow size estimation is as small as 0.21, which is up to 51.5% smaller than those of the best competitors. One special observation is that FlowRadar can report nearly no flows (FSC$\approx$0) under all cases, because its decoding procedure fails when the number of flows exceeds its capacity. Therefore, we will avoid evaluating the performance of FlowRadar in the following.



(a) Flow Set Coverage (b) Average Relative Error

Fig. 3. Performance in flow record report

Heavy Hitter Detection. In Fig. 4 we present the performance of the algorithms in detecting heavy hitters. For a threshold larger than 40, SuperFlow achieves the F1 Score of 0.99 and the ARE of 0.02, so it is perfect in heavy hitter detection. Even when the threshold is as low as 10, its F1 Score (0.94) and ARE (0.08) are still very attractive, and are at least 11.8% and 78.1% better than its competitors.

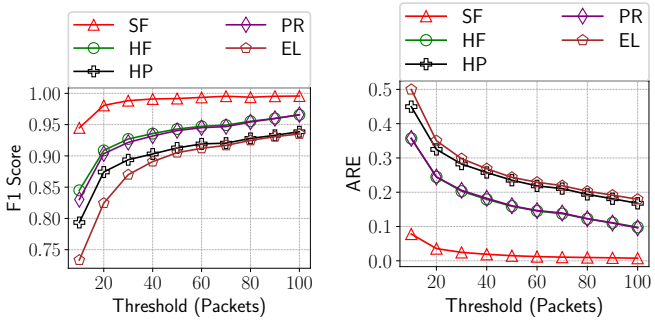


Fig. 4. Performance in heavy hitter detection

C. Memory & Bandwidth Consumption

In this section, we demonstrate that, to achieve the same level of accuracy, SuperFlow needs much less memory and bandwidth than the other algorithms, usingthe ISP2 trace. For a fair comparison, we first let SuperFlow achieve a fixed ARE (i.e., $0.1 \sim 0.5$) in flow size estimation, and compute its memory and bandwidth consumption. Then, to measure the memory usage of a competing algorithm, we adjust the memory and export period used by the algorithm, so that it consumes the same export bandwidth as SuperFlow while achieving this ARE. For bandwidth consumption, we use the

similar approach by letting them use the same memory size. As shown in Fig. 5, to achieve the ARE of 0.1, SuperFlow requires the memory size of 3.77 MB, and results in the bandwidth consumption of 12.92 Mbps. However, to achieve the same level of accuracy, the best competitors require the memory size of 6.11 MB or the bandwidth consumption of 21.58 Mbps.

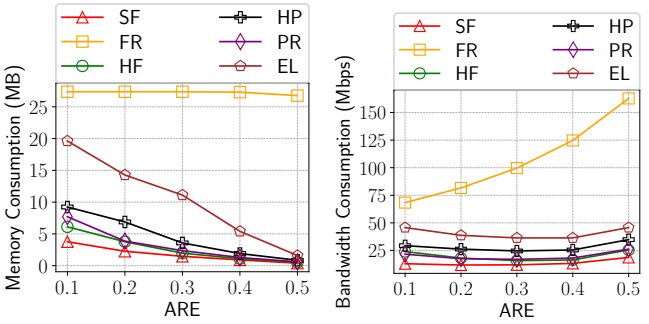


Fig. 5. Memory and bandwidth consumption

V. Conclusion

In this paper, we propose SuperFlow, a highly efficient algorithm to continuously track flow records. With carefully designed data structures and procedures, SuperFlow realizes the schemes of *elastic collision resolution*, *elephant flow promotion* and *inactive flow eviction*, which help to improve measurement accuracy and reduce memory and bandwidth consumption. These merits are demonstrated with comprehensive experiments on various traces.

VI. Acknowledgement

This work is supported by National Key R&D Program of China (No. 2018YFB1800400).

References

[1] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," in *SIGCOMM*, Aug. 2002, pp. 323–336.

[2] B. Claise, "Cisco Systems NetFlow Services Export Version 9," Tech. Rep., Oct. 2004, RFC3954.

[3] B. Claise, B. Trammell, and P. Aitken, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information," Tech. Rep., Sep. 2013, RFC7011.

[4] P. Goyal and A. Goyal, "Comparative Study of Two Most Popular Packet Sniffing Tools-Tcpdump and Wireshark," in *CICN*, 2017, pp. 77–81.

[5] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple Network Management Protocol (SNMP)," Tech. Rep., May 1990, RFC1157.

[6] *Random Sampled NetFlow*, https://www.cisco.com/c/en/us/td/docs/ios/12_2sb/feature/guide/sbrsnf.html.

[7] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a Better NetFlow," in *SIGCOMM*, 2004, pp. 245–256.

[8] R. Jang, D. Min, S. Moon, D. Mohaisen, and D. Nyang, "SketchFlow: Per-Flow Systematic Sampling Using Sketch Saturation Event," in *INFOCOM*, Jul. 2020, pp. 1339–1348.

[9] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," in *SIGCOMM*, 2016, pp. 101–114.

[10] S. Muthukrishnan, "Data Streams: Algorithms and Applications," *Foundations and Trends® in Theoretical Computer Science*, vol. 1, pp. 117–236, 2005.

[11] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications," in *LATIN 2004: Theoretical Informatics*, 2004, pp. 29–38.

[12] Z. Zhao, X. Shi, X. Yin, Z. Wang, and Q. Li, "HashFlow for Better Flow Record Collection," in *ICDCS*, Jul. 2019, pp. 1416–1425.

[13] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane," in *SOSR*, 2017, pp. 164–176.

[14] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient Measurement on Programmable Switches Using Probabilistic Recirculation," in *ICNP*, Sep. 2018, pp. 313–323.

[15] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A Better NetFlow for Data Centers," in *NSDI*, 2016, pp. 311–324.

[16] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic Sketch: Adaptive and Fast Network-wide Measurements," in *SIGCOMM*, Aug. 2018, pp. 561–575.

[17] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," in *NSDI*, 2013, pp. 29–42.

[18] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "SketchVisor: Robust Network Measurement for Software Packet Processing," in *SIGCOMM*, 2017, pp. 113–126.

[19] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant Time Updates in Hierarchical Heavy Hitters," in *SIGCOMM*, 2017, pp. 127–140.

[20] *Barefoot Tofino: World's Fastest P4-Programmable Ethernet Switch ASICs*, https://barefootnetworks.com/.

[21] P. Tune and D. Veitch, "Towards Optimal Sampling for Flow Size Estimation," in *SIGCOMM*, 2008, pp. 243–256.

[22] N. Hohn and D. Veitch, "Inverting Sampled Traffic," in *SIGCOMM*, 2003, pp. 222–233.

[23] *CAIDA UCSD Anonymized Internet Traces Dataset - 2018*, http://www.caida.org/data/passive/passive_dataset.xml.