HashFlow for Better Flow Record Collection

Zongyi Zhao*, Xingang Shi*§, Xia Yin*, Zhiliang Wang*, Qing Li†‡
zhaozong16@mails.tsinghua.edu.cn shixg@cernet.edu.cn
yxia@tsinghua.edu.cn wzl@cernet.edu.cn liq8@sustc.edu.cn
*Tsinghua University, †Southern University of Science and Technology
‡PCL Research Center of Networks and Communications, Peng Cheng Laboratory, Shenzhen
§Corresponding Author

***Abstract*—Collecting flow records is a common practice of network operators and researchers for monitoring, diagnosing and understanding a network. Traditional tools like NetFlow face great challenges when both the speed and the complexity of the network traffic increase. To keep pace up, we propose HashFlow, a tool for more efficient and accurate collection and analysis of flow records. The central idea of HashFlow is to maintain accurate records for elephant flows, but summarized records for mice flows, by applying a novel collision resolution and record promotion strategy to hash tables. We have implemented HashFlow as well as several latest flow measurement algorithms in a P4 software switch, and use traces from different operational networks to evaluate the algorithms. In these experiments, for various types of traffic analysis applications, HashFlow consistently demonstrates a clearly better performance against its state-of-the-art competitors. For example, using a small memory of 1 MB, HashFlow can accurately record around 55K flows, which is often 12.5% higher than the others. For estimating the sizes of 50K flows, HashFlow achieves a relative error of around 11.6%, while the estimation error of the best competitor is 42.9% higher. It detects 96.1% of the heavy hitters out of 250K flows with a size estimation error of 5.6%, which is 11.3% and 73.7% better than the best competitor respectively. At last, we show these merits of HashFlow come with almost no degradation of throughput.**

I. Introduction

NetFlow[1] is a widely used tool in network measurement and analysis. It records traffic statistics in the form of flow records, where each record contains important information about a flow, for example, its source and destination IP addresses, start and end timestamps, type of services, application ports, input and output ports, as well as the volume of packets or bytes, etc.

A challenge in implementing NetFlow like tools is to keep up with the ultra high speed of network traffic, especially on high-bandwidth backbone links. For example, assuming an average packet size of 700 bytes, and a 40 Gbps link, the time budget for processing one packet is only around 50 nano-seconds[2][3][4]. In an extreme case where packets of 40 bytes arrive at a speed of 100 Gbps, the time budget will be only a few nano-seconds. NetFlow also faces the high diversity of the traffic, where hundreds of thousands, even millions of concurrent flows appear in a measurement epoch. This pose stringent pressure on the scarce high-speed memories, such as on-chip SRAM with $1 \sim 10$ nano-seconds access delay[5][6].

One straightforward solution is to use sampling [7], where out of several packets, only one of them gets processed and used to update the flow records. However, sampling reduces processing overhead at the cost of fewer packets or flows being recorded, thus less accurate statistics that can be estimated. To remedy this, very enhanced sampling algorithms [8][9][10] have been proposed and tailored for specific measurement requirement, and their impact analyzed[11][12]. Another direction of solution is to use sketch (also referred as data streaming algorithms) [13][14][15], where a succinct data structure is designed and can be updated very efficiently. However, these sophisticated data structures and algorithms generally can only be used in limited scenarios, but not for the wide range of applications that the original NetFlow can support.

Towards accelerating flow record maintenance and achieving better statistics estimation, recently a few algorithms that make enhancement to a naive hash table and integrate sketches have been proposed, including OpenSketch[16], UnivMon[17], FlowRadar[5], HashPipe[18], and ElasticSketch[19], etc. Both constant bound of the worst case delay and efficient utilization of memory are achieved, making them good candidates for general measurement applications in high-speed environment.

Following these efforts, we propose HashFlow, which makes a further step in squeezing memory consumption. The central idea of HashFlow is to maintain accurate records for elephant flows (i.e., flows with many packets), as well as summarized records for mice flows (i,e., flows with a few packets), by applying novel strategies of collision resolution and record promotion to hash tables. The collision resolution part eliminates collisions that may mix up packets from different flows, keeps a flow from being evicted until another flow with larger size collides with it, while fully utilizing the memory space by filling up nearly all hash table buckets. On the other hand, the record promotion part allows the flows to grow in the summarized set, and bounces a flow back from the summarized set to the accurate set and replaces the original one which has smaller size when this flow becomes large enough. The performance bound can be analyzed with a probabilistic model, and with this strategy, HashFlow achieves a better utilization of space, and also more accurate flow records, without bringing extra complexity.

We have implemented HashFlow, as well as several latest flow measurement algorithms mentioned above, including FlowRadar, HashPipe and ElasticSketch, in a P4-

programmable [20] software switch[21]. To illustrate the implementability of HashFlow, we further implement it in a commodity P4 switch [22] which has the type of Wedge 100BF-32X[23]. We then use traces from different operational networks to evaluate their effectiveness. In these experiments, for various types of traffic analysis applications, HashFlow demonstrates a consistently better performance against its state-of-the-art competitors. For example, using a small memory of 1 MB, HashFlow can accurately record around 55K flows, which is often 12.5% higher than the others. For estimating the sizes of 50K flows, HashFlow achieves a relative error of around 11.6%, while the estimation error of the best competitor is 42.9% higher. It detects 96.1% of the heavy hitters out of 250K flows with a size estimation error of 5.6%, which is 11.3% and 73.7% better than the best competitor respectively. At last, we show that these merits of HashFlow come with negligible degradation of throughput.

The remainder of the paper is organized as follows. We introduce our motivation and central ideas in designing HashFlow in Section II. We present the algorithm details, as well as the theoretical analysis in Section III, and then present the implementation details in hardware P4 switch in Section IV. Using real traffic traces, we analyze the parameters of HashFlow and compare it against other algorithms in Section V. Finally we conclude the paper in Section VI.

II. Background and Basic Ideas

Formally, we define a flow record as a key-value pair $(key, count)$, where key is the ID of the flow, and $count$ is the number of packets belonging to this flow. A simple example is like this: the flow ID contains the source and destination IP addresses, and packets with exactly the same source and destination belong to the same flow. The definition is general, since the flow ID can also be a subnet prefix, a transport layer port, or even a keyword embedded in application data. A naive method to maintain flow records is to save them in a hash table, but multiple flows may be hashed to the same bucket in the table. Mechanisms to resolve collisions in hash tables include classic ones like separate chaining and linear probing, and more sophisticated ones like Cuckoo hashing [24]. However, in the worst case, they need unbounded time for insertion or lookup, thus are not adequate for our purpose.

Before presenting HashFlow, we briefly review several recently proposed algorithms, i.e., HashPipe[18], ElasticSketch[19] and FlowRadar[5], and try to point out some minor defects in the algorithms. We will assume that the readers are familiar with the algorithms.

HashPipe[18] uses a series of independent hash tables (each with a different hash function). The first table is used to effectively accommodate new flows and evict the existing flows when collision occurs, otherwise new flows will have little chance to stay if large flows accumulate in the table. But on the other hand, this strategy frequently splits one flow record into multiple records that are stored in different hash tables, each with a partial count, since an existing flow may be evicted but new packets of this flow may still arrive later. This effect makes the utilization of memory less efficient, and makes the packet count less accurate.

In ElasticSketch[19], due to the collisions and eviction strategy employed, a flow record may also be split into multiple records, and the packet counter is not accurate. The count-min sketch is introduced to help the flow size estimation. However, since the count-min sketch itself may not be accurate, the estimation accuracy is limited, which is especially true if the sketch is occupied by too many flows.

In FlowRadar[5], flow information is encoded into a flow set, and then we can recover (some) flow IDs and the corresponding packet counts during the post processing phase. However, the chances that such decoding succeeds drop abruptly if the table is heavily loaded and there are not enough flows that don't collide with any other ones.

With these in mind, we then analyze a few tradeoffs and design choices for time and space efficient collection of flow records.

1) *With limited memory, discard flows when necessary.* Pouring too many flows into a hash table or sketch will cause frequent collision, and either increase the processing overhead, or decrease the accuracy of the information that can be retrieved. For example, FlowRadar faces severe degradation in its decoding capability when the number of flows exceeds its capacity (this effect and the turning point can be clearly seen in our evaluation, for example, Fig. 7 for flow set monitoring and Fig. 9 for flow size estimation). In most situations, network traffic is skewed such that only a small portion of elephant flows contain a large number of packets. For example, in one campus trace we use, 7.7% of the flows contribute more than 85% of the packets. It will be better to discard mice flows with few packets than elephant ones, since the latter have a greater impact on most applications, such as heavy hitter detection, traffic engineering and billing. It is often enough to maintain summarized information for the mice flows.

2) *A flow should be consistently stored in one record.* Both HashPipe and ElasticSketch may split one flow into multiple fragments stored in different tables. This not only wastes memory, but also causes the packet count less accurate, which in turn affects the eviction strategies. By storing a flow in a consistent record, we can achieve both better memory utilization and higher accuracy.

3) *If better memory utilization can be achieved by trading off a little efficiency, have a try.* This is particularly worth to do when network equipment is becoming more "software-defined", where their functionalities can be "programmed", and the additional operations can be easily paralleled or pipelined. By the nature of the ball and urn model [25] of hash tables, there will be a few empty buckets of a hash table that have never been used, and the utilization will be improved by feeding more flows into the hash table or hashing a flow multiple times to find a proper bucket. Both HashPipe and ElasticSketch propose to split the hash table into multiple small tables and use multiple hash functions to improve the utilization, but in different ways. Our collision resolution

strategy is more similar to that of HashPipe than ElasticSketch. Later, we will show our strategy can make an effective use of the table buckets.

III. Algorithm Details

In this section we explain how HashFlow works in detail, and present some theoretical analysis results, which are based on a probabilistic model.

A. Data Structures and Algorithm

The data structure of HashFlow is composed of a main table ($\mathbf{M}$) and an ancillary table ($\mathbf{A}$), each being an array of buckets, and each bucket (also called as cell) can store a flow record in the form of *(key, count)*, as mentioned in Section II. In the main table $\mathbf{M}$, flow ID will be used as *key*, while in the ancillary table $\mathbf{A}$, a digest of flow ID will be used as *key* to save the memory space. We have a set of $d+1$ independent hash functions, i.e., $h_1, h_2, \cdots, h_d$, and g, where d is a positive integer (we call d the *depth* of $\mathbf{M}$, and typically $d = 3$). Each hash function $h_i (1 \leq i \leq d)$ randomly maps a flow ID to one bucket in $\mathbf{M}$, while g maps the ID to one bucket in $\mathbf{A}$. A digest can be generated from the hashing result of the flow ID with any h_i. When a packet arrives, HashFlow updates $\mathbf{M}$ and $\mathbf{A}$ with the following two strategies, as shown in Algorithm 1.

1) **Collision Resolution.** When a packet p arrives, we first map it into the bucket indexed at $idx = h_1(p.\text{flow\_id})$ in the main table $\mathbf{M}$. If $\mathbf{M}[idx]$ is empty, we just put the flow ID and a count of 1 in the bucket (line 5$\sim$ 6). If the bucket is already occupied by packets of this flow earlier, we just simply increment the count by 1 (line 7 $\sim$ 8). In either case, we have found a proper bucket for the packet, and the process finishes. If neither of the two cases happen, then a collision occurs, and we repeat the same process but with $h_2, h_3, \cdots, h_d$ one by one, until a proper bucket is found for the packet. This is a simple collision resolution procedure. Unlike HashPipe and ElasticSketch, it does not evict existing flow record from the main table, thus prevents a record from being split into multiple records.

If collision cannot be resolved in the main table, then we try to record it in the ancillary table $\mathbf{A}$. Here the action is more intrusive, as an existing flow will be replaced (discarded) if it collides with the new arrival (line 16 $\sim$ 19).

2) **Record Promotion.** If, in the ancillary table $\mathbf{A}$, p succeeds to find the right bucket to reside, then it updates the packet count field of the record. Moreover, if the corresponding flow record keeps growing and the packet count becomes large enough, then we will promote the record by re-inserting it into the main table, thus prevents large flows from being discarded. To implement this strategy, we keep in mind the sentinel flow record that has the smallest packet count among those records that collide with p in the collision resolution procedure (line 9 $\sim$ 11). When a flow record in the ancillary table should be promoted, it will replace this sentinel we have kept in mind (line 22 $\sim$ 23). We note that, instead of the flow ID, a shorter digest is used as keys in the ancillary table to reduce memory consumption. This may mix flows up, but with a small chance. When doing record promotion, we simply extract the flow ID from the current packet.

Algorithm 1 Update Algorithm of HashFlow on arrival of p

```
1:  //Collision Resolution
2:  flowID ← p.flow_id, min ← ∞, pos ← −1
3:  for i = 1 to d do
4:      idx ← h_i(flowID)
5:      if M[idx].key == NULL then
6:          M[idx] ← (flowID, 1) return
7:      else if M[idx].key == flowID then
8:          Increment M[idx].count by 1 return
9:      else if M[idx].count < min then
10:         min ← M[idx].count
11:         pos ← idx
12:     end if
13: end for
14: idx ← g(flowID)
15: digest ← h_1(flowID)%(2^digest width)
16: if A[idx].count == 0 or A[idx].key ≠ digest then
17:     A[idx] ← (digest, 1)
18: else if A[idx].count < min then
19:     Increment A[idx].count by 1
20: else
21:     //Record Promotion
22:     M[pos].key ← flowID
23:     M[pos].count ← A[idx].count + 1
24: end if
```

We use a simple example with $d = 2$ to illustrate the algorithm, as depicted in Fig. 1. When a packet of flow f_1 arrives, h_1 maps it into an empty bucket indexed at $h_1(f_1)$, so the record becomes $(f_1, 1)$. When a packet of flow f_2 arrives, h_1 maps it into a bucket indexed at $h_1(f_2)$, where the record $(f_2, 5)$ has the same key, and the counter is simply incremented. When a packet of flow f_3 arrives, it collides with the record $(f_4, 4)$ in the bucket indexed at $h_1(f_3)$. Then we try to resolve collision with h_2, but again, the packet collides with the record $(f_5, 10)$ in the bucket at $h_2(f_3)$. So we have to use g to find a place in the ancillary table for the packet. Sadly, it collides again with the record $(f_6, 8)$, and we let it replace the existing one. The last packet is from flow f_7, and it goes through a similar process to that of f_3. The difference is that, at last, this packet finds its corresponding flow record of $(f_7, 7)$ in the ancillary table, and after being updated the record appears to have greater packet count (i.e., 8) than the sentinel flow which has the packet count of 7. So we promote $(f_7, 8)$ by inserting it back into the main table, evicting the sentinel one.

In Algorithm 1, we use multiple independent hash functions $h_i (i = 1, 2, \cdots, d)$ in a single main table $\mathbf{M}$. Another choice is to use multiple small hash tables $\mathbf{M}_i$, each of which corresponds to an independent hash function h_i. Algorithm 1 can be modified straightforwardly: update the i-th small table $\mathbf{M}_i$ instead of $\mathbf{M}$ (line 5 $\sim$ 12), remember which small

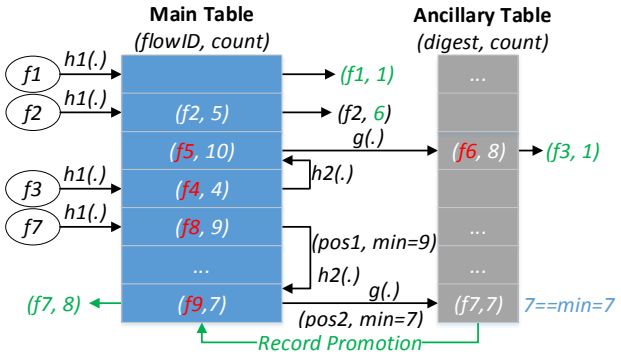


Fig. 1. An example of HashFlow

table the sentinel record resides in (line 10 $\sim$ 11), and evict the sentinel record in the right small table (line 22 $\sim$ 23) to do record promotion. In addition, we introduce a weight $\alpha (0 < \alpha < 1)$, such that the number of buckets in $\mathbf{M}_{i+1}$ is α times that in $\mathbf{M}_i$.

Readers may notice that HashPipe uses a similar scheme of pipelined tables, but there are a few important differences. First, HashFlow uses pipelined tables together with an ancillary table. Second, the update strategy of these pipelined tables is different from that of HashPipe. Third, our collision resolution procedure on the main table can be analyzed theoretically, based on which we can achieve a concrete performance guarantee on the number of accurate flow records that HashFlow can maintain.

B. Analysis

In the section, we propose a probabilistic framework that models the utilization of the main table. We first analyze the case where a multi-hash table is used for the main table, then the case where pipelined tables are used. In either case, we assume that there are m distinct flows fed into the main table, which has n buckets in total, and uses d hash functions.

Multi-hash table. First, consider the case when $d = 1$, where the analysis follows a classic ball and urn problem[25]. After inserting $m_1 = m$ flows randomly into n buckets, the probability that a given bucket is empty is

$$p_1 = (1 - \frac{1}{n})^{m_1} \approx e^{-\frac{m_1}{n}},$$

and the utilization of the table is $u_1 = 1 - p_1 = 1 - e^{-\frac{m_1}{n}}$. Since each bucket can contain only one flow record due to our collision resolution strategy, the number of flows that fail to be cached in $\mathbf{M}$ after this round is $m_1 - n \times (1 - p_1)$.

Now consider the case of $d = 2$. Essentially, a flow tries another bucket with h_2 if it finds out that the first bucket it tries has already been occupied. Since we don't care which exact flow is stored in the table, we slightly change the update process to the following one. We take two rounds. In the first round, we feed all the m_1 flows into the table with h_1, exactly the same as $d = 1$. In the second round, we feed all the remaining flows that have not been kept in $\mathbf{M}$ into the table again, but this time with h_2. Assume $\mathbf{M}$ is empty before the second round starts, then after the $m_2 = m_1 - n \times (1 - p_1)$ flows left by the first round have been inserted in the second round, a bucket will be empty with probability $e^{-\frac{m_2}{n}}$. However, $\mathbf{M}$ is actually not empty before the second round, and at that time a bucket in it is empty with probability p_1. Since h_1 and h_2 are independent, we know after the second round, the probability that a bucket is still empty becomes $p_2 \approx p_1 \times e^{-\frac{m_2}{n}}$, and the number of flows that have not been inserted into $\mathbf{M}$ will be $m_3 = m_1 - n \times (1 - p_2)$. The utilization of $\mathbf{M}$ now becomes $u_2 = 1 - p_2$.

The analysis for the slightly changed process can be extended to cases when $d > 2$. In the k-th round, m_k flows are fed into a hash table with a new hash function h_k, where there are already $n \times (1 - p_{k-1})$ buckets being occupied in the previous rounds. Then after the k-th round, the probability that a bucket is empty is

$$\begin{aligned} p_k &\approx p_{k-1} \times e^{-\frac{m_k}{n}} \\ &= p_{k-1} \times e^{-\frac{m_1 - n \times (1 - p_{k-1})}{n}} \\ &= p_{k-1} \times e^{1 - \frac{m_1}{n} - p_{k-1}} \\ &= p_{k-1} \times e^{1 - \frac{m}{n} - p_{k-1}} \end{aligned} \quad (1)$$

for $k \geq 2$. With Equation (1), for any given d, m, and n, we can recursively compute the probability p_d that a bucket is empty in the hash table after d rounds. Then the utilization of the hash table will be $1 - p_d$. We note that there is a slight difference between this model and our multi-hash table, as will be shown later.

Pipelined tables. Let n_k be the number of buckets in the k-th table $\mathbf{M}_k$ such that $n_{k+1} = \alpha \times n_k$, where α is the pipeline weight. We perform a similar modification to our collision resolution procedure with pipelined tables, such that in the k-th round, all packets goes though the k-th table before they are fed into the $k+1$-th table in the $k+1$-th round. We use the same notations p_k, m_k and u_k as those in the first model. Since $\sum_{k=1}^{d} n_k = \sum_{k=1}^{d} (\alpha^{k-1} \times n_1) = \frac{1-\alpha^d}{1-\alpha} \times n_1$, we get $n_1 = \frac{1-\alpha}{1-\alpha^d} \times n$, and $n_k = \alpha^{k-1} \times \frac{1-\alpha}{1-\alpha^d} \times n$.

The first round works exactly the same as that in the previous model, so we get $p_1 = (1 - \frac{1}{n_1})^{m_1} \approx e^{-\frac{m_1}{n_1}}$, $u_1 = 1 - p_1$, and $m_2 = m_1 - n_1 \times (1 - p_1)$.

For the k-th round, we know m_k flows are to be fed into the table $\mathbf{M}_k$ with n_k buckets, so we get $p_k \approx e^{-\frac{m_k}{n_k}}$, and the number of flows left after this round is

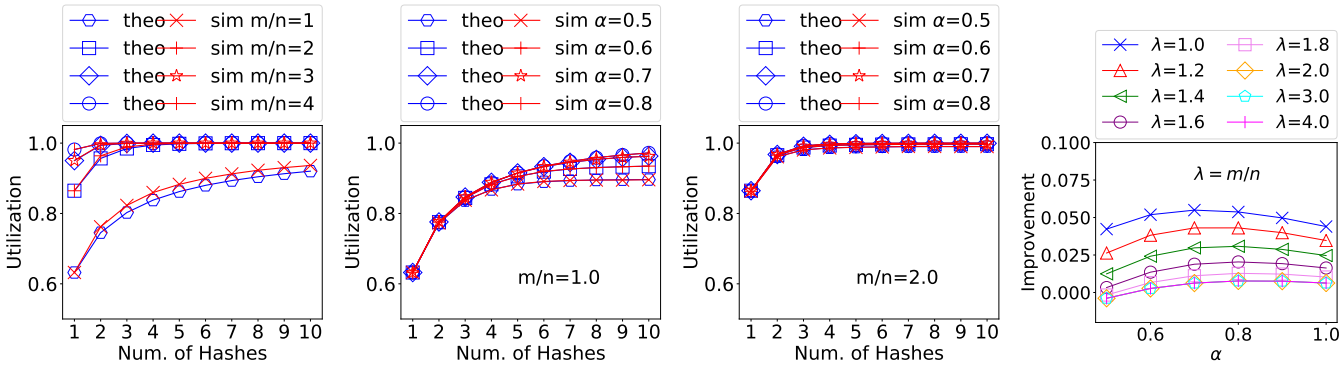
$$m_{k+1} = m_k - n_k \times (1 - p_k). \quad (2)$$

Dividing both sides of Equation (2) by n_{k+1}, we get

$$\begin{aligned} \frac{m_{k+1}}{n_{k+1}} &= \frac{n_k}{n_{k+1}} \times \frac{m_k - n_k \times (1 - p_k)}{n_k} \\ &= \alpha^{-1} \times \left(\frac{m_k}{n_k} - 1 + p_k \right), \end{aligned}$$

which is just

$$-\ln p_{k+1} = \alpha^{-1} \times (-\ln p_k - 1 + p_k). \quad (3)$$



(a) Multi-hash Table (b) Pipelined Tables (c) Pipelined Tables (d) Improvement on Utilization

Fig. 2. Utilization of the multi-hash table and the pipelined tables

From Equation (3), we finally get

$$p_{k+1} = (p_k)^{\frac{1}{\alpha}} \times e^{\frac{1-p_k}{\alpha}}. \quad (4)$$

With Equation (4), for any given d, m, and n, we can recursively compute the probability $p_k (1 \leq k \leq d)$ that a bucket is empty in the k-th hash table. Then the utilization of the pipelined tables will be

$$\frac{\sum_{k=1}^{d}(n_k \times (1-p_k))}{\sum_{k=1}^{d} n_k} = 1 - \frac{1-\alpha}{1-\alpha^d} \times \sum_{k=1}^{d}(\alpha^{k-1} \times p_k). \quad (5)$$

Now we will show how accurate the models are. In Fig. 2(a), we use n =100K buckets, with different depth d from 1 to 10, and vary the traffic load m/n from 1 to 4. As can be seen there, when $m/n \geq 2$, the multi-hash table model provides nearly perfect predictions.

In Fig. 2(b) and Fig. 2(c), we use a similar setting as above, with n =100K and d from 1 to 10, but we vary the pipeline weight α between 0.5 to 0.8. This time the model and the simulation results match quite well, since arranging the packet arrivals in rounds, as we have done in the model, actually does not affect the final probability (we omit the proof due to space limitations).

With these two models, we can predict the utilization of our main table, as long as the traffic load m/n is known. We can see that more hash functions will improve the utilization. In the case of $m/n = 1$, the utilization increases from 63% to 83% when d is increased from 1 to 3, and from 83% to 92% when d is increased from 3 to 10. As more hash functions require more hash operations and memory accesses in the worst case, 3 hash functions seems to be a sweet spot, and we use $d = 3$ by default in our evaluations.

Fig. 2(d) shows, when $d = 3$, pipelined tables always improves the utilization upon multi-hash table, regardless of the traffic load. As shown there, when $\alpha = 0.7$ and $m/n = 1$, up to 5.5% more utilization can be achieved. Our evaluation will adopt the pipelined scheme, where $\alpha = 0.7$ seems to be the best choice.

IV. Implementation in P4 Hardware Switch

As stated before, we have implemented HashFlow in bmv2, the software P4 switch, as well as in a hardware P4 switch which has the type of Wedge 100BF-32X[23]. The code can be found at [26]. Although both versions of the algorithm are implemented using $P4_{14}$, the grammar checking of the hardware switch is stricter than that of the software switch, and the implementation is heavily limited by the resource restrictions of the hardware. In this section, we will discuss the hardware implementation of HashFlow in detail.

The P4 program will be compiled into a pipeline, which consists of multiple stages. Multiple small match-action tables can be packed into a stage, and a large table may span multiple stages. The tables within the same stage can be executed in parallel, while the stages can only be executed serially and tables that are dependent on each other must be distributed among different stages. The compiler will analyze the dependency relationship of the tables and arrange the tables within the stages automatically. The amount of processing within a single stage is upper bounded, so the processing time of a single stage is limited. Moreover, to upper bound the processing delay within a single P4 switch, the number of stages that a switch can support is limited. Our switch can support 12 stages at most. In our implementation, we identify a flow with the typical 5 tuples and store the flow records into 6 register arrays, i.e., one register array for source/destination IP address, protocol, source/destination port and packet count respectively. A register array can only be accessed using stateful ALUs, which can execute a simple program atomically. As an action of P4 cannot access more than one register array (through stateful ALUs), we implement 6 match-action tables to access a flow record. Since accesses of the IP addresses, protocol and ports are independent, and the operations imposed upon packet count are determined by the result from accessing the flow ID, the tables corresponding to the arrays of IP addresses, protocol and ports can be packed into a single stage, while the table corresponding to the packet count has to be put into another stage. We define an *iteration* to be all the processing operations corresponding to

a pipelined table or the ancillary table, so HashFlow contains $d+1$ iterations. In our implementation, the first iteration needs 2 stages, and each of the following iterations needs 4 stages, so HashFlow needs $4\times d+2$ stages and only $d \leq 2$ is allowed in our P4 switch. To support more pipelined tables, more resources or advanced techniques to refine the implementation are needed.

To allow the pipelining of the packets, multiple tables sharing the same register array in a pipeline can only access the resource exclusively, which means that only one table can access the resource when processing a packet. In Algorithm 1, we may have to visit the hash table d times when doing collision resolution, violating the access restriction. We can address the problem by splitting the hash table into d small tables, so only the pipelined-tables scheme of main table is feasible in P4 hardware switch. However, when computing the index from a flow ID, the size of the index space must be the power of 2, so the table arrangement stated in Section III-B, i.e., decreasing the size of the pipelined tables in a factor of 0.7, is infeasible. We simply set the size of each table to be the same in our hardware implementation.

Another challenge in implementing HashFlow is that record promotion requires to revisit one of the pipelined tables, even if we have visited every table when doing collision resolution, thus violating the access restriction. Our solution is to resubmit the current packet and process it again when doing record promotion. By marking some metadata, we will be able to evict an existing flow record and set up a new record for this packet. Since more packets than that feed into the switch are processed when the resubmit primitive is used, the throughput of the switch will be degraded. In Section V-D we will evaluate the loss of throughput caused by resubmit primitive.

V. Evaluation

A. Methodology

We have implemented HashFlow, as well as several latest algorithms that try to improve NetFlow, including FlowRadar[5], HashPipe[18] and ElasticSketch[19], in bmv2 [21]. The code for FlowRadar and ElasticSketch are rewritten based on their published code, while HashPipe is implemented based on the algorithm in the published paper. As we have implemented HashFlow in a P4 hardware switch[23], our implementation is strongly constrained due to the resource limitation, so we still use the software switch to evaluate the performance of the algorithms.

We use 4 traces from different environment to evaluate these algorithms' performance, one from a 40 Gbps backbone link provided by CAIDA [27], one from a 10 Gbps link in a campus network, and the other two from different ISP access networks. Some flow level statistics are summarized in Table I, where we can see that the traffic in different traces differs greatly. However, by plotting the cumulative flow size distribution in Fig. 3, we find they all exhibit a similar skewness pattern, that most flows are mice flows with a small number of packets, while most of the traffic are from a small

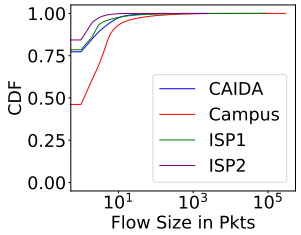


Fig. 3. Flow size distribution of the traces used for evaluation

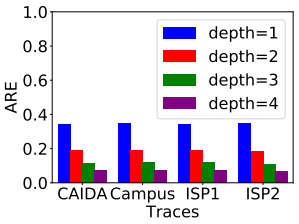


Fig. 4. Flow size estimation under different pipeline depth

number of elephant flows [28]. The only exception is the ISP2 trace, which is 1:5000 sampled from an access link and more than 99% of the flows in it have less than 5 packets (the CDF also reveals this). When evaluating the algorithms, for each trial, we select a constant number of flows from each trace, and feed the packets of these flows to each algorithm. Particularly, since the traces provided by CAIDA and ISP1 are in the granularity of packets while that provided by Campus and ISP2 are in the granularity of flows, the arrival order of packets is determined by the original trace for CAIDA and ISP1 traces, while we generate the packets sequence from the flows randomly for Campus and ISP2 traces.

TABLE I
Traces used for evaluation

Trace	Date	max flow size	ave. flow size
CAIDA	2018/03/15	92385 pkts	13.6 pkts
Campus	2014/02/07	289877 pkts	15.1 pkts
ISP1	2009/04/10	33003 pkts	7.5 pkts
ISP2	2015/12/31	2441 pkts	1.3 pkts

Suppose n flows are processed by each algorithm, where a flow is defined by the typical 5 tuples, i.e., source/destination IP addresses, protocol, and source/destination ports. The measurement applications we use to evaluate the algorithms and traffic statistics we use as performance metrics are as follows.

- *Flow Record Report.* An algorithm reports the flow records it maintains, where each record is of the form (flow ID, packet count). The performance metric we use is *Flow Set Coverage (FSC)* defined as

$$FSC = \frac{\text{num. of flow records with complete flow IDs}}{n}.$$

- *Flow Size Estimation.* Given a flow ID, an algorithm estimates the number of packets belonging to this flow. If no result can be reported, we use 0 as the default value. The performance metric we use is *Average Relative Error (ARE)* defined as

$$ARE = \frac{1}{n}\sum \left| \frac{\text{estimated size of flow } i}{\text{real size of flow } i} - 1 \right|.$$

- *Heavy Hitter Detection.* An algorithm reports heavy hitters, which are flows with more than T packets, and T is an adjustable parameter. Let c_1 be the number of heavy hitters reported by an algorithm, c_2 the number

of real heavy hitters, and among the reported c_1 heavy hitters c of them are real. The performance metric we use is *F1 Score* defined as

$$\text{F1 Score} = \frac{2 \cdot PR \cdot RR}{PR + RR},$$

where $PR = \frac{c}{c_1}$ and $RR = \frac{c}{c_2}$. We also use *ARE* of the size estimation of the heavy hitters as another metric.

- *Cardinality Estimation.* An algorithm estimates the number of flows. The performance metric we use is *Relative Error (RE)* defined as

$$RE = \left| \frac{\text{estimated number of flows}}{n} - 1 \right|.$$

Notice that, linear counting[29] is used by ElasticSketch and HashFlow to estimate the number of flows in the count-min sketch and ancillary table respectively.

Following recommendations in the corresponding papers, we set the parameters of these algorithms as follows.

- HashPipe: We use 4 sub-tables of equal size.
- ElasticSketch: We adopt the hardware version, where 3 sub-tables are used in its heavy part. The light part uses a count-min sketch of one array, and the two parts use the same number of cells.
- FlowRadar: We use 4 hash functions for its bloom filter and 3 hash functions for its counting table. The number of cells in the bloom filter is $40\times$ of that in the counting table.
- HashFlow: We use the same number of cells in the main table and the ancillary table. The main table consists of three small hash tables with the weight α of 0.7. Each digest and counter in the ancillary table costs 8 bits.

We let these algorithms use the same amount of memory in all the experiments. For each flow record, we use a flow ID of 104 bits and a counter of 32 bits, So 1 MB memory approximately corresponds to 60K flow records. In the worst case, HashFlow, HashPipe and ElasticSketch (hardware version) will compute 4 hash results to access the corresponding cells, while FlowRadar always needs to compute 7 hash results. To save space, we use the acronyms presented in Table II to denote the algorithms in figures when necessary.

TABLE II
ACRONYMS FOR ALGORITHMS

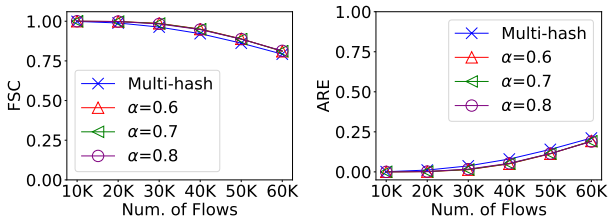
Acronym	Algorithm	Acronym	Algorithm
HF	HashFlow	HP	HashPipe
ES	ElasticSketch	FR	FlowRadar

B. Optimizing the Main Table and Ancillary Table

We first demonstrate the performance of the main table with the collision resolution strategy, under different settings and parameters, i.e, using a multi-hash table, or using pipelined tables with different weights.

In Fig. 5, there are 3 pipelined tables of which the weight is 0.6, 0.7 and 0.8 respectively and a multi-hash table. The traces are from the campus network, and we increase the number of flows from 10K to 60K. It can be seen that the pipelined tables with a weight around $\alpha = 0.7$ achieves the best result. Compared with the multi-hash table, it improves the *FSC* by 3.1%, and reduces the *ARE* by 37.3% respectively. This confirms our theoretical analysis on α in Section III-B. In the experiments thereafter, we will use a default weight of 0.7.

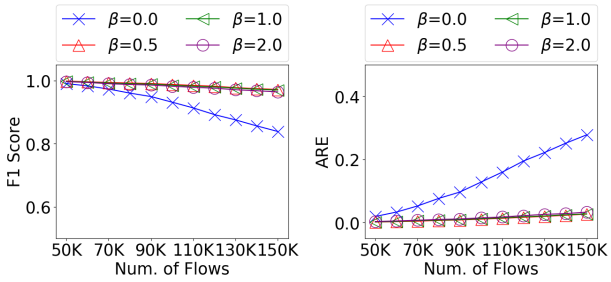
In Fig. 4, we set the number of flow to 50K, and the depth of the main table is set to 1, 2, 3 and 4. It can be seen that increasing d from 1 to 3 reduces the *ARE* by around 3 times (i.e., from 0.34 to 0.12), while increasing d from 3 to 4 will have only a minor improvement (i.e., from 0.12 to 0.075). In the experiments thereafter, we will use a default depth of 3.



(a) Flow Record Report (b) Flow Size Estimation

Fig. 5. Comparing multi-hash table with pipelined tables.

To evaluate the influence that the size of ancillary table has upon HashFlow, we define a parameter β and let $n_2 = n_1 \times \beta$, where n_1 and n_2 are number of buckets in the main table and ancillary table respectively. There is not an ancillary table at all when $\beta = 0.0$. We feed a number of flows extracted from CAIDA file, and calculate the F1 Score and ARE (Average Relative Error) of heavy hitter detection where a heavy hitter is a flow with no less than 10 packets. Fig. 6 shows that the ancillary table is crucial for the performance of HashFlow and normally $\beta = 0.5$ is good enough. However, when attacks such as DDoS occur, the accumulation of elephant flows in ancillary table will be interrupted frequently. To be robust when facing such attacks, in the following experiments we set β to 1.0 by default.



(a) F1 Score (b) ARE

Fig. 6. F1 Score and Average Relative Error (ARE) of heavy hitter detection when the size of ancillary table varies.

C. Application Performance

In this section, we evaluate the performance of HashFlow against HashPipe, ElasticSketch and FlowRadar, for typical measurement applications as described in Section V-A.

Fig. 7 shows that HashFlow nearly always performs better than the others in the sense of *Flow Set Coverage (FSC)*. For example, for a total of 250K flows, it can successfully report around 55K flows, nearly making a full use of its main table. Its *FSC* is more than 20% higher than ElasticSketch in all traces, and is that higher than HashPipe in the Campus Network trace. The only exception when HashFlow loses is that, for a very small number of flows (the left up corner in the figures), FlowRadar has the highest coverage. This is because FlowRadar can successfully decode nearly all flow records when only a few flows arrive. But its performance drops significantly soon after the flow count goes beyond a certain point, since after that, too many flows mixed up, and the decoding often fails.

Fig. 8 shows the results of estimating the total number of flows, where in most of the time, HashFlow, ElasticSketch and FlowRadar achieve a similar level of accuracy. Among them, FlowRadar works slightly better since it uses a bloom filter to count flows, which is not sensitive to flow sizes, while HashFlow and ElasticSketch are slightly affected by the flow size distribution due to their assumption on the existence of elephant and mice flows. This is particularly true in the ISP2 trace, where nearly all flows contain less than 5 packets. HashPipe always performs badly since it does not use any advanced cardinality estimation technique to compensate for the flows it drops.

Fig. 9 shows that HashFlow often achieves a much lower estimation error than its competitors when estimating the flow sizes. For example, when there are 100K flows, the relative estimation error of HashFlow is around 0.4, while the error of the others is more than 0.6 (50% higher) in most cases. FlowRadar performs very badly when there are more than 40K flows, while the accuracy of HashPipe is not very stable.

At last, we show whether they can accurately detect heavy hitters. We feed 250K flows to each algorithm, and calculate the *F1 Score* of their detection accuracy as well as the ARE of the estimated sizes of the detected heavy hitters. The results are depicted in Fig. 10 and Fig. 11, respectively. Apparently, FlowRadar is not a good candidate under such heavy load. HashPipe is designed specifically for detecting heavy hitters, but our HashFlow still outperforms it in nearly all cases, for both metrics. Not considering the extreme case of the ISP2 trace where most flows are typically very small, for a wide range of thresholds, HashFlow achieves a *F1 Score* of 1 (accurately detecting all heavy hitters) when the scores of HashPipe and ElasticSketch are around 0.9 and $0.4 \sim 0.7$, respectively. On the other hand, when HashFlow makes nearly perfect size estimation of the heavy hitters, the *ARE* of HashPipe and ElasticSketch are around $0.15 \sim 0.2$ and $0.2 \sim 0.25$, respectively. Even with a very small threshold used in the ISP2 trace, HashFlow clearly outperforms the others.

D. Throughput

We test the throughput of these algorithms with bmv2, on a PC with Intel(R) Core(TM) i5-4680K CPU@3.40GHz, where each CPU core owns a 6144 KB cache. We use *isolcpus* to isolate the cores to prevent context switches. Bmv2 achieves around 20 Kpps forwarding speed, and the throughput after loading the algorithms are depicted in Fig. 12(a). To obtain a better understanding, we also record the average number of hash operations, as well as memory accesses, for each algorithm. The results in Fig. 12(b) and Fig. 12(c) indicate that, HashFlow will perform comparably to HashPipe and ElasticSketch, and much better than FlowRadar, in the sense of the number of memory accesses and hash operations.

In our hardware implementation, the only risk degrading the throughput of the switch is that some packets will be resubmitted (or recirculated), so the switch will have to process more packets than those transmitted through it. To evaluate the possible loss of throughput, we calculate the *Resubmit Rate* defined as

$$\text{Resubmit Rate} = \frac{n_2}{n_1}.$$

where n_1 is the number of packets arriving at the switch, and n_2 is the number of packets that are resubmitted.

As shown in Fig. 12(d), when there are 100K flows fed into the switch, the *Resubmit Rate* is normally less than 2%, and it is no more than 5.2% in the trace provided by ISP2, which is far less skewed than the real network traffic. So HashFlow will achieve very good performance and the loss of throughput is negligible.

VI. Conclusion

We propose HashFlow for efficient collection of flow records, which is useful for a wide range of measurement and analysis applications. The collision resolution and record promotion strategy is of central importance to HashFlow's accuracy and efficiency. We analyze the performance bound of HashFlow based on a probabilistic model, and implement it in a software switch as well as a hardware switch. The evaluation results based on real traces from different networks show that, HashFlow consistently achieves a clearly better performance in nearly all cases. In the future, we plan to study how to make it adaptive to more accurate measurement of mice flows and network wide measurement.

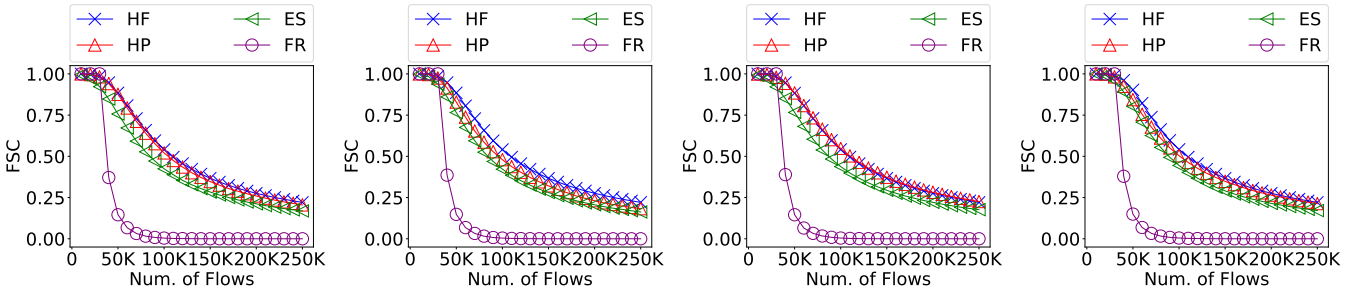
VII. Acknowledgement

We thank the anonymous ICDCS reviewers for their valuable comments. This work is supported by the National Natural Science Foundation of China(Grant No. 61702315) and PCL Future Regional Network Facilities for Large-scale Experiments and Applications (PCL2018KP001).

References

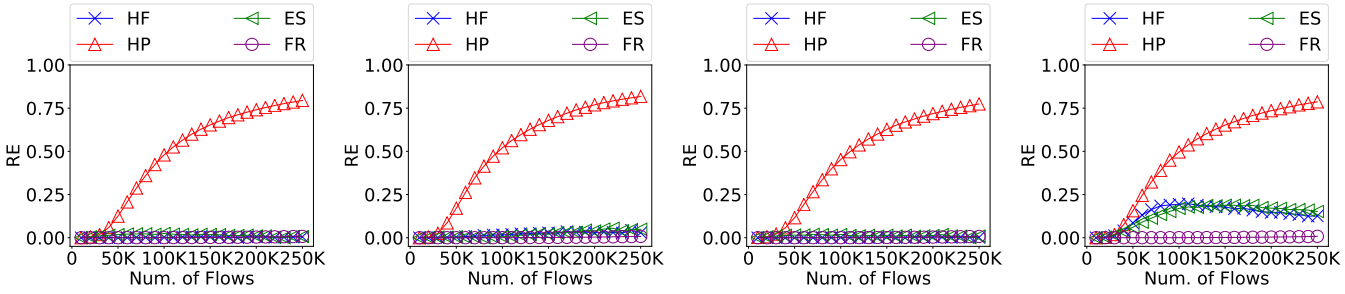
[1] B. Claise, "Cisco Systems NetFlow Services Export Version 9," 2004. [Online]. Available: https://tools.ietf.org/html/rfc3954

[2] H. Zhang, X. Shi, Y. Guo, Z. Wang, and X. Yin, "More Load, More Differentiation - Let More Flows Finish Before Deadline in Data Center Networks," *Computer Networks*, vol. 127, pp. 352–367, Nov. 2017.



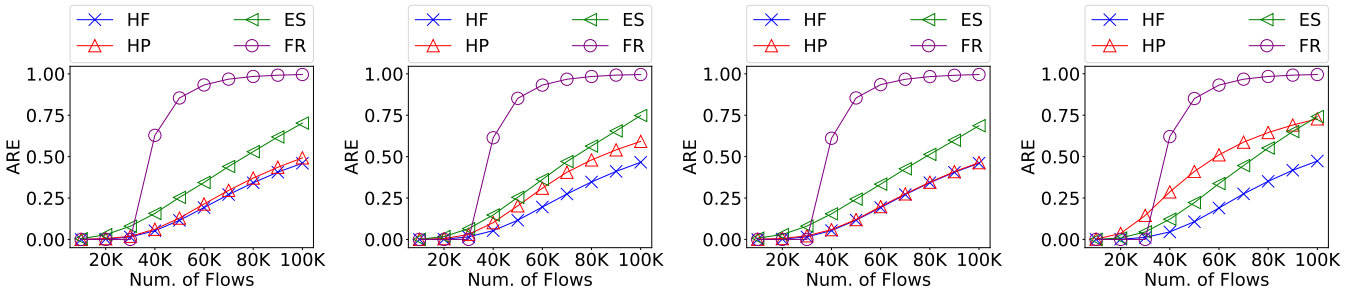
(a) CAIDA (b) Campus Network (c) ISP1 (d) ISP2

Fig. 7. *Flow Set Coverage (FSC)* for *Flow Record Report*



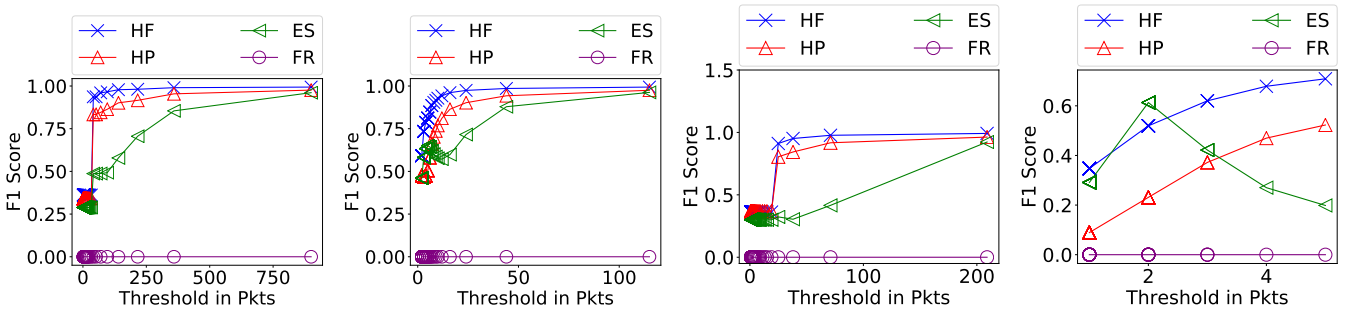
(a) CAIDA (b) Campus Network (c) ISP1 (d) ISP2 Trace

Fig. 8. *Relative Error (RE)* for *Flow Cardinality Estimation*



(a) CAIDA Trace (b) Campus Network Trace (c) HGC Trace (d) Telecom Trace

Fig. 9. *Average Relative Error (ARE)* for *Flow Size Estimation*

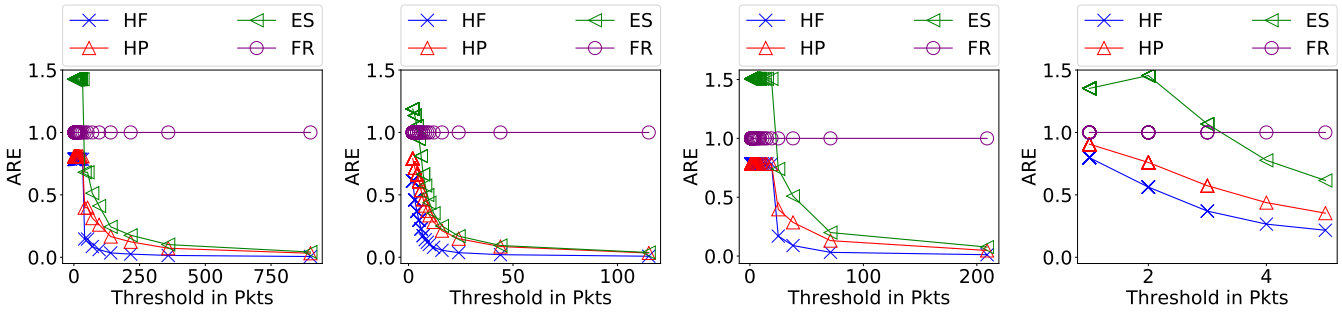


(a) CAIDA (b) Campus Network (c) ISP1 (d) ISP2

Fig. 10. *F1 Score* for *Heavy Hitter Detection*

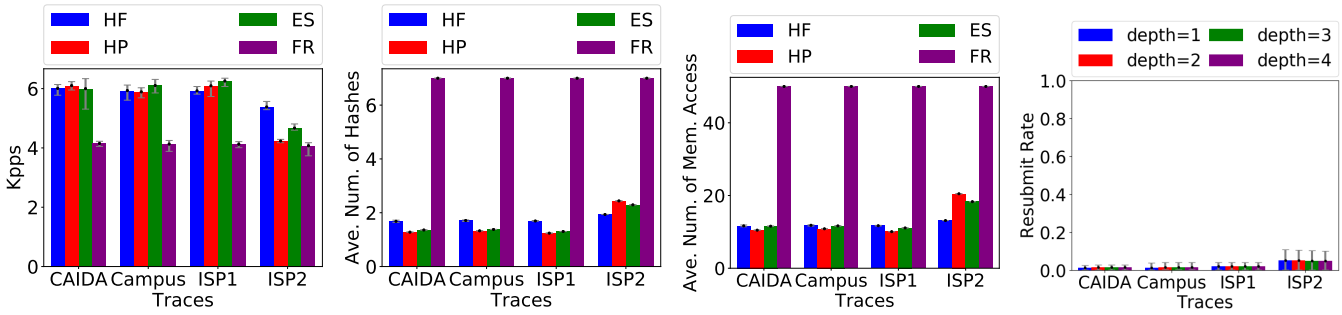
[3] H. Zhang, X. Shi, X. Yin, F. Ren, and Z. Wang, "More Load, More Differentiation - A Design Principle for Deadline-Aware Congestion Control," in *2015 IEEE Conference on Computer Communications*, Apr. 2015, pp. 127–135.

[4] Z. Wang, H. Zhang, X. Shi, H. Geng, Y. Li, X. Yin, J. Liu, and Q. Wu, "Efficient Scheduling of Weighted Coflows in Data Centers," *IEEE*



(a) CAIDA (b) Campus Network (c) ISP1 (d) ISP2

Fig. 11. *Average Relative Error(ARE)* for *Heavy Hitter Detection*



(a) Throughput (b) Number of hash operations (c) Number of memory accesses (d) Resubmit Rate

Fig. 12. Throughput, Hash Operation, Memory Access and Increase in Processing

Transactions on Parallel and Distributed Systems, pp. 1–1, 2019.

[5] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A Better NetFlow for Data Centers," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, 2016, pp. 311–324.

[6] "Access Time of DRAM and SRAM." [Online]. Available: https://www.webopedia.com/TERM/A/access_time.html

[7] "Sampled NetFlow." [Online]. Available: https://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html

[8] N. Hohn and D. Veitch, "Inverting Sampled Traffic," in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, 2003, pp. 222–233.

[9] N. Duffield, C. Lund, and M. Thorup, "Estimating Flow Distributions from Sampled Flow Statistics," *IEEE/ACM Trans. Netw.*, vol. 13, no. 5, pp. 933–946, Oct. 2005.

[10] P. Tune and D. Veitch, "Towards Optimal Sampling for Flow Size Estimation," in *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, 2008, pp. 243–256.

[11] N. Duffield, "Sampling for Passive Internet Measurement: A Review," *Statist. Sci.*, vol. 19, no. 3, pp. 472–498, 08 2004.

[12] V. Carela-Español, P. Barlet-Ros, A. Cabellos-Aparicio, and J. Solé-Pareta, "Analysis of the Impact of Sampling on NetFlow Traffic Classification," *Comput. Netw.*, vol. 55, no. 5, pp. 1083–1099, Apr. 2011.

[13] S. Muthukrishnan, "Data Streams: Algorithms and Applications," *Found. Trends Theor. Comput. Sci.*, vol. 1, no. 2, pp. 117–236, Aug. 2005.

[14] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 113–126.

[15] M. Chen, S. Chen, and Z. Cai, "Counter Tree: A Scalable Counter Architecture for Per-Flow Traffic Measurement," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1249–1262, 2017.

[16] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013, pp. 29–42.

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

[18] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 164–176.

[19] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2018, p. 14.

[20] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[21] "bmv2: P4 Software Switch," 2018. [Online]. Available: https://github.com/p4lang/behavioral-model

[22] "Barefoot Tofino: World's Fastest P4-Programmable Ethernet Switch ASICs." [Online]. Available: https://barefootnetworks.com/

[23] "Edgecore Networks." [Online]. Available: https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=335

[24] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[25] N. L. Johnson and S. Kotz, *Urn Models and Their Application :An Approach to Modern Discrete Probability Theory*. New York: John Wiley and Sons, 1977.

[26] Z. Zhao, "HashFlow Implemented in P4," Dec. 2018.

[27] "CAIDA UCSD Anonymized Internet Traces Dataset - 2018." [Online]. Available: http://www.caida.org/data/passive/passive_dataset.xml

[28] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, 2010, pp. 267–280.

[29] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A Linear-Time Probabilistic Counting Algorithm for Database Applications," *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 208–229, 1990.