Efficient and Accurate Flow Record Collection With HashFlow

Zongyi Zhao, Xingang Shi, *Member, IEEE*, Zhiliang Wang, *Member, IEEE*, Qing Li, Han Zhang, *Member, IEEE*, and Xia Yin, *Senior Member, IEEE*

Abstract—Traditional tools like NetFlow face great challenges as both the speed and the complexity of the network traffic increase. To keep the pace up, we propose HashFlow for more efficient and accurate collection of flow records. HashFlow keeps large flows in its main flow table and uses an ancillary table to summarize the other flows when the main table is full. With our *flow collision resolution* and *flow record promotion* schemes, a flow in the ancillary table is promoted back to the main flow table with a guaranteed probability when it becomes large enough. These operations can be performed highly efficiently, so HashFlow can keep up with ultra-high traffic speed. We implement HashFlow in a Tofino switch, and using traces from different operational networks, we compare its performance against some state-of-the-art flow measurement algorithms. Our experiments show that, for various types of traffic analysis applications, HashFlow consistently demonstrates clearly better performance than its competitors. For example, the performance of HashFlow in flow size estimation, flow size distribution estimation and heavy hitter detection is up to 21, 60 and 35 percent better than those of the best competitors respectively, and these merits of HashFlow come with almost no degradation of throughput.

Index Terms—Network measurement, flow record measurement, sketch

1 Introduction

Flow record collection tools (e.g., NetFlow [1] and IPFIX [2]) maintain information such as source and destination IP addresses, transport protocol, source and destination ports, start and end timestamps, and the volume of packets and bytes, for flows. These tools are widely used in network measurement and analysis, because, unlike SNMP counters [3] and packet capture tools [4], they achieve a good balance between the accuracy and scalability of the maintained information.

The challenge in flow record collection is to keep up with the ultra high speed of network traffic while dealing with the enormous number of concurrent flows. For example, on a 100 Gbps link of the backbone network, hundreds of thousands, even millions, of concurrent flows may exist, and, with the average packet size of 700 bytes, the time budget for processing one packet is around 56 nanoseconds [5], [6], [7]. Therefore, huge-capacity and high-speed memory is required for flow record collection. However, the high-speed memory such as SRAM in the commodity routers/switches is scarce (e.g., with the capacity of a few hundreds of Mbits in our Tofino switch), while the abundant DRAM is too slow (e.g., with the access time of 50~150 nanoseconds [8]) for packet processing.

One straightforward solution is to use sampling [9], where only a portion of the packets are used to update the flow record table. Some advanced sampling algorithms [10], [11], [12] have been proposed and tailored for specific measurement tasks, with their performance analyzed [13], [14]. However, sampling results in fewer packets and flows being recorded, undermining the accuracy of the maintained statistics. On the other hand, sketches, which are highly succinct data structures, usually maintain only counters [15], [16], [17] or a few special flows (e.g., the top-K largest flows) [18], so cannot deal with flow record collection nicely. The last category of solutions [19], [20], [21], [22], [23] typically maintain a record for each flow in SRAM, but, by assuming that the information of large flows is more valuable, evict small flows when there isn't enough space. However, the difficulty lies in that, the flow table must be updated highly efficiently so that the delay does not exceed a constant bound, and it's not easy to determine which flow is to be evicted.

We follow the last direction and propose HashFlow to make a further step in improving the efficiency and accuracy of flow record collection. The central idea of HashFlow is to keep large flows (which we also call heavy flows) in its main flow table as intact as possible, while summarizing the other flows in an ancillary table when they are bypassed by the main table due to hash collisions. However, with our

- *Zongyi Zhao is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: zhaozong16@mails.tsinghua.edu.cn.*
- *Xingang Shi, Zhiliang Wang, and Han Zhang are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology (BNRIST), Beijing 100084, China. E-mail: {shixg, wzl}@cernet.edu.cn, zhhan@tsinghua.edu.cn.*
- *Qing Li is with the Peng Cheng Laboratory, Shenzhen 518066, China, and also with the Southern University of Science and Technology, Shenzhen 518055, China. E-mail: liq@pcl.ac.cn.*
- *Xia Yin is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology (BNRist), Beijing 100084, China. E-mail: yxia@tsinghua.edu.cn.*

Manuscript received 13 Dec. 2020; revised 12 July 2021; accepted 13 July 2021. Date of publication 26 July 2021; date of current version 15 Oct. 2021.
(Corresponding author: Xingang Shi.)
Recommended for acceptance by P. Bangalore.
Digital Object Identifier no. 10.1109/TPDS.2021.3099442

1045-9219 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

carefully designed *flow collision resolution* and *flow record promotion* schemes, a flow in the ancillary table also gets a chance to be promoted back to the main flow table when it becomes large later, and all the operations can be performed highly efficiently. In particular, the effectiveness of the promotion scheme can be theoretically analyzed with a probabilistic model. With these designs, HashFlow brings no extra complexity so that it can keep up with ultra-high traffic speed, and achieves better coverage and accuracy in flow record collection than its state-of-the-art competitors.

We have implemented HashFlow in a Tofino switch [24], which supports data plane programming with the P4 [25] language. Then we use traces from different operational networks to evaluate its performance under typical metrics related to flow record analysis. Our experiments show that HashFlow demonstrates consistently better accuracy than its state-of-the-art competitors in nearly all cases. For example, using a small memory of 1 MB, for a CAIDA trace with 250K flows, HashFlow estimates the sizes of its recorded flows with an average relative error of 0.12, and estimates the flow size distribution with a relative error of 0.08, which are 21 and 60 percent better than those of the best competitor respectively. Moreover, under 500K flows, HashFlow detects 81 percent of the heavy hitters (which we define as flows containing at least 10 packets) with a relative size estimation error of 0.33, which are 19 and 35 percent better than those of the best competitors respectively. At last, we show that these merits of HashFlow come with negligible potential degradation of throughput.

Our contributions are summarized as follows:

- We design HashFlow, a novel algorithm for accurate flow record collection in high-speed networks. HashFlow uses a main table and an auxiliary table to complement each other, and uses *flow collision resolution* and *flow record promotion* to efficiently record flows, especially large flows.
- We implement HashFlow in a Tofino switch and achieve the line-speed packet processing on multiple 100 Gbps ports.
- We conduct extensive experiments using traces from several operational networks and demonstrate that HashFlow has clearly better performance than the state-of-the-art competitors.

The remainder of the paper is organized as follows. We first review some of the works in network measurement in Section 2, then introduce the rationale behind our design choices of HashFlow in Section 3. We present the algorithm details of HashFlow in Section 4, and give the corresponding theoretical analysis in Section 5. Then we describe the implementation details of HashFlow as well as the other algorithms used in our performance evaluation in Section 6. After that, using both real and synthetic traffic traces, we compare HashFlow against its state-of-the-art competitors in Section 7. Finally, we conclude the paper in Section 8.

2 Related Work

In this section, we briefly review several state-of-the-art algorithms in network measurement to prepare the readers for this field.

The most widely adopted method of network measurement is packet sampling, where, given a sampling rate p, a packet is captured with the probability p. Afterwards, the size of a flow is recovered by multiplying the number of captured packets of the flow by $1/p$. However, packet sampling suffers from flow size bias since the heavy flows containing more packets are captured with a higher probability than the light flows. Moreover, it's impossible to obtain the accurate size of a flow since two flows with different sizes appear the same if the same number of packets are captured for them. Although many advanced sampling-based algorithms [10], [11], [12], [26] have been proposed to mitigate the drawbacks, the loss of information is inevitable.

Another popular trend in network measurement is to design various sketches, which record traffic information using a compact data structure and fixed computations. One widely used sketch is count-min (CM) sketch [15], which consists of d rows of counters where each row contains r counters and has an independent hash function associated with it. When a packet arrives, it is mapped to a counter, and increment the counter by 1, in each row of the sketch. Finally, given a flow, we get the counters corresponding to the flow ID in each row and take the minimum one as the estimated size of the flow. Although the update strategy of CM sketch is simple and efficient, it causes a huge positive bias in flow size estimation, especially for the light flows. Conservative-update (CU) sketch [16] and Count sketch [17] adopt the same data structure as CM sketch, and the query procedure of CU sketch is the same as that of CM sketch. However, when a packet arrives, instead of incrementing each counter this packet is mapped to, CU sketch only increments the smallest counter by 1, then, taking the updated counter value in mind, sets each of the other counters to the larger one of this updated value and its original value. Count sketch takes a completely different approach for update and query, as it has two hash functions h_i and g_i associated with each row of the data structure. Both functions take a flow ID as the input, but h_i outputs the index of the counter this flow ID is mapped to, while g_i outputs 1 or -1. Therefore, when a packet arrives, Count sketch increments each counter this packet is mapped to by the output of the respective function g_i. To obtain the size of a flow, Count sketch multiplies each of the counters this flow is mapped to by the output of g_i, then takes the median of the new values as the estimate.

In our experience, the sketches stated above don't have satisfying performance in flow size estimation, as shown in Fig. 10, as they rely heavily on approximation. More importantly, they cannot generate flow records by themselves as they don't maintain flow IDs, which seriously constrains their usage in network measurement. Therefore, many algorithms maintaining flow IDs explicitly have been proposed. In the following, we will illustrate some of the most influential falling in this category.

In FlowRadar [21], when a packet arrives, its information is encoded into a flow set, then flow records are recovered from the flow set during the post-processing stage. However, the chances that such decoding succeeds drop abruptly if the table is overloaded, rendering the flow set unusable, which is the price of maintaining information for every packet indiscriminately with limited memory.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

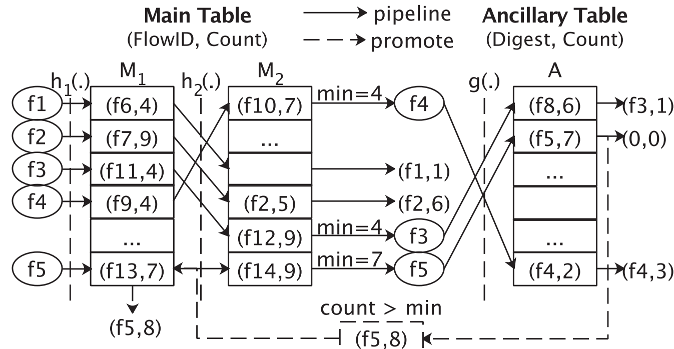


Fig. 1. A few examples of HashFlow.

Unlike FlowRadar, many algorithms propose to track the heavy flows preferentially, as the heavy flows are usually more important for higher applications than the light ones. A well-known algorithm of this type is Space-Saving (SS) [18], which maintains an ordered array of counters, with a flow ID attached to each counter. When a packet arrives, the counter with the same flow ID as the packet is incremented by 1, or a counter is allocated to this flow ID if such a counter doesn't exist. However, if neither a counter with the desired flow ID nor an empty counter is available, the smallest counter is located, its value is incremented by 1, and its flow ID is replaced by that of this packet. While the error of SS in flow size estimation is theoretically bounded, it has poor performance in practice. Moreover, locating a counter with the same flow ID as a given packet is time consuming, thus it's challenging, if not impossible, to implement SS in the programmable switches like P4 switch.

As the popularity of the programmable switches grows, many algorithms, such as HashPipe [22], PRECISION [27] and Elastic [23], which track the heavy flows preferentially and can be readily implemented in P4 platform have been proposed. The algorithms have the similar data structures, i.e., several hash tables, each fitting into some stages of the pipeline of the P4 switch. When a packet arrives, it traverses the hash tables sequentially, and simple updates are conducted on the hash tables. We argue that the algorithms are far from perfect, as HashPipe and Elastic tend to split one flow record into multiple records that are stored in different locations, each with a partial count, which makes the memory utilization less efficient, and PRECISION adopts the approximation skill when creating a flow record, which undermines the accuracy of flow size estimation. Inspired by the algorithms, in this paper we propose HashFlow, which overcomes the drawbacks of the algorithms and achieves much better performance.

3 Motivation and Design Choices

In this paper we abstract a flow record as a key-value pair (flowID, count), where the typical 5-tuple (i.e., source and destination IP addresses, source and destination ports, and transport protocol) [28] is taken as the flowID by default, and the count field can be either the packet count or the byte volume. In practice information such as start and end timestamps, packet count, and byte volume are usually maintained altogether, [2] so that many statistics such as mean packet size and mean inter-arrival time of packets can be derived for each flow record.

Due to traffic skewness, we often face an uneven flow size distribution. For example, a 1-minute CAIDA trace [29] contains 1.3×10^6 active flows but 87 percent of the packets are from the top 10^5 heavy flows, while on a backbone link of a campus network, we find 4.8×10^6 active flows in 1 minute, where 83 percent of the packets are from the top 4×10^5 heavy flows. With a limited memory budget, it is usually better to maintain records for the heavy flows preferentially, and discard light flows when the memory is insufficient, since the heavy flows usually have a greater impact on most applications such as heavy hitter detection, traffic engineering and billing.

To implement this strategy, when a new flow is to be stored but the memory is insufficient, a natural solution is to replace the smallest flow record stored in the memory. While Space-Saving [18] and similar data structures [30], [31] are perfect in picking the smallest flow record, the time complexity of locating and updating an existing flow record in the data structures when a new packet arrives is not $O(1)$, thus they are insufficient for our purpose. For example, Space-Saving [18] stores the counters in a linked list and n flow IDs are attached to the corresponding counters. Although the flow ID(s) for the smallest counter can be accessed with $O(1)$ time, in the worst case it may take $O(n)$ time to locate the counter given a desired flow ID. A good alternative to this, which we adopted in this paper, is to maintain the flow records in multiple hash tables of which each associated with an independent hash function. The new flow is mapped into a bucket in each hash table, and the smallest one of the records residing in the buckets is replaced. Theorem 1 of Section 5 shows that, using this multi-hashing method, the replaced flow record is very close to the smallest one in the memory.

Another choice we make in designing HashFlow is to ensure that a flow never gets split into multiple sub-records and occupies more than one bucket in the hash tables, since high-speed SRAM in network devices is a scarce and precious resource, and the sub-records are more likely to be discarded because of their relatively small packet counts and the heavy-flows-first strategy. This design helps to achieve both better memory utilization and higher accuracy. On the contrary, some existing algorithms are prone to splitting large flows. For example, in our experiments, up to 13 percent of the flows tracked by HashPipe [22] and Elastic [23] are split into at least 2 sub-records in the memory.

4 Algorithm

In this section, we will present the detailed data structure of HashFlow and explain how the different components of HashFlow interact with each other.

The data structure of HashFlow is composed of a main table $\mathbf{M}$ and an ancillary table $\mathbf{A}$, and $\mathbf{M}$ further consists of d sub-tables $\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_d$, where $d = 3$ typically. Each of $\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_d$ and $\mathbf{A}$ is a hash table without bucket chaining, where each bucket can store a flow record in the form of (key, count). The count field can be either the packet count or the volume of bytes, and flowID is used as the key in $\mathbf{M}$, while an 8-bit digest of flowID is used as key in $\mathbf{A}$ to save

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

space. Besides, we have $d+1$ independent hash functions $h_1, h_2, \ldots, h_d$, and g, where h_i $(i = 1, 2, \ldots, d)$ maps a flowID to one bucket in $\mathbf{M}_i$, and g maps the flowID to one bucket in $\mathbf{A}$. The digest can be generated by truncating some $h_i(\mathsf{flowID})$.

When a packet p arrives, HashFlow updates $\mathbf{M}$ and $\mathbf{A}$ in the following two stages, as illustrated in Algorithm 1, where $\Delta(p)$ returns 1 if the packet count is considered, or the size of p if the byte volume is considered.

Stage I. First, we map packet p into the bucket indexed by $\mathrm{idx} = h_1(p.\mathsf{flowID})$ in $\mathbf{M}_1$. If $\mathbf{M}_1[\mathrm{idx}]$ is empty, we initialize the bucket by putting the flowID and $\Delta(p)$ into the bucket. If the bucket is already occupied by the flow that p belongs to, we simply increment the count field by $\Delta(p)$. In either case, a proper bucket for the packet is found, so the process finishes. Otherwise, a collision occurs, so we repeat the same process but with $h_2, \ldots, h_d$ on $\mathbf{M}_2, \ldots, \mathbf{M}_d$ one by one, until a proper bucket is found for the packet. This is a simple *flow collision resolution* procedure similar to d-left hashing, but we deliberately avoid bucket chaining. It also differs from existing algorithms such as cuckoo hashing [32], HashPipe [22] and Elastic [23] by not evicting any existing flow record, thus preventing any flow record from being split into multiple sub-records. In case that no proper bucket is found in $\mathbf{M}$ to store $p.\mathsf{flowID}$, p will be sent to stage II. Besides, along the path p travels in $\mathbf{M}$, we save a sentinel flow record, which is the one with the smallest count among the d records that have collided with p.

Stage II. Packet p enters this stage only if a flow record corresponding to p doesn't exist in $\mathbf{M}$ and an empty bucket is not found there. In this stage, p is mapped into the bucket $\mathbf{A}[\mathrm{idx}]$ where $\mathrm{idx} = g(p.\mathsf{flowID})$. Similar to that in stage I, if $\mathbf{A}[\mathrm{idx}]$ is empty, we store the flow into this bucket but record only the digest of $p.\mathsf{flowID}$, instead of the full flowID. When a collision occurs (i.e., the digest of $p.\mathsf{flowID}$ is different from the key field of this bucket), we make a different *flow collision resolution* by replacing the existing flow record with the digest of $p.\mathsf{flowID}$ and $\Delta(p)$. Otherwise, the key value is the same as the digest of $p.\mathsf{flowID}$, so we update the flow record by adding $\Delta(p)$ to its count. If this updated count is larger than that of the sentinel flow record we have tracked in stage I, the *flow record promotion* procedure replacing the sentinel one in $\mathbf{M}$ with $(p.\mathsf{flowID}, \mathbf{A}[\mathrm{idx}].\mathsf{count})$ is triggered.

Notice that after a flow record in $\mathbf{A}[\mathrm{idx}]$ is promoted to $\mathbf{M}$, nothing will happen to the bucket $\mathbf{A}[\mathrm{idx}]$. This won't have any negative influence on HashFlow, since once the flow record is recorded in $\mathbf{M}$, all the following packets with its flow ID will be recorded in $\mathbf{M}$ and never reach $\mathbf{A}$. Meanwhile, when another packet with a different flow ID is mapped to bucket $\mathbf{A}[\mathrm{idx}]$, this bucket will be rewritten immediately with high probability.

In HashFlow, we use multi-hashing method to decide where to store a flow record in the $\mathbf{M}$ table. This may be similar to the multi-level hash functions widely used by large-scale object storage systems such as Hadoop [33] and Lustre [34], but there is a cirtical difference in how they resolve collisions. A typical collision resolution strategy adopted by the multi-level hash functions is chaining, i.e., storing those objects hashed to the same bucket in a list. Therefore, the table storing objects of the storage systems can be dynamically expanded, and objects will be stored as long as there is free space. However, the space used in HashFlow is fixed, so each bucket can be occupied by at most one flow record. If a collision occurs, HashFlow will use its specifically designed *flow collision resolution* scheme to decide which one should be kept, and it's possible that a flow record is dropped.

No cooperation from other switches is required for HashFlow to do the flow record collection, but the collected flow records have to be exported periodically to a separate data server, so that HashFlow can refresh $\mathbf{M}$ and $\mathbf{A}$ to accommodate new flow records. Packets are always processed at line speed if HashFlow is implemented in a Tofino switch, but throughput degradation is likely to happen if HashFlow is implemented in other platforms, especially the CPU based ones such as Open vSwitch. Load balancing [35] techniques may help HashFlow to reduce such degradations, and we leave the study of that to our future work.

Algorithm 1. HashFlow

```
Input: packet p
//Stage I
flowID ← p.flowID, min ← ∞, pos ← −1, t ← −1
//flow collision resolution
for i = 1 to d do
  idx ← h_i(flowID)
  if M_i[idx] == NULL then
    M_i[idx] ← (flowID, Δ(p)) return ⇐ f_1
  else if M_i[idx].key == flowID then
    M_i[idx].count ← M_i[idx].count + Δ(p) ⇐ f_2
    return
  else if M_i[idx].count < min then
    //save the sentinel flow record
    min ← M_i[idx].count
    t ← i, pos ← idx
//Stage II
idx ← g(flowID)
digest ← h_1(flowID)%(2^(digest width))
if A[idx] == NULL or A[idx].key ≠ digest then
  A[idx] ← (digest, Δ(p)) ⇐ f_3
else
  A[idx].count ← A[idx].count + Δ(p) ⇐ f_4
  if A[idx].count > min then
    //flow record promotion ⇐ f_5
    M_t[pos].key ← flowID
    M_t[pos].count ← A[idx].count
```

We use a few examples to illustrate the above algorithm, and plot the workflow in Fig. 1. For ease of understanding, we use $d = 2$ and the packet count as the count field of a flow record. We also make corresponding annotations in Algorithm 1. When a packet of flow f_1 arrives, it is mapped into a bucket of $\mathbf{M}_1$ indexed by $h_1(f_1)$, where it collides with the flow record $(f_6, 4)$. Then it is mapped into an empty bucket of $\mathbf{M}_2$ indexed by $h_2(f_1)$, and the content of the bucket becomes $(f_1, 1)$. When a packet of flow f_2 arrives, it collides with flow record $(f_7, 9)$ in $\mathbf{M}_1$, but the bucket it is mapped into in $\mathbf{M}_2$ happens to have been occupied by f_2, so the count field of the bucket is simply incremented by 1. When a packet of flow f_3 arrives, it collides with the flow records in $\mathbf{M}_1$ and $\mathbf{M}_2$, so it is mapped to $\mathbf{A}$. However, it collides with the flow record $(f_8, 6)$ there as well, so it replaces

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

the flow record with $(f_3, 1)$. Collisions also occur when a packet of f_4 is mapped to $\mathbf{M}_1$ and $\mathbf{M}_2$, so it further goes to a bucket of $\mathbf{A}$ indexed by $g(f_4)$. Then the count field is incremented by 1 since the key field of the bucket is the same as the digest of f_4. A packet of f_5 goes through the same procedure and the count of flow record $(f_5, 7)$ in $\mathbf{A}$ is incremented by 1. Now the count value (i.e., 8) becomes larger than that of the sentinel flow record (i.e., $(f_{13}, 7)$ in $\mathbf{M}_1$). Therefore, *flow record promotion* is triggered and the sentinel flow record is replaced by $(f_5, 8)$.

5 Analysis

We analyze the performance of HashFlow in the case of packet count theoretically in this section. First, we prove that the sentinel flow records that may be replaced in the *flow record promotion* procedure are relatively small among the flow records maintained by $\mathbf{M}$ (Theorem 1). Then, we show how many colliding packets a flow is expected to encounter in $\mathbf{A}$ under a certain memory budget (Theorem 2). At last, we prove that a large flow in $\mathbf{A}$ will have a good chance to be promoted back to $\mathbf{M}$ (Theorem 3). As Theorems 1 and 2 can be easily extended to the case of byte volume, this is not true for Theorem 3. Therefore, we will leave the theoretical analysis of the byte volume case to the future work.

Suppose that $\mathbf{M}$ consists of d subtables $\mathbf{M}_1$, $\mathbf{M}_2$, ..., $\mathbf{M}_d$, each with w buckets, and $\mathbf{A}$ consists of c buckets, so wd and c flow records can be stored in $\mathbf{M}$ and $\mathbf{A}$ respectively.

Theorem 1. *On average, a sentinel flow record is smaller than $\frac{d}{d+1}$ of the wd flow records stored in $\mathbf{M}$.*

Proof. For a given sentinel flow record (with flowID f and size l) in $\mathbf{M}$, according to its definition, there must be a flow record f_i (with size l_i) in each sub-table $\mathbf{M}_i$, such that $l = \min\{l_1, l_2, \ldots, l_d\}$. Use $f(x)$ to represent the probability that x is larger than the sizes of at least a fraction γ of the flow records in $\mathbf{M}$, then $f(l_i) = 1 - \gamma$ since f_i is randomly selected by the hash functions. Therefore

$$f(l) = f(\min\{l_1, \ldots, l_d\}) = f(l_1) \ldots f(l_d) = (1-\gamma)^d.$$

Then the expectation of $f(l)$ is

$$\int_0^1 (1-\gamma)^d d\gamma = \frac{1}{d+1}.$$

Hence the sentinel record f is expected to be larger than $\frac{1}{d+1}$, or smaller than $\frac{d}{d+1}$, of the records stored in $\mathbf{M}$. □

This result shows that, the top $wd \times \frac{d}{d+1}$ heavy flow records in $\mathbf{M}$ tend to not be evicted as sentinel ones, so heavy flows can keep accumulating in $\mathbf{M}$. However, a heavy flow may also fail to find an empty bucket in $\mathbf{M}$ in the first place, so next we analyze how *flow record promotion* helps to record it. Notice that when we refer to a flow record in Theorem 1, we mean the content maintained in our data structures, while when we say a flow below, we are referring to all the packets belonging to this flow.

Suppose that the number of buckets of $\mathbf{A}$ is $c = \lceil \frac{1}{\epsilon\delta} \rceil$, where $\epsilon > 0$ is a positive error bound, and $1 - \delta$ ($\delta < 1$) is some level of probability guarantee, and N packets from n different flows, where flow f_i has l_i packets ($1 \leq i \leq n$), are processed by $\mathbf{A}$. Let X_i denote the total number of packets mapped into the bucket $\mathbf{A}[g(f_i)]$ (i.e., the bucket f_i is mapped into) but belonging to flows other than f_i.

Theorem 2.

$$\Pr(X_i \leq \epsilon N) \geq 1 - \delta.$$

Proof. For any two different flows f_i and f_j, we define I_{ij} as follows:

$$I_{ij} = \begin{cases} 1, & \text{if } \ g(f_i) = g(f_j) \\ 0, & \quad \text{otherwise} \end{cases},$$

then the number of packets colliding with f_i is

$$X_i = \sum_{j=0, j\neq i}^{n} I_{ij} l_j.$$

For a given i, if we assume g is a standard 2-universal hash function, then $\Pr(I_{ij} = 1) = \frac{1}{c}$, and

$$\begin{aligned} E(X_i) &= \sum_{j=0, j\neq i}^{n} E(I_{ij} l_j) \\ &= \sum_{j=0, j\neq i}^{n} l_j \Big(0 \times \Pr(I_{ij} = 0) + 1 \times \Pr(I_{ij} = 1)\Big) \\ &= \frac{1}{c} \sum_{j=0, j\neq i}^{n} l_j \leq \epsilon\delta N. \end{aligned}$$

By Markov Inequality [36]

$$\Pr(X_i \leq \epsilon N) \geq 1 - \frac{E(X_i)}{\epsilon N} \geq 1 - \delta.$$

□

This result is general for any flow f_i, so in the following, we will simply use f, l and X to denote a flow processed in $\mathbf{A}$, its packet count, and the number of colliding packets f may encounter there, respectively.

Now we discuss how *flow record promotion* is carried out. A flow f with l packets can be promoted back from $\mathbf{A}$ to $\mathbf{M}$ only if it first bypasses $\mathbf{M}$, then its packets start to accumulate in the bucket $\mathbf{A}[g(f)]$, and finally its packet count in $\mathbf{A}[g(f)]$ becomes larger than that of its sentinel flow record in $\mathbf{M}$. Denote the size of the sentinel flow record by k, and the total number of packets mapped into $\mathbf{A}[g(f)]$ by m. According to Algorithm 1, to promote f to $\mathbf{M}$, more than k packets of f have to arrive in a row, without being intervened by any of the $m - l$ packets belonging to the other flows. This can be abstracted as picking l boxes from an array of m ordered boxes randomly, and more than k boxes picked must be consecutive in location. We denote the total number of ways where no more than k of the l picked boxes are consecutive by $G(m, l, k)$, then it is easy to see, under the assumption that any packet order is equally possible, the probability that f can be promoted back to $\mathbf{M}$ is $p(m, l, k) = 1 - G(m, l, k)/\binom{m}{l}$. In the following we will show how to calculate $G(m, l, k)$.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

It's obvious that the following equation holds:

$$G(l,l,k) = \begin{cases} 1, & \text{if } \ 0 \le l \le k \\ 0, & \text{otherwise} \end{cases}. \tag{1}$$

To calculate the value of $G(m,l,k)$, we consider the first box that is not picked for any eligible picking method. It's straightforward that the first $k+1$ boxes of the array must contain at least a box that is not picked. Specifically, if the ith box is not picked, where $i = 0, 1, \ldots, k$, and all the boxes before it are picked, then we have to pick another $l-i$ boxes from the remaining $m-(i+1)$ boxes, and no more than k picked boxes should be consecutive in location. Therefore

$$G(m,l,k) = \sum_{i=0}^{k} G(m-1-i, l-i, k). \tag{2}$$

Lemma 1. $G(l+r,l,k) = \sum_{i=0}^{rk} a_i G(l-i,l-i,k)$.

Proof. For any $G(m,l,k)$, we define its *argument distance* as $m-l$. It's obvious that, for any $G(m,l,k)$, when it is expanded once using Equation (2), all the expanded items, namely $G(m-1-i,l-i,k)$ for any $i = 0,1,\ldots,k$, have the same argument distance $m-l-1$. Therefore, to expand $G(l+r,l,k)$ into the sum of items which have the argument distance of 0, it must be expanded r times recursively.

Consider the second argument of the expanded items (e.g., $l-i$ in $G(m-1-i,l-i,k)$). In the best case, the second argument keeps the same after every expansion, but it is reduced by k after every expansion in the worst case. Therefore, the final expansion of $G(l+r,l,k)$ is enclosed by an item $G(l,l,k)$ and another item $G(l-rk,l-rk,k)$.

The item $G(l-(xk+c), l-(xk+c), k)$, where $0 \le x < r$, $0 \le c < k$, can be derived by the following process:

$$\begin{aligned} & G(l+r,l,k) \\ = & \cdots + G(l+(r-1)-k, l-k, k) + \cdots \\ & \cdots \\ = & \cdots + G(l+(r-x)-xk, l-xk, k) + \cdots \\ = & \cdots + G(l+(r-x-1)-(xk+c), l-(xk+c), k) + \cdots \\ & \cdots \\ = & \cdots + G(l-(xk+c), l-(xk+c), k) + \cdots \end{aligned}$$

Therefore, after being expanded r times, the expansion of $G(l+r,l,k)$ contains and only contains the items $G(l-i,l-i,k)$ where $i = 0,1,\ldots,rk$. Hence the theorem holds. □

Lemma 2. *Denote the coefficient of $G(l-t,l-t,k)$ in the expansion of $G(l+r,l,k)$ by $[G(l-t,l-t,k)]G(l+r,l,k)$, and the coefficient of x^t in the expansion of $(1+x+\cdots+x^k)^r$ by $[x^t](1+x+\cdots+x^k)^r$. For $t = 0,1,\cdots,rk$*

$$[G(l-t,l-t,k)]G(l+r,l,k) = [x^t](1+x+\cdots+x^k)^r.$$

Proof. We prove the theorem by induction. First, in the case where $r=1$, $G(l+1,l,k) = \sum_{i=0}^{k} 1 \times G(l-i,l-i,k)$. So the coefficient of $G(l-t,l-t,k)$ (i.e., 1) is the same as the coefficient of x^t in $(1+x+\cdots+x^k)^1$.

Now suppose that the theorem holds for the case where $r=b$. Based on the assumption, we will prove that the theorem holds for the case where $r=b+1$.

By definition

$$\begin{aligned} & G(l+b+1,l,k) = \sum_{i=0}^{k} G((l-i)+b, l-i, k) \\ \Rightarrow & [G(l-t,l-t,k)]G(l+b+1,l,k) \\ = & \sum_{i=0}^{k} [G(l-t,l-t,k)]G((l-i)+b,l-i,k). \end{aligned}$$

Meanwhile, for any $i = 0,1,\ldots,k$

$$G(l-t,l-t,k) = G((l-i)-(t-i),(l-i)-(t-i),k).$$

Since the theorem holds for the case where $r=b$, the coefficient of $G(l-t,l-t,k)$ in the expansion of $G((l-i)+b,l-i,k)$ is equal to $[x^{t-i}](1+x+\cdots+x^k)^b$. This in turn gives

$$[G(l-t,l-t,k)]G(l+b+1,l,k) = \sum_{i=0}^{k} [x^{t-i}](1+x+\cdots+x^k)^b.$$

Notice that the coefficient of x^t in the expansion of $(1+x+\cdots+x^k)^{b+1}$ is

$$[x^t](1+x+\cdots+x^k)^{b+1} = \sum_{i=0}^{k} [x^{t-i}](1+x+\cdots+x^k)^b.$$

So $[G(l-t,l-t,k)]G(l+(b+1),l,k)$ is equal to $[x^t](1+x+\cdots+x^k)^{b+1}$. Hence the theorem always holds. □

Lemma 3. *The value of $G(m,l,k)$ can be calculated as follows:*

$$G(m,l,k) = \sum_{i=l-k}^{l} \sum_{t=0}^{r} (-1)^t \binom{r}{t} \binom{i-t(k+1)+r-1}{r-1} \cdot \sigma(i-t(k+1))\sigma(rk-i),$$

where

$$\sigma(x) = \begin{cases} 1, & \text{if } \ x \ge 0 \\ 0, & \text{otherwise} \end{cases}, \tag{3}$$

and $r = m-l$.

Proof. According to Lemmas 1 and 2

$$G(m,l,k) = G(l+r,l,k) = \sum_{i=0}^{rk} a_i G(l-i,l-i,k),$$

where $a_i = [x^i](1+x+\cdots+x^k)^r$.

As stated earlier

$$G(l,l,k) = \begin{cases} 1, & \text{if } \ 0 \le l \le k \\ 0, & \text{otherwise} \end{cases}.$$

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

Therefore

$$G(m,l,k) = \sum_{i=l-k}^{l} a_i \sigma(rk-i). \quad (4)$$

By multinomial theorem [37], the coefficient of x^i in the expansion of $(1+x+\cdots+x^k)^r$ is

$$a_i = \sum_{t=0}^{r}(-1)^t \binom{r}{t}\binom{r+i-1-t(k+1)}{r-1}\sigma(i-t(k+1)).$$

So the value of $G(m,l,k)$ is

$$G(m,l,k) = \sum_{i=l-k}^{l}\sum_{t=0}^{r}(-1)^t\binom{r}{t}\binom{i-t(k+1)+r-1}{r-1}\cdot \sigma(i-t(k+1))\sigma(rk-i),$$

where $r=m-l$. □

In the following, we will establish a connection between the possibility that a flow is promoted and the size of **A**, when a total number of N packets are processed by **A**.

Theorem 3. *Suppose that the packet count of the sentinel flow record is k, denote the event that a flow f containing l packets is promoted by A, then*

$$\Pr(A) = \sum_{i=0}^{N-l}\left[1-\frac{G(l+i,l,k)}{\binom{l+i}{l}}\right]\cdot\binom{N-l}{i}\cdot\frac{(\epsilon\delta)^i}{(1-\epsilon\delta)^{l+i-N}} \geq [1-G(l+\epsilon N,l,k)/\binom{l+\epsilon N}{l}](1-\delta).$$

Proof. Suppose that every packet except those of f are mapped into the buckets of **A** with the same probability, and X packets besides those of f are mapped into the bucket that f is mapped into, then, as stated before, the probability that f is promoted is $\Pr(A) = 1-G(l+X,l,k)/\binom{l+X}{l}$, and

$$\Pr(X=i) = \binom{N-l}{i} \times \left(\frac{1}{c}\right)^i\left(1-\frac{1}{c}\right)^{N-l-i} = \binom{N-l}{i}\frac{(\epsilon\delta)^i}{(1-\epsilon\delta)^{l+i-N}},$$

where $c=\lceil\frac{1}{\epsilon\delta}\rceil$ is the number of buckets of **A**. Therefore

$$\Pr(A) = \sum_{i=0}^{N-l}\Pr(A|X=i)\Pr(X=i) = \sum_{i=0}^{N-l}\left[1-\frac{G(l+i,l,k)}{\binom{l+i}{l}}\right]\cdot\binom{N-l}{i}\cdot\frac{(\epsilon\delta)^i}{(1-\epsilon\delta)^{l+i-N}}.$$

Meanwhile, by Theorem 2 and total probability theorem

$$\begin{aligned}\Pr(A) &= \sum_{i=0}^{N}\Pr(A|X=i)\Pr(X=i)\\ &\geq \sum_{i=0}^{\epsilon N}\Pr(A|X=i)\Pr(X=i)\\ &\geq \sum_{i=0}^{\epsilon N}\Pr(A|X=\epsilon N)\Pr(X=i)\\ &= \Pr(A|X=\epsilon N)\Pr(X\leq\epsilon N)\\ &\geq \left[1-G(l+\epsilon N,l,k)/\binom{l+\epsilon N}{l}\right](1-\delta).\end{aligned}$$

Hence the theorem holds. □

We denote the top $\frac{wd^2}{d+1}$ flows in the monitored traffic by heavy flows, and the other flows by light flows. According to Theorem 1, the heavy flows and a few light flows should be maintained in **M** with high probability. Suppose the sum of packet counts in **M** is Ω, the heavy flows contains Θ packets, and the other light flows maintained in **M** contains a packets, then $\Omega = \Theta + a$. Meanwhile, suppose Λ packets are directly recorded by **M**, and the *flow record promotion* is triggered s times, then $\Omega = \Lambda + s$, since a sentinel flow record of size k in **M** is usually replaced by another flow record of size $k+1$ when *flow record promotion* is triggered. Since s and a should be far smaller than Θ, $\Lambda = \Theta + a - s \approx \Theta$. Therefore, the number of packets processed by **A**, which we denote by N, should approximate the number of packets from the light flows defined above, which we denote by N'. To prove the prediction, we replay the CAIDA trace using HashFlow with the memory of 1 MB. As shown in Fig. 2a, the number of packets processed by **A** is very close to the number of packets belonging to the light flows.

To illustrate the equality part of Theorem 3, we set δ to 0.1, set ϵ to 0.01, and set the packet count of the sentinel record (i.e., k) to 5. First, we generate and replay $N = 10000$ packets which consist of a flow containing l packets and another $N-l$ random packets, with l increasing from 10 to 50. We repeat the experiment 10000 times and calculate the probability that the l-packet flows are promoted. Second, we take and replay $N = 10000$ packets from the CAIDA trace. We repeat the experiment 2757 times, then calculate the probability that flows with various sizes are promoted. Fig. 2b shows that the probability that the flows of the synthesized trace are promoted strictly conforms to that predicted by Theorem 3. Notice that the flows in the CAIDA trace are promoted with higher probability than that predicted. We argue that it's because strong locality exists in the real traffic since the packets of the same flow tend to arrive continuously, which helps the flows to be promoted.

6 Implementation

We have implemented the algorithms used in this paper in python (software implementation), and HashFlow and PRECISION [27] in P4 [25] in a Tofino switch [24] with the type of Wedge 100BF-32X [38] (hardware implementation) in particular. Especially, the codes for FlowRadar [21], Elastic [23], HashPipe [22] and PRECISION [22] are rewritten

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

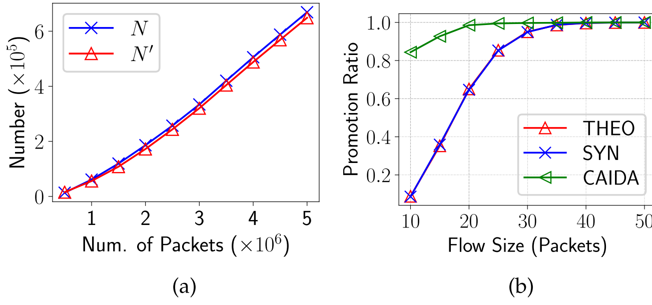


Fig. 2. (a) The number of packets processed by the **A** Table (denoted by N) and that predicted by Theorem 1 (denoted by N'); (b) The promotion ratios predicted by Theorem 3, achieved by the synthesized traffic, and achieved by the CAIDA trace.

based on their published code, while SpaceSaving [18], CM sketch [15], CU sketch [16] and Count sketch [17] are implemented based on the algorithms in the published papers. The memory allocated to the algorithms is 1 MB by default.

A P4 program is composed of a bunch of match-action tables, and is compiled into a pipeline consisting of multiple stages. Multiple match-action tables can be packed into a stage and executed in parallel if they are independent of each other, but two tables with dependency relationship must be placed in different stages. Moreover, every stage contains a certain amount of SRAM[1] which can only be accessed by the tables located in the same stage. There are only very limited number of stages in the Tofino switch.[2] Therefore, the P4 programs must be very carefully designed so that HashFlow and PRECISION can fit into the pipeline.

We use SRAM to store flow records in our implementation. $\mathbf{M}$ consists of three sub-tables $\mathbf{M}_1, \mathbf{M}_2, \mathbf{M}_3$ in the implementation of HashFlow. We use 5 arrays of buckets for the key fields (i.e., one array for the srcip, dstip, protocol, srcport, dstport respectively) of $\mathbf{M}_1, \mathbf{M}_2, \mathbf{M}_3$ in stage 0, 2, 4, and one array of buckets, each bucket with the width of 32 bits, for their count fields in stage 1, 3, 5. The **A** table where the key fields and count fields all have the width of 8 bits is in stage 6. We record the count fields and indexes of the buckets a packet is mapped into when the packet is processed by $\mathbf{M}_1, \mathbf{M}_2$ and $\mathbf{M}_3$, and determine the sentinel flow record and update **A** in stage 6. Stage 7 determines whether to trigger the *flow record promotion*, and if does, it is triggered in stage 8. This implementation contains more than 1000 lines of P4 source code, but uses 9 stages only. We share the source code here [39].

While the *flow record promotion* procedure requires to revisit one of the sub-tables of $\mathbf{M}$, backtracking is not allowed in the pipeline. Our solution is to use the *resubmit* primitive, which sends a packet and some metadata to the ingress of the pipeline. When the resubmitted packet is processed, the resubmitted metadata can provide enough information for replacing the sentinel flow record. Since more packets than that transmitted through the Tofino switch are processed when the *resubmit* primitive is used, the throughput of the switch is influenced negatively. In Section 7.6 we will evaluate the loss of throughput caused by the *resubmit* primitive.

Following recommendations in the corresponding papers, we use four sub-tables of equal size for HashPipe and PRECISION. With regards to FlowRadar, we use four hash functions for its bloom filter and three hash functions for its counting table. The number of buckets in the bloom filter is $40\times$ of that in the counting table.

Elastic paper [23] provides various versions of Elastic. We pick the P4 version (denoted by P4 Elastic) and hardware version (denoted by Hardware Elastic) in this paper for evaluation. Hardware Elastic contains a heavy part, which consists of three sub-tables with the same number of buckets, and a light part. Instead, P4 Elastic doesn't have a light part, and its heavy part consists of four sub-tables with the same number of buckets. As recommended in the paper, the value of λ is 8 for Hardware Elastic and 32 for P4 Elastic.

The implementation of SpaceSaving consists of a SpaceSaving sketch and a hash table, each occupying half of the allocated memory. When a new packet arrives, if a flow record corresponding to it exists in the hash table, we simply update the flow record there. Otherwise, we create a flow record for this packet in the hash table and the sketch if the hash table is not full, or query the sketch (and synchronize it with the hash table meanwhile) for the flow record with the minimum packet count and replace it with a new one corresponding to the new packet if the hash table is full.

The implementations of Count, CM, CU use three data structures, namely, a Count/CM/CU sketch, a hash table, and a min heap. The sketch occupies half of the allocated memory, and each of the min heap and hash table occupies 1/4 of the allocated memory. Moreover, the sketch uses four hash functions. When a new packet arrives, we use it to update the sketch, which will return an estimated size of the flow record corresponding to the packet. Then we use this packet and the estimated size to update the hash table and the min heap. If a flow record corresponding to the packet exists in the hash table, the packet count of the flow record is incremented by 1 simply. Otherwise, the min heap will synchronize with the hash table and give a flow record with the minimum packet count. This flow record will be replaced by the flow record consisting of the flow ID of the packet and the estimated size given by the sketch if its packet count is smaller than the estimated size.

To save space, we use HF, HP, FR, SS, PE, HE in place of HashFlow, HashPipe, FlowRadar, SpaceSaving, P4 Elastic, Hardware Elastic, respectively, when plotting figures.

7 Evaluation

In this section, we evaluate the performance of HashFlow in various applications, but we mainly focus on flow records in terms of packet count, and we will state it explicitly when byte volume is considered.

7.1 Methodology

We use 4 traces from different operational networks, namely, one from a 40 Gbps backbone link provided by CAIDA [29], one from a 10 Gbps link in a campus network, one from a backbone link of the Hutchison Global

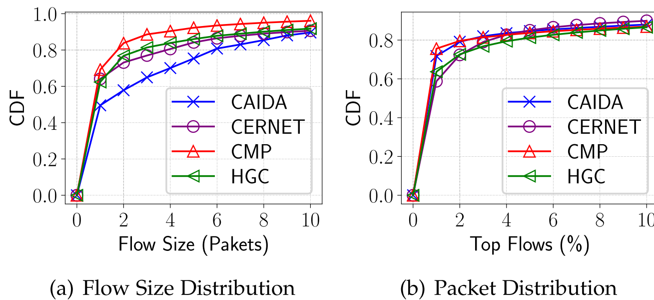
1. It is called *register* in the context of Tofino, but we avoid this terminology to avoid confusion with the registers in the CPU architecture.
2. We cannot present the specific number of stages available in our Tofino switch due to the Non-Disclosure Agreement.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Traces Used for Evaluation

Trace	Max Flow Size	Ave. Flow Size	# Concur. Flows[3]
CAIDA	460212 pkts	20.4 pkts	149339
Campus	1554222 pkts	9.1 pkts	774510
HGC	213979 pkts	11.4 pkts	124244
CERNET	27209 pkts	16.7 pkts	93977

[3] *the number of concurrent flows.*

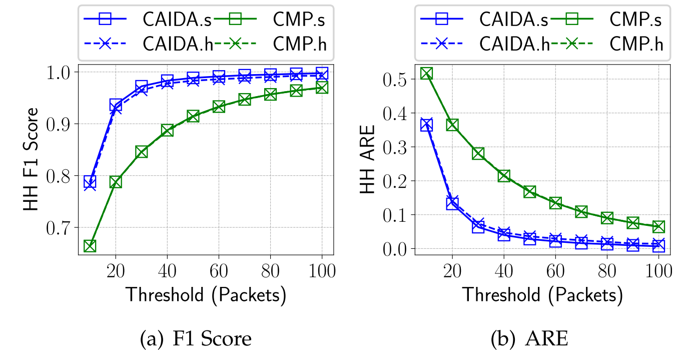


(a) Flow Size Distribution (b) Packet Distribution

Fig. 3. Light flows accounts for a great portion of the flows, but most packets are from the Heavy flows.

TABLE 2
Metrics Used for Various Applications

	WRE	ARE	F1 Score
General Flow Size Estimation	✓	✓	×
Flow Size Distribution Estimation	×	✓	×
Heavy Hitter Detection	×	✓	✓
Top-K Heavy Flows Detection	×	✓	✓



(a) F1 Score (b) ARE

Fig. 4. The performance of hardware and software implementations of HashFlow in heavy hitter detection.

Communications (HGC), and the other one from a 10 Gbps backbone link in CERNET [40], for performance evaluation. Some flow level (defined by the typical 5-tuple [28]) statistics, as summarized in Table 1, shows that the traffic in different traces differs greatly from each other. However, by having a deeper inspection into the flows, we find that they all exhibit a similar skewness pattern. For example, about 90 percent of the flows are light flows containing no more than 10 packets (as shown in Fig. 3a), while the top 10 percent heavy flows account for about 85 percent of the packets (as shown in Fig. 3b).

Applications considered for performance evaluation include general flow size estimation, flow size distribution estimation, top-k heavy flows detection, and heavy hitter detection, where, given a threshold T, a flow is recognized as a heavy hitter only if it contains no less than T packets. Given a packet sequence, with regards to each application, a target set $\mathbb{S}$ is defined, and the algorithm to be evaluated will give an estimated set $\hat{\mathbb{S}}$. For example, $\mathbb{S}$ is the set of all the flows, heavy hitters, or top-k heavy flows, contained in the packet sequence for general flow size estimation, heavy hitter detection, or top-k heavy flows detection, respectively. Given a flow f, $\mathbb{S}[f]$ (or $\hat{\mathbb{S}}[f]$) is the size of f in $\mathbb{S}$ (or $\hat{\mathbb{S}}$). For flow size distribution estimation, $\mathbb{S}$ consists of pairs in the form of (k, n_k) and $\mathbb{S}[k] = n_k$, where k is an integer and n_k is the number of flows containing k packets. For all the applications, $\hat{\mathbb{S}}[x] = 0$ by default if $x \notin \hat{\mathbb{S}}$ and $\mathrm{len}(\mathbb{S})$ is the number of elements in $\mathbb{S}$.

We use Average Relative Error (ARE), Weighted Relative Error (WRE) and F1 score, to evaluate the performance of the algorithms. ARE and WRE are defined as follows:

$$\mathrm{ARE} = \frac{1}{\mathrm{len}(\mathbb{S})} \sum\nolimits_{x\in\mathbb{S}} \left| \frac{\mathbb{S}[x] - \hat{\mathbb{S}}[x]}{\mathbb{S}[x]} \right|$$

$$\mathrm{WRE} = \frac{\sum_{x\in\mathbb{S}} \left|\mathbb{S}[x] - \hat{\mathbb{S}}[x]\right|}{\sum_{x\in\mathbb{S}} \mathbb{S}[x]}.$$

We define $\tilde{\mathbb{S}}$ to be $\tilde{\mathbb{S}} = \mathbb{S} \cap \hat{\mathbb{S}}$, so the precision rate (PR) and recall rate (RR) are

$$\mathrm{PR} = \frac{\mathrm{len}(\tilde{\mathbb{S}})}{\mathrm{len}(\hat{\mathbb{S}})}, \quad \mathrm{RR} = \frac{\mathrm{len}(\tilde{\mathbb{S}})}{\mathrm{len}(\mathbb{S})},$$

and F1 score is defined as

$$\mathrm{F1\ Score} = \frac{2 \cdot \mathrm{PR} \cdot \mathrm{RR}}{\mathrm{PR} + \mathrm{RR}}.$$

The relationship between the applications and the metrics are shown in Table 2.

7.2 Hardware versus Software

In the following, we will evaluate the performance of HashFlow using the software implementation for flexible parameter tuning and statistics collection, but before that, we need to prove the correctness of this implementation by comparing its performance with that of the hardware implementation. Meanwhile, to evaluate the impact different hash functions have on the performance of HashFlow, we use CRC for the hardware implementation and MD5 for the software implementation to generate the digest and indexes. We use the two implementations to detect the heavy hitters with the threshold increasing from 10 packets to 100 packets while replaying 10^7 packets from the CAIDA trace and the Campus trace, using a Cisco UCS server to push the packets through the Tofino switch particularly for the hardware implementation. As shown in Fig. 4, the difference in performance between the two implementations is negligible, which assured that the software implementation is correct, and the impact different hash functions have on the performance of HashFlow is insignificant.

7.3 Application Performance

General Flow Size Estimation & Flow Size Distribution Estimation. In this experiment, we replay a number of packets

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

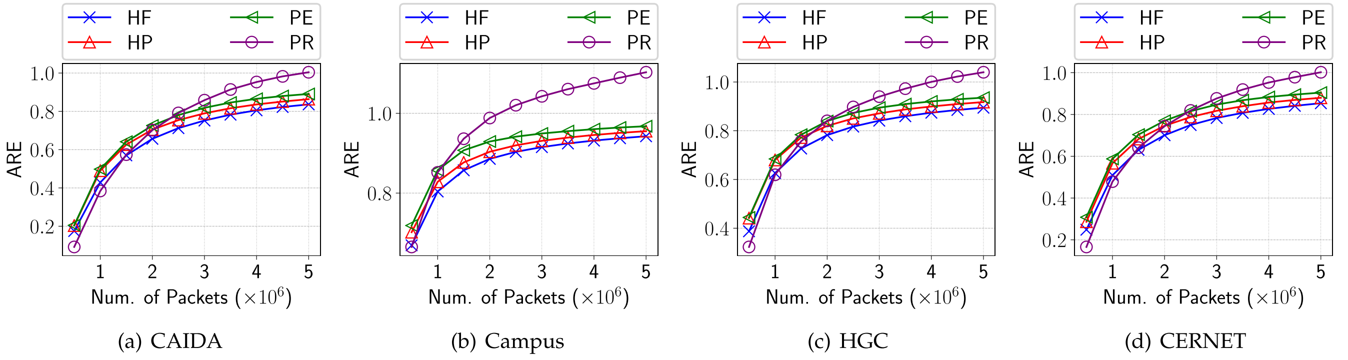


Fig. 5. *Average relative error (ARE)* for *general flow size estimation*.

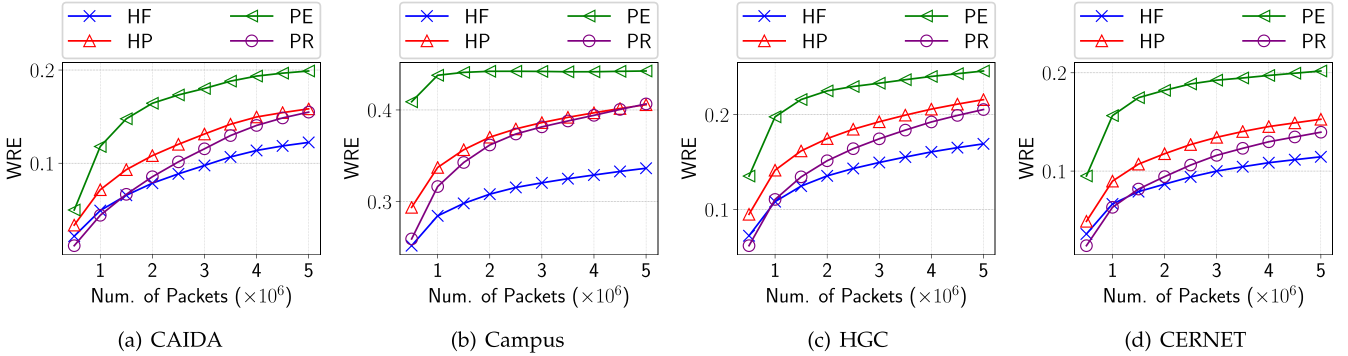


Fig. 6. *Weighted Relative Error (WRE)* for *General flow size estimation*.

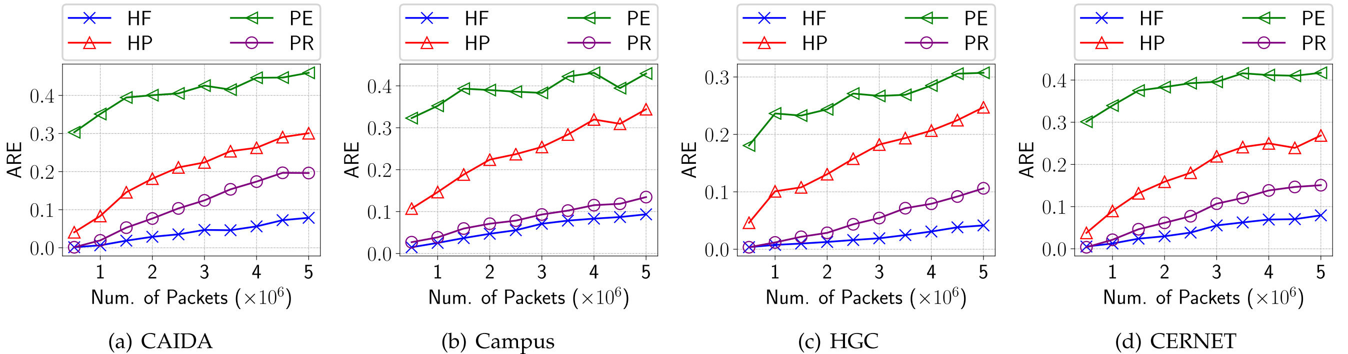


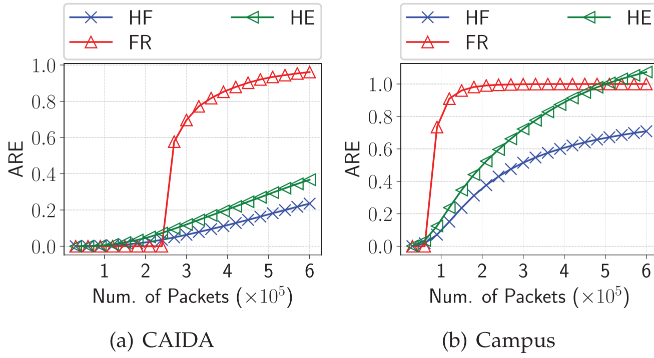
Fig. 7. *Average relative error (ARE)* for *flow size distribution estimation*.

from each of the traces, then evaluate the performance of HashFlow, P4 Elastic, HashPipe and PRECISION in general flow size estimation and flow size distribution estimation. The number of replayed packets increases from 5×10^5 to 5×10^6. As shown in Fig. 5, although the ARE of PRECISION is smaller than that of HashFlow when the number of replayed packets is small, the ARE of HashFlow is better than those of the competitors when the number of packets is no less than 1.5×10^6. When it comes to the metric of WRE, the performance of HashFlow is much more outstanding. For example, as shown in Fig. 6, when 5×10^6 packets from the CAIDA trace are replayed, the WRE of HashFlow is 0.12, which is 21 percent better than that of the best competitor. Fig. 7 shows that, when doing flow size distribution estimation, the performance of HashFlow is clearly better than those of the competitors. For example, when 5×10^6 packets of CAIDA are replayed, the ARE of HashFlow (i.e., 0.08) is 60 percent smaller than that of the best competitor.

Notice that the performance of HashFlow varies across data sets, which we believe is caused by the difference in the traffic patterns of the traces. Consider the flow size entropy defined as $-\sum \frac{m_i}{m} \ln \frac{m_i}{m}$, where m is the total number of packets and m_i is the number of packets in flow f_i. It is expected that HashFlow will perform worse for a data set with a larger entropy, since that means greater randomness in flow sizes. Actually, the CAIDA trace has an entropy of 9.58 and the Campus trace has an entropy of 11.72, while the other traces sit in between.

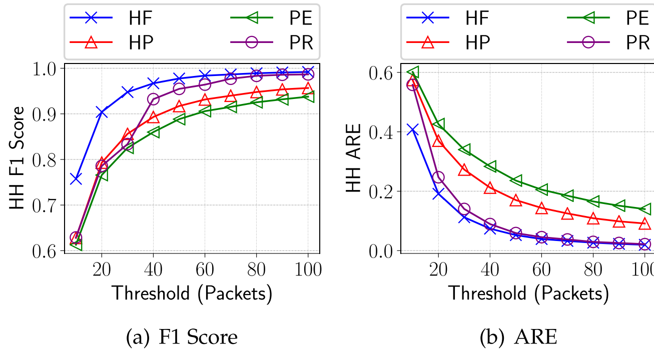
When the number of replayed packets is small, FlowRadar collects the flow records very accurately, and Hardware Elastic usually has better performance than P4 Elastic since the light part of Hardware Elastic can compensate for the packets that are not recorded by the heavy part. However, if too many packets are processed, the light part of Hardware Elastic and the counting table of FlowRadar may be overwhelmed, making their performance degrade abruptly. Therefore, we conduct another experiment, where we

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.



(a) CAIDA (b) Campus

Fig. 8. *Average relative error (ARE)* for *general flow size estimation*.



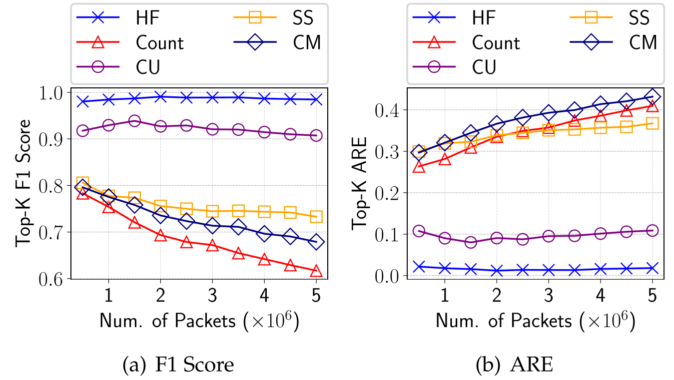
(a) F1 Score (b) ARE

Fig. 9. The performance of HashFlow in heavy hitter detection.

increase the number of replayed packets from 3×10^4 to 6×10^5, and evaluate the performance of HashFlow, Hardware Elastic and FlowRadar in general flow size estimation. As shown in Fig. 8, when no more than 2.4×10^5 packets of CAIDA are replayed, the ARE of FlowRadar is 0, which is perfect performance. However, when the number of replayed packets becomes 2.7×10^5, the ARE of FlowRadar increases to 0.58 abruptly. On the contrary, the ARE of HashFlow increases smoothly, and is clearly better than that of Hardware Elastic. For example, when 6×10^5 packets of CAIDA are replayed, the ARE of HashFlow (i.e., 0.24) is 36 percent smaller than that of Hardware Elastic.

Since the performances of HashFlow as well as the other algorithms on the traces are similar, in the following we will present the performance of the algorithms using only one of the traces as the representative case.

Heavy Hitter Detection. To evaluate the performance of HashFlow in heavy hitter detection, we replay 10^7 packets from the HGC trace, then calculate the ARE and F1 score of the algorithms as the threshold of heavy hitters increases from 10 packets to 100 packets. As shown in Fig. 9, the algorithms have better performance as the threshold increases, and HashFlow achieves the F1 score of 0.99 and the ARE of 0.02 when the threshold is 100 packets. While the performance of PRECISION is very close to that of HashFlow when the threshold is large, it has bad performance for the smaller thresholds since in PRECISION the light flows are replaced randomly, and a new flow is recorded with an initial packet count irrelevant to its size. For the thresholds of 10 packets and 20 packets, the F1 scores of HashFlow are 0.76 and 0.90 respectively, which are 20 and 14 percent better than those of the best competitor, and the AREs of HashFlow are 0.41 and 0.19 respectively, which are more than 23 percent better than those of the best competitors. It's worth noting that when the CAIDA trace is replayed, for the heavy hitters with the threshold of 10 packets, the F1 score and ARE of HashFlow are 0.81 and 0.33 respectively, which are 19 and 35 percent better than those of the best competitors.



(a) F1 Score (b) ARE

Fig. 10. The performance of HashFlow in top-K heavy flows detection.

Top-K Heavy Flows Detection. To evaluate the performance of HashFlow in top-k heavy flows detection, we replayed a number of packets from the Campus trace. The number of replayed packets increases from 5×10^5 to 5×10^6. Then we evaluate the performance of HashFlow, Count sketch, CU sketch, CM sketch and SpaceSaving in collecting the top 10^4 heavy flows. As shown in Fig. 10, the F1 score of HashFlow ($\approx$0.99) is about 7 percent better than that of the best competitor, and the ARE of HashFlow ($\approx$0.015) is about 84 percent smaller than that of the best competitor. It's clear from Fig. 10 that as the number of replayed packets increases, the performance of HashFlow is very steady while that of the other algorithms degrade constantly. That's because the old information in a bucket of **A** of HashFlow is erased whenever collision occurs in it, while for the other algorithms, the information in the sketches is accumulated as more packets are processed, making the error in flow size estimation increase constantly, which may result in the tracked heavy flows being replaced by the light flows mistakenly.

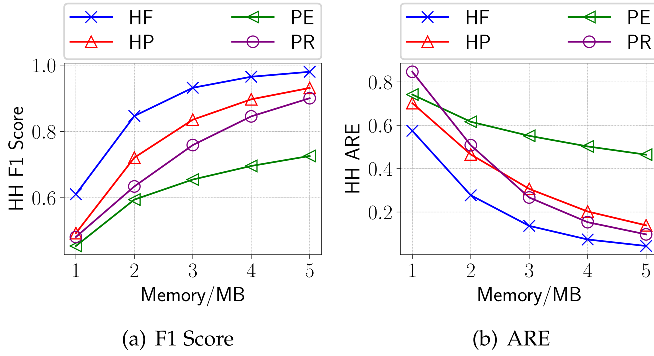
7.4 Measurement Accuracy and Memory Consumption

To evaluate the performance of HashFlow when the memory allocated changes, we replay 2×10^7 packets of the CAIDA trace, and increase the memory consumption of the algorithms from 1 MB to 5 MB. Since 5 MB of memory can accommodate about 2.5×10^5 flow records only, and the 2×10^7 packets contains more than 10^6 flows, it's meaningless to evaluate the performance of HashFlow in general flow size estimation. Instead, we evaluate the performance of HashFlow and the competitors in heavy hitter detection with the threshold of 10 packets. As shown in Fig. 11, as the allocated memory increases from 1 MB to 5 MB, the F1 score of HashFlow increases from 0.61 to 0.98, which is 5%$\sim$24% better than that of the best competitor, and the ARE of HashFlow decreases from 0.58 to 0.04, which is 18%$\sim$69% better than that of the best competitor.

7.5 Measurement Accuracy and Traffic Skewness

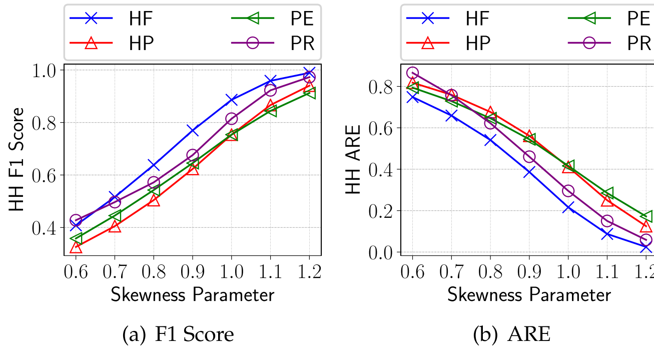
To evaluate the performance of HashFlow for various traffic patterns, we generate a packet sequence following the zipf

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.



(a) F1 Score (b) ARE

Fig. 11. Performance for heavy hitter detection with $T = 10$.



(a) F1 Score (b) ARE

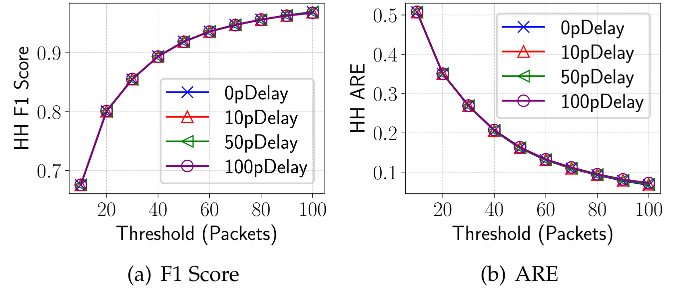
Fig. 12. Performance for heavy hitter detection with $T = 10$.

distribution [41]. The packet sequence contains 5×10^6 packets and the average flow size is 20 packets. We increase the skewness parameter of the zipf distribution from 0.6 to 1.2, then compare the performance of HashFlow in heavy hitter detection with the threshold of 10 packets against those of the competitors. As shown in Fig. 12, HashFlow usually has better performance than its competitors. Especially, when the skewness parameter is 0.9, the F1 Score and ARE of HashFlow are 0.77 and 0.39, which are 13.9% and 16.0% better than those of the best competitor respectively.

7.6 Influence of the *Resubmit* Primitive

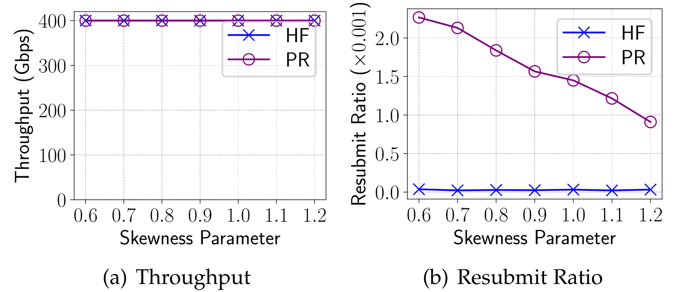
As stated in Section 6, we use the *resubmit* primitive to do the *flow record promotion*. The influence of this solution is twofold. On the one hand, when a packet is resubmitted, it is passed to the ingress of the pipeline, and several other packets that are already in the pipeline have to be processed before the sentinel flow record is replaced, thus the effectiveness of the *flow record promotion* may be undermined. On the other hand, the *resubmit* operation results in some packets being processed twice, which may degrade the throughput of the Tofino switch. In this section, we will evaluate the influence the use of *resubmit* primitive has on the performance of HashFlow.

First, we set the delay (in packets) for replacing the sentinel record of the software implementation to a certain value t, so that once the *flow record promotion* is triggered, the sentinel record is replaced only after another t packets are processed. Notice there are only a few tens of stages in the pipeline of a Tofino switch, the delay should be very small. We replay 10^7 packets from the Campus trace, then collect heavy hitters using HashFlow with the delays of 0 packets, 10 packets, 50 packets and 100 packets respectively. As shown in Fig. 13, the difference in performance of HashFlow with different delays is negligible, so it's usually safe to use the *resubmit* primitive for *flow record promotion*.



(a) F1 Score (b) ARE

Fig. 13. The performance of HashFlow in heavy hitter detection.



(a) Throughput (b) Resubmit Ratio

Fig. 14. The throughput and resubmit ratio for the Tofino switch.

Notice that both HashFlow and PRECISION adopt the *resubmit* primitive. To evaluate the throughput of HashFlow and PRECISION, we connect the Tofino switch to a Spirent SPT-N4U tester [42] using optical fibers and 100 Gbps optical transceiver modules. We use four ports of the Tofino switch, and initiate 16384 flows for each port, so that 65536 flows in total are active in the traffic. The packet size is 700 bytes, for each port the flow speeds follow the zipf distribution [41], with their sum being about 99.99 Gbps meanwhile (it is impossible to make the sum 100 Gbps exactly), and the experiment duration is 5 seconds. We allocate about 540 KB of memory to the algorithms so that the algorithms can maintain about 30K flow records. As shown in Fig. 14a, as no packet losses are observed during the 5 seconds, the throughput of the Tofino switch is almost 400 Gbps. Some packet losses are observed when the experiment lasts 60 seconds, but the number of packet losses is much smaller than that of the *resubmit* operations, implying that not all *resubmit* operations result in packet losses.

We also define *Resubmit Ratio* as follows:

$$\text{Resubmit Ratio} = \frac{\text{number of resubmit operations}}{\text{number of packets}}.$$

As shown in Fig. 14b, in this experiment the *resubmit* primitive is triggered about 14.5K times by HashFlow, making the *Resubmit Ratio* about 3.2×10^{-5}, which is significantly smaller than that of PRECISION. In a more realistic experiment where 5×10^6 packets from the HGC trace are pushed by a Cisco UCS server through the Tofino switch, the *resubmit* primitive is triggered 51.1K times by HashFlow, and the

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

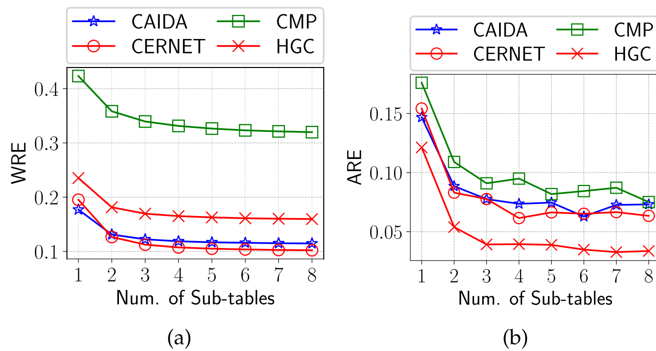


Fig. 15. (a) Weighted relative error for general flow size estimation; (b) Average relative error for flow size distribution estimation.

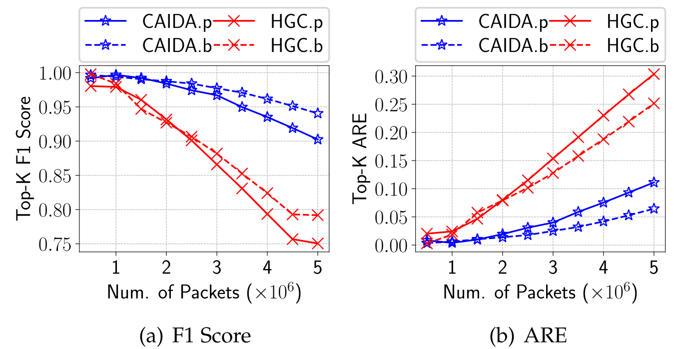


Fig. 16. Performance of HashFlow in detecting the top 10 percent flows when packet volume and byte volume are considered respectively.

Resubmit Ratio of HashFlow (1.02%) is much smaller than that of PRECISION (6.39%) as well. Therefore, in the extreme case where each *resubmit* operation results in a packet being dropped, the throughput degradation of HashFlow is no more than 1.02%.

7.7 The Choice of Number of Sub-Tables

Notice that we use 3 sub-tables for the main table **M** by default. In this section, we will demonstrate the performance of HashFlow when more or fewer sub-tables are used. We create eight instances of HashFlow where the number of sub-tables of **M** increases from 1 to 8, then replay 5×10^6 packets from each of the four traces, and use the eight instances to do the general flow size estimation and the flow size distribution estimation. As shown in Fig. 15, when the number of sub-tables increases from 1 to 3, the performance of HashFlow is improved significantly, but when it increases further, the improvements become trivial. For example, when this number increases from 2 to 3, the WRE for the Campus trace decreases from 0.358 to 0.339, but when it increases from 3 to 4, the WRE decreases from 0.339 to 0.331 only. In our implementation, each sub-table consumes 2 stages of the pipeline, and a maximum number of 4 sub-tables are supported by our Tofino switch, but this setting requires to do 6 (i.e., $\binom{4}{2}$) comparisons to obtain the sentinel flow record, which is much more tedious when configuring the match-action tables from the control plane than the case where there are 3 sub-tables and 3 (i.e., $\binom{3}{2}$) comparisons are needed to obtain the sentinel flow record. Therefore, we believe that it is a good choice to take 3 as the default number of sub-tables.

7.8 Packet Count versus Byte Volume

As mentioned in Section 4, HashFlow is able to collect flow records based on the packet count as well as the byte volume. To illustrate the strength of HashFlow in byte volume oriented flow record collection, we replay a sequence of packets from the CAIDA trace and the HGC trace, with the number of replayed packets increasing from 5×10^5 to 5×10^6, then use HashFlow to collect the top 10 percent heavy flows by considering the packet count and the byte volume respectively. As shown in Fig. 16, HashFlow generally has better performance when byte volume, instead of the packet count, is considered. For example, when 5×10^6 packets from the CAIDA trace are replayed, the ARE when byte volume is considered is 42.0% better than that when the packet count is considered.

8 Conclusion

We propose HashFlow for efficient collection of flow records, which is useful for a wide range of measurement and analysis applications. The *flow collision resolution* and *flow record promotion* are 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 P4 in a Tofino switch. The evaluation results based on real traces from different operational networks show that HashFlow consistently achieves clearly better performance than its competitors in all the cases. In the future, we plan to study how to make it adaptive to more accurate measurement of light flows and network-wide measurement.

Acknowledgments

This work was supported in part by the National Key R&D Program of China under Grant 2018YFB1800401 and in part by National Natural Science Foundation of China under Grants 61972189 and 62002009.

References

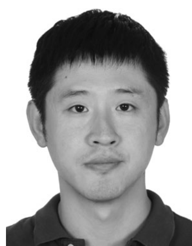
[1] B. Claise, “Cisco systems netflow services export version 9,” Tech. Rep. RFC 3954, Oct. 2004.
[2] B. Claise, B. Trammell, and P. Aitken, “Specification of the IP flow information export (IPFIX) protocol for the exchange of flow information,” Tech. Rep. RFC 7011, Sep. 2013.
[3] J. Case, M. Fedor, M. Schoffstall, and J. Davin, “Simple network management protocol (SNMP),” Tech. Rep. RFC 1157, May 1990.
[4] P. Goyal and A. Goyal, “Comparative study of two most popular packet sniffing tools-Tcpdump and Wireshark,” in *Proc. 9th Int. Conf. Comput. Intell. Commun. Netw.*, 2017, pp. 77–81.
[5] 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,” *Comput. Netw.*, vol. 127, pp. 352–367, Nov. 2017.
[6] H. Zhang, X. Shi, X. Yin, F. Ren, and Z. Wang, “More load, more differentiation — A design principle for deadline-aware congestion control,” in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2015, pp. 127–135.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

[7] Z. Wang *et al.*, "Efficient scheduling of weighted coflows in data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 9, pp. 2003–2017, Sep. 2019.
[8] Access Time of DRAM and SRAM, 2018. Accessed: Jul. 6, 2018. [Online]. Available: https://www.webopedia.com/TERM/A/access_time.html
[9] Random Sampled NetFlow, 2006. Accessed: Jul. 31, 2020. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/ios/12_2sb/feature/guide/sbrsnf.html
[10] N. Hohn and D. Veitch, "Inverting sampled traffic," in *Proc. 3rd ACM SIGCOMM Conf. Internet Meas.*, 2003, pp. 222–233.
[11] 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.
[12] P. Tune and D. Veitch, "Towards optimal sampling for flow size estimation," in *Proc. 8th ACM SIGCOMM Conf. Internet Meas.*, 2008, pp. 243–256.
[13] N. Duffield, "Sampling for passive internet measurement: A review," *Stat. Sci.*, vol. 19, pp. 472–498, 2004.
[14] 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, pp. 1083–1099, Apr. 2011.
[15] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," in *Proc. LATIN Theor. Inf.*, 2004, pp. 29–38.
[16] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. Conf. Appl., Technol., Architectures, Protoc. Comput. Commun.*, 2002, pp. 323–336.
[17] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proc. Int. Colloq. Automata, Lang. Program.*, 2002, pp. 693–703.
[18] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-K elements in data streams," in *Proc. Int. Conf. Database Theory*, 2005, pp. 398–412.
[19] 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.
[20] 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. Conf. ACM SIGCOMM*, 2016, pp. 101–114.
[21] 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.
[22] 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.
[23] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 561–575.
[24] Explore the Power of Intel® Programmable Ethernet Switch Products. Accessed: Nov. 23, 2020. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html
[25] P. Bosshart, *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, 2014.
[26] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," in *Proc. Conf. Appl., Technol., Architectures, Protoc. Comput. Commun.*, 2004, pp. 245–256.
[27] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient measurement on programmable switches using probabilistic recirculation," in *Proc. IEEE 26th Int. Conf. Netw. Protoc.*, 2018, pp. 313–323.
[28] M. Bagnulo, P. Matthews, and I. van, "Stateful NAT64: Network address and protocol translation from IPv6 Clients to IPv4 servers," Tech. Rep. RFC 6146, Apr. 2011.
[29] CAIDA UCSD Anonymized Internet traces dataset - 2018, 2018. Accessed: Jul. 27, 2018. [Online]. Available: http://www.caida.org/data/passive/passive_dataset.xml
[30] M. Mitzenmacher, T. Steinke, and J. Thaler, "Hierarchical heavy hitters with the space saving algorithm," in *Proc. Meeting Algorithm Eng. Experiments*, 2012, pp. 160–174.
[31] D. Anderson, P. Bevan, K. Lang, E. Liberty, L. Rhodes, and J. Thaler, "A high-performance algorithm for identifying frequent items in data streams," in *Proc. Internet Meas. Conf.*, 2017, pp. 268–282.
[32] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, pp. 122–144, May 2004.
[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[34] P. Braam, "The lustre storage architecture," 2019, *arXiv:1903.01955*.
[35] Z. Zeng and B. Veeravalli, "Design and performance evaluation of queue-and-rate-adjustment dynamic load balancing policies for distributed networks," *IEEE Trans. Comput.*, vol. 55, no. 11, pp. 1410–1422, Nov. 2006.
[36] E. W. Weisstein, "Markov's Inequality." Accessed: Nov. 27, 2020. [Online]. Available: https://mathworld.wolfram.com/MarkovsInequality.html
[37] "Multinomial theorem," *Wikipedia*, 2020. Accessed: Nov. 26, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Multinomial_theorem&oldid=986621683
[38] Edgecore Networks, 2010. Accessed: Jan. 8, 2019. [Online]. Available: https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=335
[39] P4 Source Code of Hashflow. Accessed: May 3, 2020. [Online]. Available: http://gitee.com/zhao_zong_yi/open-p4-projects
[40] IP Trace and Service of CERNET, 2011. Accessed: Oct. 27, 2020. [Online]. Available:http://www.iptas.edu.cn/show.php
[41] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp. 267–280.
[42] Spirent SPT-N4U Compact Chassis. Accessed: Nov. 25, 2020. [Online]. Available: https://www.spirent.com/assets/spirent_n4u_chassis_datasheet

Zongyi Zhao received the BS degree in software engineering from Nankai University in 2013 and the ME degree in computer science and technology in 2016 from Tsinghua University, where he is currently working toward the PhD degree in computer science and technology. His research interests include network measurement, data plane programming, and packet loss detection.

Xingang Shi (Member, IEEE) received the BS degree from Tsinghua University and the PhD degree from The Chinese University of Hong Kong. He is currently with the Institute for Network Sciences and Cyberspace, Tsinghua University and the Beijing National Research Center for Information Science and Technology. His research interests include network measurement and routing protocols.



Zhiliang Wang (Member, IEEE) received the BE, ME, and PhD degrees in computer science from Tsinghua University in 2001, 2003, and 2006, respectively. He is currently with the Institute for Network Sciences and Cyberspace, Tsinghua University and the Beijing National Research Center for Information Science and Technology. His research interests include formal methods and protocol testing, next generation internet, and network measurement.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.

Qing Li received the BS degree from the Dalian University of Technology in 2008 and the PhD degree from Tsinghua University, in 2013, both in computer science and technology. He is currently a research associate professor with the Southern University of Science and Technology. His research interests include reliable and scalable routing of the internet, software defined networking, network function virtualization, in-network caching/computing, intelligent self-running network, and edge computing.

Han Zhang (Member, IEEE) received the BS degree in computer science and technology from JiLin University and the PhD degree from Tsinghua University. He is currently with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include computer networks, network security, and network system.

Xia Yin (Senior Member, IEEE) received the BE, ME, and PhD degrees from Tsinghua University in 1995, 1997, and 2000, respectively. She is currently with the Department of Computer Science and Technology, Tsinghua University and the Beijing National Research Center for Information Science and Technology. Her research interests include future internet architecture, formal method, protocol testing, and large-scale internet routing.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on January 08,2024 at 02:58:47 UTC from IEEE Xplore. Restrictions apply.