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  
***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. 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 such as FlowRadar, HashPipe and ElasticSketch, in a P4 software switch. We implement**  
**HashFlow in a commodity P4 switch in particular to illustrate its**  
**implementability. Then we 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  
A proper view of the statistics and the dynamics of a  
network is of great importance to network management. It  
enables network operators to detect and correct configuration  
errors, allocate resources and perform traffic engineering, or  
detect network attacks. 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. 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 measurement epoch. This pose  
stringent pressure on the scarce high speed memories, such as  
on-chip SRAM with 1 *∼* 10 nano-seconds access delay[2][3].  
One straightforward solution is to use sampling [4], 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 less packets or flows being  
recorded, thus less accurate statistics that can be estimated.  
To remedy this, very enhanced sampling algorithms [5][6][7]  
have been proposed and tailored for specific measurement requirement, and their impact analyzed[8][9]. Another direction  
of solution is to use sketch (also referred as data streaming  
algorithms) [10][11][12], 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[13], UnivMon[14], FlowRadar[2], HashPipe[15],  
and ElasticSketch[16], etc. Both constant bound of 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 few  
packets), by applying a novel collision resolution and record  
promotion strategy to hash tables. The collision resolution part  
eliminates collisions that may mix up packets from different  
flows, keeps a flow from being evicted until a flow with larger  
size collides with it, and fills up nearly all hash table buckets.  
On the other hand, the promotion part bounces a flow back  
from the summarized set to the accurate set, when this flow  
becomes large enough, and replaces the original one which  
has smaller size. 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 [17] software switch[18]. To illustrate the  
implementability of HashFlow, we further implement it in  
a commodity P4 switch [19] which has the type of Wedge  
100BF-32X[20]. 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 almost no 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  
[21]. 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 introduce the  
mechanisms of several recently proposed algorithms, i.e.,  
HashPipe[15], ElasticSketch[16] and FlowRadar[2]. In doing  
this, we try to point out some minor defects in the algorithms,  
not for criticism, but for possible enhancement and new  
strategy we may introduce to HashFlow.  
HashPipe[15] uses a series of independent hash tables (each  
with a different hash function). When a packet comes to the  
first table, if in the bucket it is hashed into, there already exists  
a record of another flow, then the old flow will be evicted to  
make room for the new flow. The evicted flow record will  
then try to find an empty bucket in the remaining hash tables  
as a newcomer. When a collision happens there, among the  
newcomer and the existing record, the one with a smaller  
packet count will be kicked out, and become the newcomer  
to the next table. This process goes on until either an empty  
bucket is found, or there is no remaining hash table (in which  
case the last evicted flow will be discarded).  
HashPipe uses the first table 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.  
ElasticSketch[16] uses, in the hash table, two packet counters (vote+ and vote*−*) instead of one in each flow record.  
Vote+ maintains the number of packets belonging to the flow,  
while vote*−* for the packets belonging to all the other flows  
that have been hashed into the same bucket. It also uses a  
count-min sketch[22] to maintain summarized flow records,  
corresponding to both flows that have never been stored in  
the hash table, and flows that have been evicted when the  
corresponding vote*−* is too large, i.e., *vote vote−* + is greater than  
a predefined threshold *λ*. However, due to collisions and the  
eviction strategy ElasticSketch employs, 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.  
FlowRadar[2] uses a bloom filter[23] to determine whether  
a new flow comes in. It also adds a flow count and a flow set  
field to each flow record in the hash table. For each packet,  
FlowRadar hashes it into multiple buckets, and updates the  
corresponding packet count fields. If a new flow comes in  
as reported by the bloom filter, the flow set fields of the  
buckets will be “xor”-ed with the new ID, and the flow count  
fields will be incremented by 1. During post processing phase,  
FlowRadar can decode (some) flow IDs that are encoded into  
the flow set fields, and also recover the corresponding packet  
counts. 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 “soft 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 [24] 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 (**M**) and an ancillary table (**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 Sec  
II. In the main table **M**, flow ID will be used as *key*, while  
in the ancillary table **A**, a digest of flow ID will be used as  
*key*. We have a set of *d* + 1 independent hash functions, i.e.,  
*h*1*, h*2*, · · · , hd*, and *g*1, where *d* is a positive integer (we call  
*d* the *depth* of **M**, and typically *d* = 3). Each hash function  
*hi*(1 *≤ i ≤ d*) randomly maps a flow ID to one bucket in **M**,  
while *g*1 maps the ID to one bucket in **A**. A digest can be  
generated from the hashing result of the flow ID with any *hi*.  
When a packet arrives, HashFlow updates **M** and **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.*flow\_id) in the  
main table **M**. If **M**[*idx*] is empty, we just put the flow ID  
and a count of 1 in the bucket (line 5*∼* 6). If the bucket is  
already occupied by packets of this flow earlier, we just simply  
increment the count by 1 (line 7 *∼* 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*, · · · , hd* 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 **A**. Here the action is more  
intrusive, as an existing flow will be replaced (discarded) if it  
collides with the new arrival (line 16 *∼* 19).  
2) **Record promotion.** If, in the ancillary table **A**, *p*  
succeeds to find the right bucket to reside, then it updates the  
packet count field of the record. However, if the corresponding  
flow record keeps growing and the packet count becomes large  
enough, then we will promote the record by re-inserting it to  
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 *∼* 11). When a flow record in the ancillary table should be  
promoted, it will replace this sentinel we have kept in mind  
(line 22 *∼* 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.  
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 a bucket indexed at *h*1(*f*1), where  
the record(*f*1*,* 5) has the same key, and the counter is simply  
incremented. When a packet of flow *f*2 arrives, *h*1 maps it  
into an empty bucket, so the record becomes (*f*2*,* 1). When a  
packet of flow *f*5 arrives, it collides with the record (*f*4*,* 4) in  
the bucket indexed at *h*1(*f*5). Then we try to resolve collision  
with *h*2, but again, the packet collides with the record (*f*6*,* 10)  
in the bucket at *h*2(*f*5). So we have to use *g*1 to find a place  
in the ancillary table for *p*. Sadly, it collides again with the  
record (*f*3*,* 8), and we let it replace the existing one. The last  
packet is from flow *f*8, and it goes through a similar process  
to that of *f*5. The difference is that, at last, this packet finds  
its corresponding flow record of (*f*8*,* 7) in the ancillary table,  
and the record becomes elephant (as the sentinel flow with the  
smallest packet count in the main table has a packet count of  
7). So we promote (*f*8*,* 8) by inserting it back into the main  
table, evicting the sentinel one.  
In Algorithm 1, we use multiple independent hash functions  
*hi*(*i* = 1*,* 2*, · · · , d*) in a single main table **M**. Another choice  
is to use multiple small hash tables **M***i*, each of which  
corresponds to an independent hash function *hi*. Algorithm  
1 can be modified straightforwardly: to update the *i*-th small  
**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 ← hi*(*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*1(*flowID*)  
15: *digest ← h*1(*flowID*)%(2digest width)  
16: **if A**[*idx*]*.count* == 0 or **A**[*idx*]*.key 6*= *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**  
*f1*  
*f5*  
*f8*  
*h1(.)*  
*h1(.)*  
*h1(.)*  
**Main Table**  
*(flowID, count)*  
*(f1, 6)*  
*(pos1, min=9)*  
*h2(.)*  
*g1(.)*  
*(pos2, min=7)*  
**Ancillary Table**  
*(digest, count)*  
*(f8, 8) 7==min=7*  
*h2(.)*  
*g1(.)*  
*(f5, 1)*  
*f2 h1(.) (f2, 1)*  
*(f1, 5)*  
*(f6, 10)*  
*(f4, 4)*  
*(f7, 9)*  
*...*  
*(f9,7)*  
*...*  
*(f3, 8)*  
*...*  
*...*  
*(f8,7)*  
Fig. 1. An example of HashFlow  
table **M***i* instead of **M** (line 5 *∼* 12), to remember which  
small table the sentinel record resides in (line 10 *∼* 11), and  
to evict the sentinel record in the right small table (line 22 *∼*  
23). In addition, we introduce a weight *α*(0 *< α <* 1), such  
that the number of buckets in **M***i*+1 is *α* times that in **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 **M**. We first analyze  
the case where a multi-hash table is used for **M**, then the case  
where pipelined tables are used. In either case, we assume that  
there are *m* distinct flows fed into **M**, 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[24].  
After inserting *m*1 = *m* flows randomly into *n* buckets, the  
probability that a given bucket is empty is  
*p*1 = (1 *−* 1  
*n*  
)*m*1 *≈ e−*  
*m*1  
*n*  
*,*  
and the utilization of the table is *u*1 = 1 *− p*1 = 1 *− e−*  
*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 **M** after this round is *m*1 *− n ×* (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 **M** into the table  
again, but this time with *h*2. Assume **M** is empty before the  
second round starts, then after the *m*2 = *m*1 *− n ×* (1 *−*  
*p*1) flows left by the first round have been inserted in the  
second round, a bucket will be empty with probability *e−*  
*m*2  
*n*  
.  
However, **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 *≈ p*1 *× e−*  
*m*2  
*n* , and the number of flows that have not been  
inserted into **M** will be *m*3 = *m*1*−n×*(1*−p*2). The utilization  
of **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, *mk* flows  
are fed into a hash table with a new hash function *hk*, where  
there are already *n ×* (1 *− pk−*1) buckets being occupied in  
the previous rounds. Then after the *k*-th round, the probability  
that a bucket is empty is  
*pk ≈ pk−*1 *× e−*  
*mk*  
*n*  
= *pk−*1 *× e−*  
*m*1*−n×*(1*−pk−*1)  
*n*  
= *pk−*1 *× e*1*− m n*1 *−pk−*1  
= *pk−*1 *× e*1*− m n −pk−*1 (1)  
for *k ≥* 2. With Equation (1), for any given *d*, *m*, and *n*, we  
can recursively compute the probability *pd* that a bucket is  
empty in the hash table after *d* rounds. Then the utilization of  
the hash table will be 1 *− pd*. We note that there is a slight  
difference between this model and our multi-hash table, as  
will be shown later.  
(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  
**Pipelined tables.** Let *nk* be the number of buckets in the  
*k*-th table **M***k* such that *nk*+1 = *α × nk*, 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 *pk*, *mk* and *uk* as those in the first  
model. Since P*d k*=1 *nk* = P*d k*=1(*αk−*1 *× n*1) = 11 *− −α α d × n*1,  
we get *n*1 = 11 *− −α α d × n*, and *nk* = *αk−*1 *×* 11 *− −α α d × n*.  
The first round works exactly the same as that in the  
previous model, so we get *p*1 = (1 *− n* 1 1 )*m*1 *≈ e−*  
*m*1  
*n*1  
,  
*u*1 = 1 *− p*1, and *m*2 = *m*1 *− n*1 *×* (1 *− p*1).  
For the *k*-th round, we know *mk* flows are to be fed into  
the table **M***k* with *nk* buckets, so we get *pk ≈ e−*  
*mk*  
*nk* , and the  
number of flows left after this round is  
*mk*+1 = *mk − nk ×* (1 *− pk*)*.* (2)  
Dividing both sides of Equation (2) by *nk*+1, we get  
*mk*+1  
*nk*+1  
=  
*nk*  
*nk*+1  
*×*  
*mk − nk ×* (1 *− pk*)  
*nk*  
= *α−*1 *× m nk k −* 1 + *pk* *,*  
which is just  
*−* ln *pk*+1 = *α−*1 *×* (*−* ln *pk −* 1 + *pk*)*.* (3)  
From Equation (3), we finally get  
*pk*+1 = (*pk*) *α* 1 *× e* 1*− α pk .* (4)  
With Equation (4), for any given *d*, *m*, and *n*, we can  
recursively compute the probability *pk*(1 *≤ k ≤ d*) that a  
bucket is empty in the *k*-th hash table. Then the utilization of  
the pipelined tables will be  
P*d k*=1(*nk ×* (1 *− pk*))  
P*d k*=1 *nk* = 1*−*  
1 *− α*  
1 *− αd ×*  
*d*X *k*  
=1  
(*αk−*1*×pk*)*.* (5)  
Now we will show how accurate the models are. In Fig.  
2(a), we compare the utilization provided by our multi-hash  
table model against the results from simulations on some real  
traces. 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, only under a light load of *m/n* = 1, there is  
a slight difference between the model and the real algorithm.  
When *m/n ≥* 2, the multi-hash table model provides nearly  
perfect predictions.  
Fig. 2(b) and Fig. 2(c) depict the utilization provided by our  
model on pipelined tables, as well as results from simulations,  
for traffic load *m/n* = 1*.*0 and 2.0, respectively. We use a  
similar setting as above, with *n* =100K and *d* from 1 to 10,  
but we vary the pipeline weigh *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 compute the utilization of  
our main table, as long as the traffic load *m/n* is known.  
Since each record is accurate (neglecting the minor chance  
that a flow record promoted back to the main table happens to  
have an inaccurate count), this provides a concrete prediction  
on the number of records HashFlow can report. We can see  
more hash functions will improve the utilization. For example,  
in the case of *m/n* = 1, the utilization increases from 63% to  
80% 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 *α* = 0*.*7 and *m/n* = 1,  
up to 5.5% more utilization can be achieved. Our evaluation  
will adopt the pipelined scheme, where *α* = 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[20]. The code can  
be found at [25]. Although both versions of the algorithm  
are implemented using P414, 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 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. As an action of P4 cannot  
access more than one register array, we implement 6 tables  
to access a flow record. Since accesses of the IP addresses,  
protocol and ports are independent, the tables corresponding  
to these arrays 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 *× d* + 2 stages and only *d ≤* 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 resource (e.g., a register array storing the  
packet counts) 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, when  
doing collision resolution, we may have to visit the hash table  
*d* times, 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 the hash 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 tables in a factor of 0.7, is infeasible. We simply  
set the size of each table to be the same in our implementation.  
Another challenge in implementing HashFlow is that record  
promotion requires to revisit one of the 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 and set up a new flow for this packet. Since  
more packets than that inputted 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  
sacrifice in 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[2], HashPipe[15] and ElasticSketch[16], in bmv2  
[18], which is a software switch with P4 [17] programmability.  
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 stated in  
Section IV, we have implemented HashFlow in a P4 hardware  
switch[20] and it functions well. However, our implementation  
is strongly constrained due to the resource limitation and  
HashPipe cannot be implemented in a hardware switch either,  
as illustrated in [26], 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  
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.  
Fig. 3. Flow size distribution of  
the traces used for evaluation  
Fig. 4. Flow size estimation under  
different pipeline depth  
*• 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* = num. of flow records with complete flow IDs  
*n*  
*.*  
Notice that HashPipe, ElasticSketch and HashFlow maintains flow records individually, while FlowRadar can  
decode flow records from a coded flow set.  
*• 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* = 1  
*n*  
X  
 estimated size of flow real size of flow *i i −* 1 *.*  
Notice that, FlowRadar can decode flow sizes from a  
coded flow set, while ElasticSketch can use an additional  
count-min sketch to estimate flow sizes.  
*• 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  
F1 Score =  
2 *· PR · RR*  
*PR* + *RR ,*  
where *PR* = *c*  
*c*1  
and *RR* = *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* =  
 estimated number of flows *n −* 1 *.*  
Notice that, linear counting[29] is used by ElasticSketch  
to estimate the number of flows in its count-min sketch,  
and used by HashFlow to estimate the number of flows  
in its ancillary table.  
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*×* 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, while the weight *α* is 0.7 unless  
otherwise stated. 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  
Acronyms Algorithm Acronyms Algorithms  
HF HashFlow HP HashPipe  
ES ElasticSketch FR FlowRadar  
*B. Optimizing the Main 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.  
We plot the *Flow Set Coverage (FSC)* for flow record report  
in Fig. 5(a), and plot the *Average Relative Error (ARE)* for  
flow size estimation in Fig. 5(b), where there are 3 pipelined  
tables, and the weight is 0.6, 0.7 and 0.8 respectively. The  
traces are from the campus network, and as the number of  
flows increases from 10K to 60K, the *FSC* decreases, while  
the *ARE* increases slowly. It can be seen that using pipelined  
tables with a weight around *α* = 0*.*7 achieves the best  
result. Compared with a multi-hash table, pipelined tables can  
improve the *FSC* by 3.1%, and reduce 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 plot the *Average Relative Error (ARE)* for flow  
size estimation of 50K flows, when 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 ancillary table has upon  
HashFlow, we define a parameter *β* and *n*2 = *n*1*×β*, where *n*1  
and *n*2 are number of buckets in the main table and ancillary  
table respectively. A larger value of *β* implies that we have to  
allocate more memory to the ancillary table. Fig. 6 shows that  
normally *β* = 0*.*25 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 attacks, in the following experiments we set *β* to  
1.0 by default.  
(a) CAIDA (b) HGC  
Fig. 6. ARE of flow size estimation 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 depicts the *Flow Set Coverage (FSC)* for flow record  
report achieved by these algorithms. We can see HashFlow  
nearly always performs better than the others. 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 how accurate these algorithms can estimate the  
flow sizes. HashFlow often achieves a much lower estimation  
error than its competitors. 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, where the threshold  
for a flow to be treated as a heavy hitter varies. We also  
measure their *ARE* when estimating the 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 *∼* 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 *∼* 0.2 and  
0.2 *∼* 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, even if the throughput on a software switch is not  
convincing, 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 P4 switch, there will not be a single action bottlenecking  
the processing of packets, and the packets will be processed  
in linear speed when the algorithm is correctly implemented.  
The only risk is that some packets will be resubmitted, so  
(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*  
the switch will have to process more packets than that is  
transmitted through it, degrading the throughput. To evaluate  
the possible loss of throughput, we calculate the *Increase*  
*in Processing* introduced by resubmission operation in Hash-  
(a) CAIDA (b) Campus Network (c) ISP1 (d) ISP2  
Fig. 11. *Average Relative Error(ARE)* for *Heavy Hitter Detection*  
Flow. Suppose there are *n*1 packets arrives at the switch, and  
there are *n*2 packets processed by the switch where *n*2 *≥ n*1.  
The *Increase in Processing* is defined as *increase* = *n*2  
*n*1  
*−* 1*.*  
As shown in Fig. 12(d), the increase in processing is  
normally around 2%, and it is no more than 6.7% in the  
trace provided by ISP2, which is far less skewed than the  
real network traffic. So HashFlow will achieve very good  
performance in the sense of throughput.  
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 clear better performance in  
nearly all cases. This is due to its high utilization of memory  
with only few operations. In the future, we plan to study  
how to make it adaptive to traffic variation and network wide  
measurement.  
REFERENCES  
[1] B. Claise, “Cisco Systems NetFlow Services Export Version 9,” 2004.  
[Online]. Available: https://tools.ietf.org/html/rfc3954  
[2] 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*, ser. NSDI’16, 2016,  
pp. 311–324.  
[3] “Access Time of DRAM and SRAM.” [Online]. Available: https:  
//www.webopedia.com/TERM/A/access\_time.html  
[4] “Sampled NetFlow.” [Online]. Available: https://www.cisco.com/c/en/  
us/td/docs/ios/12\_0s/feature/guide/12s\_sanf.html  
[5] N. Hohn and D. Veitch, “Inverting Sampled Traffic,” in *Proceedings*  
*of the 3rd ACM SIGCOMM Conference on Internet Measurement*, ser.  
IMC ’03, 2003, pp. 222–233.  
[6] 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.  
[7] P. Tune and D. Veitch, “Towards Optimal Sampling for Flow Size  
Estimation,” in *Proceedings of the 8th ACM SIGCOMM Conference*  
*on Internet Measurement*, ser. IMC ’08, 2008, pp. 243–256.  
[8] N. Duffield, “Sampling for Passive Internet Measurement: A Review,”  
*Statist. Sci.*, vol. 19, no. 3, pp. 472–498, 08 2004.  
[9] 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.  
[10] S. Muthukrishnan, “Data Streams: Algorithms and Applications,”  
*Found. Trends Theor. Comput. Sci.*, vol. 1, no. 2, pp. 117–236, Aug.  
2005.  
[11] 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*, ser. SIGCOMM ’17,  
2017, pp. 113–126.  
[12] M. Chen, S. Chen, and Z. Cai, “Counter Tree: A Scalable Counter Architecture for Per-Flow Traffic Measurement,” *IEEE/ACM Transactions*  
*on Networking (TON)*, vol. 25, no. 2, pp. 1249–1262, 2017.  
[13] 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*, ser. NSDI’13, 2013,  
pp. 29–42.  
[14] 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*,  
ser. SIGCOMM ’16, 2016, pp. 101–114.  
[15] 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*, ser. SOSR ’17, 2017,  
pp. 164–176.  
[16] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao,  
X. Li, and S. Uhlig, “Elastic Sketch: Adaptive and Fast NetworkWide Measurements,” in *Proceedings of the Conference of the ACM*  
*Special Interest Group on Data Communication*, ser. SIGCOMM’18,  
2018, p. 14.  
[17] 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.  
[18] “bmv2: P4 Software Switch,” 2018. [Online]. Available: https:  
//github.com/p4lang/behavioral-model  
[19] “Barefoot Tofino: World’s Fastest P4-Programmable Ethernet Switch  
ASICs.” [Online]. Available: https://barefootnetworks.com/  
[20] “Edgecore Networks.” [Online]. Available: https://www.edge-core.com/  
productsInfo.php?cls=1&cls2=180&cls3=181&id=335  
[21] R. Pagh and F. F. Rodler, “Cuckoo Hashing,” *Journal of Algorithms*,  
vol. 51, no. 2, pp. 122–144, May 2004.  
[22] G. Cormode and S. Muthukrishnan, “An improved data stream summary: the count-min sketch and its applications,” *Journal of Algorithms*,  
vol. 55, no. 1, pp. 58–75, Apr. 2005.  
[23] B. H. Bloom, “Space/Time Trade-offs in Hash Coding with Allowable  
Errors,” *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.  
[24] 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.  
[25] Z. Zhao, “HashFlow Implemented in P4,” Dec. 2018. [Online].  
Available: https://github.com/xinshengzzy/HashFlow  
[26] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, “Efficient  
Measurement on Programmable Switches Using Probabilistic Recircula-  
(a) Throughput (b) Number of hash operations (c) Number of memory access (d) Increase in processing access  
Fig. 12. Throughput, Hash Operation, Memory Access and Increase in Processing  
tion,” in *2018 IEEE 26th International Conference on Network Protocols*  
*(ICNP)*, ser. ICNP’18, Sep. 2018, pp. 313–323.  
[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*, ser. IMC ’10, 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 (TODS)*, vol. 15, no. 2, pp. 208–  
229, 1990.