A SDN Soft Computing Application for Detecting Heavy Hitters

Yi-Bing Lin, *Fellow, IEEE,* Ching-Chun Huang, and Shi-Chun Tsai, *Senior Member, IEEE,*

***Abstract*—To avoid DDoS attacks or real-time TCP incast in the software-defined networking (SDN) environment, the HashPipe algorithm was developed following the space saving approach. Unfortunately, HashPipe implemented in the behavioral model (bmv2) cannot be directly executed at a real P4 switch due to P4 pipeline limitation. Based on the Banzai machine model, this paper shows how to smartly utilize the Banzai atoms to develop HashPipe as a soft computing application in a real P4 switch. Then we propose an enhanced HashPipe algorithm that significantly improves the accuracy of the original HashPipe. The proposed heavy hitter detection is executed at the line-rate of the Tofino P4 switch with the highest process rate in the world.**

***Index Terms*—HashPipe, heavy hitter detection, P4, software-defined networking, space saving**

I. Introduction

DDOS attacks or real-time TCP incast result in outliers of traffic in a data network. To identify traffic outliers, "heavy hitter" detection was proposed [1], [2]. Such detection identifies the flows exceeding a pre-determined threshold, and is often conducted by tracking the packet destinations of a large number of distinct sources.

Heavy hitter detection is typically achieved by analyzing packet samples or flow logs, which consumes significant amount of time to count the numbers of packets for flows, and large memory space to store flow logs. For example, heavy hitter detection may consume memory space in the order of tera bits, and many hours are spent to count the number of packets for each flow every day in a network with 100 GB traffic flows. Therefore, the above process is conducted offline, and the network administrator cannot give prompt responses based on the events observed from heavy hitter detection.

By analyzing packet samples [3], [4], heavy hitter detection can be sped up. This approach either samples some flows to reduce the process overhead, or samples all flows and saves the temporary results in the SRAM. The former is not accurate and the latter consumes significant amount of storage. Therefore, reasonable accuracy may not be achievable under acceptable time and space budgets. For example, packet sampling through NetFlow has been widely deployed, but the CPU and bandwidth overheads prohibit processing the sampled packets at a reasonable rate (say, sampling one in 1000 packets). Alternatively, sketches [5], [6] are used to hash and count all packets by switch hardware. This approach does not record the flow identities, and therefore the accumulated counts cannot be mapped back to specific flows.

To resolve the above issues, the concept of space saving [7], [8], [9] was proposed, which has been identified as the most accurate and efficient algorithm for computing heavy hitters with strong theoretical error guarantees. Space saving is a counter-based algorithm for estimating item frequencies. The algorithm tracks a subset of items from the universe to maintain an approximate count for each item in the subset. Specifically, the input of the algorithm is a stream of pairs (id, c) where id is the identity of an item and $c > 0$ is the frequency count for that item. The algorithm tracks a set T of items and their counts. If the next item id in the stream is found in T, its count is updated appropriately. Otherwise, the item with the smallest count in T is removed and replaced by id, and the count for id is set to the count value of the item plus c. Previous studies [8], [9] indicated that if T is large enough, all heavy hitters will appear in the final set.

This paper considers the heavy hitter issue for software-defined networking (SDN). Compared with the traditional data network, SDN facilitates network management and programmatically enables efficient network configuration to improve network performance and monitoring. SDN centralizes network intelligence in one network component by disassociating the forwarding process of network packets (data plane) from the routing process (control plane). That is, unlike the traditional data network, the control plane and the data plane are separated in SDN. The controller in the control plane is responsible for giving instruction to the switches in the data plane. On the other hand, the SDN switches provide high-speed processing for packet forwarding. Many network functions in traditional networks have been implemented differently in SDN, and their time and space complexities have been investigated [10].

To support heavy hitter detection in SDN, one may implement the space saving algorithm in the SDN controller. To do so, the data traffic will pass through the controller, which violates the original design goals of SDN. It is more reasonable to implement heavy hitter detection at the SDN switch. However, the switch is not designed for developing the soft computing applications such as heavy hitter detection. To support this detection function, programmability of the data plane becomes essential SDN switch features. Such features can be provided by the Programming Protocol-Independent Packet Processor (P4) technology [11]. With P4, the control intelligence can be shifted from the SDN controller to the SDN

Yi-Bing Lin, Ching-Chun Huang, and Shi-Chun Tsai are with the Department of Computer Science, National Chiao Tung University, Taiwan (e-mails: liny@nctu.edu.tw, d3350233.cs05g@nctu.edu.tw, sctsai@cs.nctu.edu.tw). This work was supported in part by Ministry of Science and Technology (MOST) 106N490, 106-2221-E- 009-049-MY2, 107-2221-E-009-039, "Center for Open Intelligent Connectivity" of National Chiao Tung University and Ministry of Education, Taiwan, R.O.C.

Manuscript received 18 Mar, 2019; revised 10 Feb, 2019.

switch, and the switch becomes a platform for developing soft computing applications.

In SDN, a switch uses a set of match-action tables to apply rules for packet processing, and P4 provides an efficient way to configure the packet processing pipelines. A P4 program describes how the header fields of a packet are parsed and processed by using the match-action tables, where the matched operations may modify the packet header fields or the content of the switch's metadata and registers. A packet arriving at the P4 switch is processed through the parser, the ingress pipeline, the traffic manager (queues), the egress pipeline and the deparser. The parser of the P4 switch extracts the header fields and the payload of an incoming packet following the parse graph defined in the P4 program. Then the ingress pipeline consisting of match-action tables arranged in the stages of the pipeline manipulates the packet headers and generates an egress specification that determines the set of ports to which the packet will be sent. At the traffic manager, the packets are queued before being sent to the egress pipeline. At the egress pipeline, the packet header may be further modified. At the deparser, the headers are assembled back to a well-formed packet.

Besides packet forwarding, a P4 switch allows to create and modify algorithmic states in the switch as part of packet processing following, for example, the Banzai machine model [12]. Banzai targets for programmable line-rate switches, which extends recent programmable switch architectures with stateful processing units to implement data-plane algorithms. Stateful memories (counters, meters and registers) persist across packets and stateless memories (packet headers and metadata) are used per-packet state. Banzai relaxes two major constraints to support stateful line-rate operations: the inability to share switch state between different packet-processing units, and any state modifications are not visible to the next packet entering the switch. To free these constraints, multiple processing units called *atoms* are used to model atomic operations that are natively supported by a programmable line-rate switch. Banzai provides a stateful atomic operation at line rate with a stateful atom to read, modify, and write back in a single stage within one clock cycle. Each pipeline stage in Banzai consists of a vector of atoms. These atoms modify mutually exclusive sections of the same packet headers in parallel at every clock cycle.

Based on the above description, it is clear that the P4 pipeline architecture is very different from the general CPU architecture, and space saving algorithm cannot be implemented directly on the above P4 pipeline architecture. Although the space saving algorithm only updates one counter per packet, it requires finding the item with the minimum count value in the table. Unfortunately, data structures such as sorted linked lists or priority queues are not available at the P4 switch, and therefore the function for scanning the entire table to find the minimum count is not directly supported by the P4 switch. To develop effective heavy hitter detection within the P4 switch architecture constraints, [13] proposed the HashPipe algorithm by modifying the general space saving algorithm. HashPipe was successfully implemented in the P4 behavioral model (bmv2), but cannot be directly executed in a real P4 switch. In order to maintain high packet-processing throughput, the real switches based on Tofino chip follow a more restrictive computational model that limits read/write access. In this paper, we develop HashPipe as a soft computing application in a real P4 switch. To further improve the accuracy of heavy hitter detection, we propose enhanced HashPipe (EHP), a modification to HashPipe that reduces the number of hash operations with the same memory budget.

Also, [13] conducted the performance for HashPipe with different numbers of hash tables, but did not mention the effects of the ratio of the table sizes. In this paper, we find the optimal table sizes through both experiments and analytic analysis. The paper is organized as follows. Section II describes the HashPipe algorithm implemented in bmv2. Section III shows the Banzai machine model and its implementation of the HashPipe algorithm with 2 hash tables in a real P4 switch. Section IV proposes an enhanced HashPipe algorithm and shows the implementation with 3 hash tables and its enhancement. Section V evaluates the error rates for HashPipe as a soft computing application in the P4 switch, and shows that 3 hash tables are enough to produce good results for HashPipe.

II. HashPipe Algorithm

With HashPipe, a P4 switch identifies the packet flows (the items) that generate heavy traffic. HashPipe uses multiple match-action tables in the switch pipeline for updating each hash table. In the HashPipe algorithm, each match-action table has a single default action for every packet. The hash tables are implemented in the register memory that persist across successive packets. Through the rules of the match-action tables, the HashPipe program manipulates on the hash tables in the P4 switch.

Like the space saving algorithm, HashPipe maintains both a subset of the flow identifiers called "keys" (i.e., the flow ids) and their counts in N hash tables $T_1, T_2, \ldots, T_N$, where T_n is the n-th hash table for $1 \leq n \leq N$. Ideally, the flow ids saved in the tables will be the identifiers of the heaviest flows. Fig. 1 lists the HashPipe algorithm implemented in bmv2 [13]. Suppose that every packet belongs to a flow with the identity id. At Lines 1-3 of Algorithm HashPipe (Fig. 1), when a packet of flow id arrives at the P4 switch, the switch stores id into the $mHvH$ metadata (where "m" means metadata and "HvH" means heavy hitter). Then $mHvH.id$ is hashed to a memory slot l in the hash table T_1. If $T_1(l).id = mHvH.id$ (a hit at Line 4) or if the location is not occupied at Line 6, the count $T_1(l).c$ is incremented by 1 at Line 5 and is set to 1 at Line 8. If there is a miss, i.e., $T_1(l).id \neq mHvH.id$, the id stored at the l-th slot is replaced by the new flow id (Lines 9-14). Let the replaced key and its count at T_n be id_n and c_n, respectively. In Lines 9-14, $n = 1$. The P4 switch stores id_n and c_n into the $mHvH$ metadata. In the downstream hash table T_{n+1} (Lines 15-28), the flow id_n and its count c_n just evicted are carried in $mHvH$ along with the packet. For $n \geq 2$, the carried flow id_{n-1} is looked up in the current hash table T_n; i.e., l is set to $Hash_n(id_{n-1})$ at Line 16. Between the flow id looked up in the hash table (i.e., id_n is $T_n(l).id$)

and the carried id_{n-1}, the flow id with the larger count is retained in the hash table (Lines 18, 20-21, and 25-26), while the other is either carried along with the packet to the next hash table (Lines 27 and 28), or is removed from the switch if the packet has arrived at the last hash table. From the description of Algorithm HashPipe, the P4 pipeline operations are used to continuously sample locations in the hash tables, and find the top heavy packet flows with limited available memory (a limited number of hash tables).

Line	**Algorithm: HashPipe (bmv2) with N hash tables**
1	$mHvH.id \leftarrow id$
2	$mHvH.c \leftarrow 1$
3	$l \leftarrow Hash_1(mHvH.id)$
4	**if** $T_1(l).id = mHvH.id$ **then**
5	$T_1(l).c \leftarrow T_1(l).c + 1$
6	**else if** $T_1(l).id = 0$ **then** {
7	$T_1(l).id \leftarrow mHvH.id$
8	$T_1(l).c \leftarrow mHvH.c$
	} **else** {
9	$tmp\_id \leftarrow T_1(l).id$
10	$tmp\_c \leftarrow T_1(l).c$
11	$T_1(l).id \leftarrow mHvH.id$
12	$T_1(l).c \leftarrow mHvH.c$
13	$mHvH.id \leftarrow tmp\_id$
14	$mHvH.c \leftarrow tmp\_c$
15	**for** $n \leftarrow 2$ **to** N **do** {
16	$l \leftarrow Hash_n(mHvH.id)$
17	**if** $T_n(l).id = mHvH.id$ **then** {
18	$T_n(l).c \leftarrow T_n(l).c + mHvH.c$ **break**
19	} **else if** $T_n(l).id = 0$ **then** {
20	$T_n(l).id \leftarrow mHvH.id$
21	$T_n(l).c \leftarrow mHvH.c$ **break**
22	} **else if** $T_n(l).c < mHvH.c$ **then** {
23	$tmp\_id \leftarrow T_n(l).id$
24	$tmp\_c \leftarrow T_n(l).c$
25	$T_n(l).id \leftarrow mHvH.id$
26	$T_n(l).c \leftarrow mHvH.c$
27	$mHvH.id \leftarrow tmp\_id$
28	$mHvH.c \leftarrow tmp\_c$ }}}

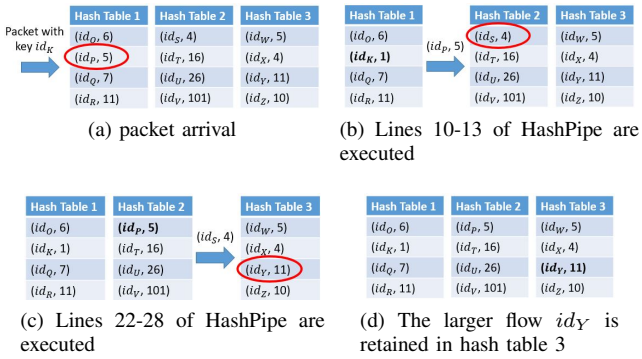
Fig. 1. Algorithm HashPipe for bmv2.

Fig. 2 illustrates an example of processing a packet using HashPipe with 3 hash tables. A packet of flow id_K enters the switch pipeline (Fig. 2a), and is not found in the first table T_1. The flow id and the count pair (id_K, 1) for flow id_K are inserted in T_1 (Fig. 2b). Flow id_P (that was previously stored in the slot currently occupied by id_K) is carried by the packet to the next hash table, and is hashed to the slot containing flow id_S. Since the count of id_P is larger than that of id_S, id_P is stored in T_2 and id_S is carried by the packet to the next hash table (Fig. 2c). At T_3, the count of id_Y (where id_S is hashed to id_Y's slot) is larger than that of id_S. Therefore, id_Y stays in T_3 (Fig. 2d). After the packet with the flow id_K is processed, flow id_K is inserted in the table, and the flow with the minimum count among id_P, id_S and id_Y (which is id_S) is removed.

The algorithm in Fig. 1 is nicely implemented in the bmv2 model, but cannot be directly executed in a real P4 switch. In the subsequent sections we show how to implement Algorithm HashPipe on a real line-rate P4 switch.

III. HashPipe with 2 Hash Tables

This section illustrates the Banzai machine model for HashPipe implementation in a real P4 switch. We implement a



(a) packet arrival

(b) Lines 10-13 of HashPipe are executed

(c) Lines 22-28 of HashPipe are executed

(d) The larger flow id_Y is retained in hash table 3

Fig. 2. An illustration of the HashPipe execution with 3 hash tables.

2-Table HashPipe program abbreviated as HP-2. The P4 compiler compiles HP-2 to an atom pipeline for the Banzai machine [12]. An atom corresponds to an action in a match-action table, which contains a stateless or a stateful variable, and a digital circuit modifying this variable. In [12], eight atom types in the Banzai machine are introduced. In this paper, we use five types of atoms, including Stateless, ReadAddWrite (RAW), Predicated RAW (PRAW), IfElseReadAddWrite (IfElseRAW) and Paired updates (Pairs). A Stateless atom updates a stateless variable with arithmetic, logic, relational and conditional operations. A RAW adds or writes (but not both) a packet field/constant to a stateful variable. A packet field can be a metadata field or a header field. A PRAW atom executes a RAW on a stateful variable only if a predicate is true; otherwise it does nothing. An IfElseRAW atom executes one of two RAWs on a stateful variable. If a predicate is true, a RAW is executed; otherwise it executes another RAW on that variable. A Pair atom executes one IfElseRAW nested in another IfElseRAW to provide 4-way predication. The Pair atom manipulates on two stateful variables and a stateless variable. Other stateful atoms (RAW, PRAW and IfElseRAW) can update a stateful variable and a stateless variable.

Figs. 3-8 illustrate our implementation, which consist of stateless atoms (Stateless; represented by the white oval rectangle icons) and stateful atoms (PRAW, IfElseRAW and Pairs; represented by the gray oval rectangle icons). A packet arriving at a line-rate P4 switch (Fig. 3(1)) is parsed by a programmable parser (Fig. 3(2)). The parser extracts the packet header fields (Fig. 3(3)). These header fields are processed by the ingress pipeline through a series of match-action tables arranged in stages (e.g., hashtable1_init at Stage 1; see Fig. 3(4)). In Figs. 3-8, a dashed box represents a stage (or a part of a stage) in the ingress pipeline. Through the match-action rules, processing a packet at a stage may modify its header fields (or metadata fields) as well as some persistent states (counters, meters, or registers) at that stage.

At the first stage, HP-2 implements two Stateless atoms (Fig. 3 (5) and (7)) to modify the metadata $mHvH$ (Fig. 3(6)), where $mHvH.id$ is the flow id of the packet, $mHvH.c$ is the count for the flow id, $mHvH.f\_h$ is a flag to indicate if a hit at the hash table occurs and $mHvH.tmp\_id$ is used to temporarily store the flow id read from the hash table. Without loss of generality, we assume that HP-2 identifies a packet flow

through its source IP address.

The hashtable1_init flow table is always hit, and Stage 1 is executed where two Stateless atoms initialize $mHvH$ by storing the source IP address ($ipv4.srcAddr$) into $mHvH.id$ (Fig. 3(5)) and setting $mHvH.f\_h$ to zero (Fig. 3(7)).

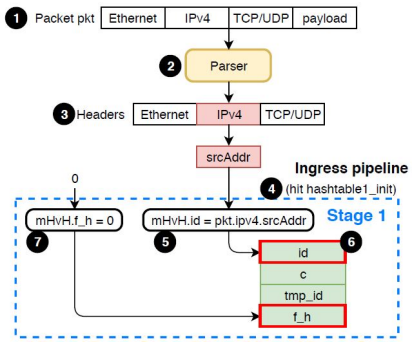


Fig. 3. HP-2 (Stage 1).

The execution for Stage 2 is illustrated in Fig. 4 with the following pseudo code.

Stage	Line	Pseudo Code	Atom
2	3	$l \leftarrow Hash_1(mHvH.id)$	PRAW
	4	$mHvH.tmp\_id \leftarrow T_1(l).id$	
	5	**if** $T_1(l).id \neq mHvH.id$ **then**	
	6	$T_1(l).id \leftarrow mHvH.id$	

The hashtable1_update_key flow table is always hit and Stage 2 is executed. A PRAW atom hashes $mHvH.id$ to a location l in T_1 at Line 3 (Fig. 4(8)). The value $T_1(l).id$ is temporarily saved in $mHvH.tmp\_id$ at Line 4 (the path from Fig. 4(12) to Fig. 5(12)). Line 5 checks if $T_1(l).id$ is not equal to $mHvH.id$ (Fig. 4(9)). If so, a miss occurs, and $mHvH.id$ is stored in $T_1(l).id$ as the new flow id at Line 6 (Fig. 4(10) and the path from Fig. 4(13) to Fig. 5(13)). Otherwise, $T_1(l).id$ remains unchanged (Fig. 4(11) and the path from Fig. 4(13) to Fig. 5(13)). At this stage, the stateless variable is $mHvH.tmp\_id$ and the stateful variable is $T_1(l).id$.

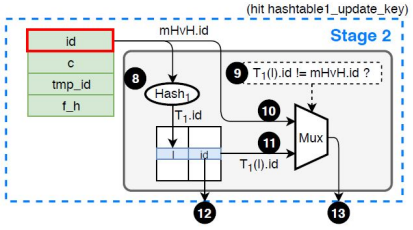


Fig. 4. HP-2 (Stage 2).

After Stage 2, the control program checks if $mHvH.tmp\_id$ is non-zero at Line 7 of the psuedo code below (see Fig. 5(14)). If so, slot l is not empty, the hashtable1_check_miss flow table is hit, and Stage 3 is executed. Otherwise, slot l is empty and Stage 3 is skipped.

Stage	Line	Pseudo Code	Atom
	7	**if** $mHvH.tmp\_id \neq 0$ **then**	
3	8	$mHvH.f\_h \leftarrow mHvH.tmp\_id - mHvH.id$	Stateless

At Line 8, Stage 3 assigns flag $mHvH.f\_h$ as $mHvH.tmp\_id - mHvH.id$ (the path from Fig. 5(15) to Fig. 6(15)). At this stage, the stateless variable is $mHvH.f\_h$.

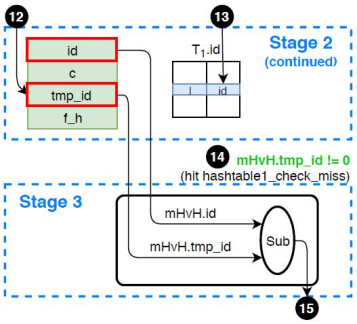


Fig. 5. HP-2 (Stage 3).

The pseudo code for Stage 4 is listed below, and the execution flow is illustrated in Fig. 6.

Stage	Line	Pseudo Code	Atom
4	9	$l \leftarrow Hash_1(mHvH.id)$	IfElseRAW
	10	$mHvH.c \leftarrow T_1(l).c$	
	11	**if** $mHvH.f\_h = 0$ **then**	
	12	$T_1(l).c \leftarrow T_1(l).c + 1$	
	13	**else**	
	14	$T_1(l).c \leftarrow 1$	

The hashtable1_update_count flow table is always hit, and at Stage 4, an IfElseRAW atom hashes $mHvH.id$ to l (Line 9). The value $T_1(l).c$ is temporarily stored in $mHvH.c$ at Line 10 (the path from Fig. 6(19) to Fig. 7(19)). Line 11 checks if $mHvH.f\_h$ is equal to zero (Fig. 6(16)). If so, a hit occurs or the location is not occupied, and $T_1(l).c$ is incremented by 1 at Line 12 (Fig. 6(17) and the path from Fig. 6(20) to Fig. 7(20)). If $mHvH.f\_h \neq 0$ (a miss occurs), $T_1(l).c$ is set to 1 at Line 14 (Fig. 6(18) and the path from Fig. 6(20) to Fig. 7(20)). The replaced flow id and its count at T_1 are stored in $mHvH.tmp\_id$ and $mHvH.c$, respectively. At this stage, the stateless variable is $mHvH.c$, and the stateful variable is $T_1(l).c$. If the replaced id and count pair is carried along with the packet to be processed with the second hash table T_2, then the following stages are executed.

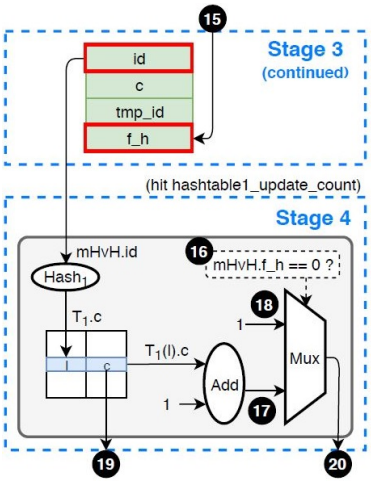


Fig. 6. HP-2 (Stage 4).

The control program checks if $mHvH.f\_h$ is non-zero at Line 15 of the pseudo code below (see Fig. 7(21)). If so, a miss occurs, the hashtable2_init flow table is hit, and both Stage 5 and Stage 6 are executed. Otherwise, these two stages are skipped.

Stage	Line	Pseudo Code	Atom
	15	**if** $mHvH.f\_h \neq 0$ **then**	
5	16	$mHvH.id \leftarrow mHvH.tmp\_id$	Stateless

At Line 16, Stage 5 uses a Stateless atom to store the carried flow id into $mHvH.id$ (the path from Fig. 7(22) to Fig. 8(22)). At this stage, the stateless variable is $mHvH.id$.

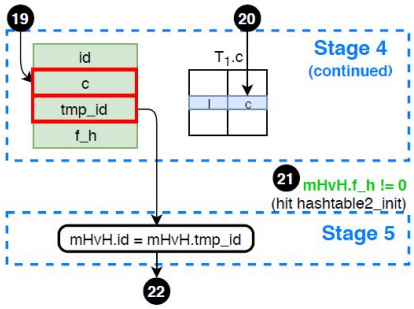


Fig. 7. HP-2 (Stage 5).

After Stage 5 is executed, the hashtable2_update table is always hit and Stage 6 is executed. At Stage 6, one of the following three conditions is met:

- C1. $T_2(l).id = mHvH.id$ ($T_2(l).id$ is equal to the carried id)
- C2. $T_2(l).id \neq mHvH.id$ and $T_2(l).c < mHvH.c$ ($T_2(l).c$ is smaller than the carried count)
- C3. $T_2(l).id \neq mHvH.id$ and $T_2(l).c \geq mHvH.c$

Since a Pairs atom can operate on two stateful variables (i.e., $T_2(l).id$ and $T_2(l).c$) and obtain the compared results to determine which one of C1-C3 occurs, we can update $T_2(l).id$ and $T_2(l).c$ simultaneously. Without the Pairs atom, the above operations cannot be completed in the ingress pipeline, and the packet needs to be re-submitted to the parser as a new packet to continue this action. The pseudo code for Stage 6 is listed below.

Stage	Line	Pseudo Code	Atom
6	17	$l \leftarrow Hash_2(mHvH.id)$	Pairs
	18	**if** $T_2(l).id = mHvH.id$ **then** {	
	19	$T_2(l).c \leftarrow T_2(l).c + mHvH.c$	
	20	} **else** {	
	21	**if** $T_2(l).c < mHvH.c$ **then** {	
	22	$T_2(l).id \leftarrow mHvH.id$	
	23	$T_2(l).c \leftarrow mHvH.c$ }}	

The Pairs atom hashes $mHvH.id$ to a location l in T_2 at Line 17 (Fig. 8(23)). Line 18 checks if $T_2(l).id$ is equal to $mHvH.id$ (Fig. 8(24)). If so, condition C1 is satisfied (a hit occurs) and $T_2(l).c$ is incremented by $mHvH.c$ at Line 19 (Fig. 8(25) and (32)). If $T_2(l).id \neq mHvH.id$, then either C2 or C3 is met (there is a miss). Note that if the location is not occupied, it also meets C2. We compare $T_2(l).c$ and $mHvH.c$ at Line 21 (Fig. 8(26)). If $T_2(l).c < mHvH.c$, condition C2 is met, and $mHvH.id$ is stored into $T_2(l).id$ as the new flow id at Line 22 (Fig. 8(27) and (31)). At the same time, $T_2(l).c$ is set to $mHvH.c$ at Line 23 (Fig. 8(28) and (32)). On the other hand, if C3 is met, then $T_2(l).id$ remains unchanged (Fig. 8(29) and (31)). Also, $T_2(l).c$ remains unchanged (Fig. 8(30) and (32)).

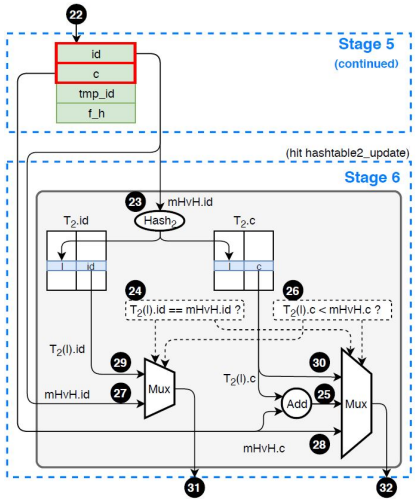


Fig. 8. HP-2 (Stage 6).

After Sage 6, the operations on the ingress pipeline are completed, and the packet is queued at the traffic manager. Then it is sent to the egress pipeline and then the deparser assembles the headers back to a well-formed packet. Finally, the packet is sent out of the P4 switch.

Note that in HP-2, when we update T_1, we do not need to use the Pairs atom because the HashPipe algorithm always inserts the new flow id in the first hash table. That is, we can update $T_1(l).id$ without the knowledge whether $T_1(l).c$ is smaller than the count for the new flow id or not. Therefore, we can use a PRAW atom and an IfElseRAW atom to update $T_1(l).id$ and $T_1(l).c$ respectively. On the other hand, conditions C1-C3 need to be identified by processing both the $T_2(l).id$ and the $T_2(l).c$ inequalities at the same time by the 4-way predication, which requires the usage of the Pairs atom.

IV. Enhanced HashPipe Algorithm

In the previous section, we show that HP-2 can be nicely implemented in the P4 pipeline architecture. For $N > 2$, HP-N (the HashPipe algorithm with N tables) needs to use $N-1$ Pairs atoms to manipulate T_n for $2 \leq n \leq N$. Unfortunately, due to the P4 hardware limitation for stateful memory access, we cannot use more than one Pairs atom in the pipeline for the HashPipe algorithm with the $mHvH$ metadata and the N hash tables. Without relaxing the stateful memory access limitation, the packet needs to be resubmitted to the switch to complete the execution for HP-N when $N > 2$. Resubmission of packets incurs significant overheads in time and space complexities, and may have synchronization problems on the hash tables. We propose two approaches to resolve the stateful memory access issue for P4 switch. Section IV-A proposes a modification to HashPipe called Enhanced HashPipe (EHP)

that can manipulate one more table on HP-N with the same number of the Pairs atoms used in HP-N. In other words, EHP-N can manipulate on the same number of hash tables as HP-$(N+1)$ with less time and space complexities (one less Pairs atom). Section IV-B uses extra stateful memory data structure that allows HP-N to execute $N-1$ Pairs atoms.

A. Enhanced HP

In Enhanced HashPipe (EHP), T_1 is partitioned into two equal-sized tables. Consider the N-table HashPipe Algorithm HP-N. Its enhancement EHP-N has $N+1$ hash tables where the first table is partitioned into $T_{1,A}$ and $T_{1,B}$. When a packet arrives, it is hashed to $T_{1,A}(l)$. If the slot is empty, then the packet is stored in the slot (just like what we did at T_1 in HP-N). If a miss occurs, then unlike HP-N (that replaces the flow id/count at $T_1(l)$), EHP-N checks if the packet hits $T_{1,B}(l)$, the counterpart slot of $T_{1,A}(l)$ at $T_{1,B}$. Then the action takes on $T_{1,B}(l)$ is the same as what we did at $T_1(l)$ in HP-N. For a flow id/count pair stored in the first tables, EHP gives it a second chance not to be replaced. For $N=2$, the main advantage of EHP-2 is that it can be implemented in the P4 switch without packet resubmission.

B. HashPipe with N Hash Tables ($N>2$)

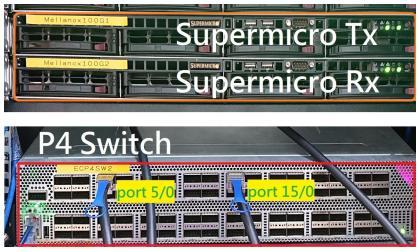
For $N>2$, to implement HP-N, we need to use $N-1$ Pairs atoms to manipulate T_n for $2 \le n \le N$. However, due to the P4 hardware limitation in stateful memory access, we cannot use more than one Pairs atom in the pipeline for $N>2$. To solve this problem, we introduce a new data structure. We first note that the hash table T_n consists of two parts: $T_n.id$ (for storing the flow ids) and $T_n.c$ (for storing the counts). To implement HP-N, we partition T_n into three portions: $T_n.id$, $T_n.c$ and $T_n^*.c$ (for $2 \le n < N$) where $T_n^*.c$ is a duplicate of $T_n.c$. Therefore, the sizes of the three portions are the same. After updating the hash table T_n, the portion $T_n^*.c$ is updated to make sure that it has the same contents as $T_n.c$. When the carried flow id and its count arrive at T_n, it is hashed to $T_n(l)$. If a miss occurs and $T_n(l).c$ is smaller than the carried count, the flow id/count at $T_n(l)$ are replaced by the carried flow id and its count. Like what we did for HP-2, HP-N uses a Pairs atom to update $T_n(l).id$ and $T_n(l).c$, and store $T_n(l).id$ into $mHvH.tmp\_id$. According to $mHvH.tmp\_id$, we can confirm if there is a hit, a miss, or an empty slot and take the corresponding operations to synchronize $T_n^*(l).c = T_n(l).c$. At the same time, $T_n^*(l).c$ is stored into $mHvH.tmp\_c$. Therefore, the replaced flow id and its count at T_n are stored in $mHvH.tmp\_id$ and $mHvH.tmp\_c$, respectively. Note that we need to use $T_n^*(l).c$ to update $mHvH.tmp\_c$ because we cannot directly assign $T_n(l).c$ to $mHvH.tmp\_c$ due to the stateful memory access limitation in the P4 pipeline. With the data structure $T_n^*(l).c$, we can carry the replaced id and count pair along with the packet, and process them at the next stage with T_{n+1}.

In our HP-N implementation, $(N-2)/(3N-2)$ of the memory space are used by $T_n^*.c$ to duplicate $T_n.c$ for $2 \le n < N$.

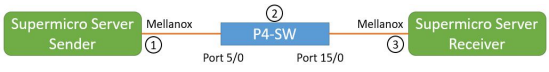
V. Performance Evaluation

We have implemented HashPipe with HP-2, EHP-2, HP-3 and EHP-3 in the P4 switch. We also implemented HashPipe with different numbers of hash tables in a CPU architecture to validate that the P4 switch implementations are correct. We have conducted measurement experiments where the input of HP-N is a trace obtained from a 10Gb/s ISP backbone link provided from CAIDA website [14]. This trace is a sequence of 23,676,763 packets belonging to $K=385625$ different flow ids. The output of HP-N is 200 flow ids, hopefully to be the ids for the top 200 heavy flows. The output measure for HP-N is the error rate $\mathbf{e_N}$ defined as the percentage of top heavy flows that are not captured by HP-N. For example, among the 200 flow ids reported by HP-N, if 10 of them are not among top flow ids, then $\mathbf{e_N} = 10/200 = 5\%$. The error rates for HP-N implemented in the P4 switch are exactly the same as that for those in the CPU architecture (with the same hash functions), which indicates that our HP-N implementation in the P4 switch is correct.

Our experiments use two Supermicro servers (Intel(R) Xeon(R) CPU E5-2675 v3 @ 1.80GHz (16 Cores 32 Threads)) with the Mellanox Connect-X5 100G NICs and a EdgeCore P4 switch (see Fig. 9a). A Supermicro server generates 100Gbps traffic by using Mellanox Connect-X5 with DPDK and Pktgen. The P4 switch handles the packets without queueing, and its maximum processing capability per output port is given by its "line rate". When the packets arrive within the line rate, no packet is dropped at the switch. One Supermicro server (Fig. 9b(1)) generates the packets sent to the EdgeCore P4 switch (Fig. 9b(2)). Another Supermicro server (Fig. 9b(3)) receives the packets from the P4 switch. The line rate for the EdgeCore P4 switch based on Barefoot Tofino chip is 100Gbps per port (and there are 64 ports in the switch, which results in 6.4 tera bits per second).



(a) Physical connection of the EdgeCore switch and the Supermicro servers



(b) Functional block diagram

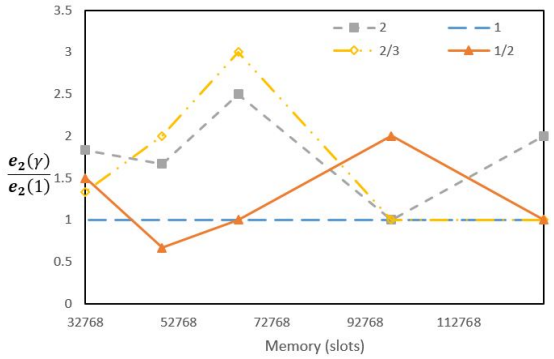
Fig. 9. The experiment setup with 2 Supermicro servers and one EdgeCore P4 switch.

Now we investigate the impact of total memory budget, the sizes of T_n and the probability that a packet belongs to a specific flow. Let L_n be the size (the unit is "slot") of the table T_n, where a slot can store a flow identity or a count.

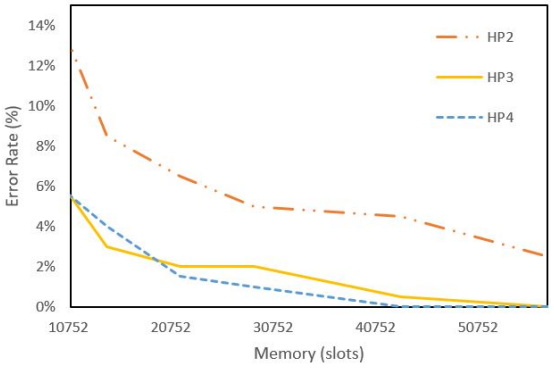
The total memory budget for HP-N is

$$L = \sum_{n=1}^{N} L_n$$

Consider HP-2. Let $\gamma = L_1/L_2$, and $\mathbf{e_2}(\gamma)$ be the error rate of HP-2 with the ratio γ. Fig. 10a plots $\mathbf{e_2}(\gamma)/\mathbf{e_2}(1)$, the error rate normalized by the case when $\gamma = 1$. The figure shows that for all L and γ values, $\mathbf{e_2}(\gamma) \leq \mathbf{e_2}(1)$ except for the case when $L = 52768$ slots, where $\mathbf{e_2}(1/2) < \mathbf{e_2}(1)$. Similar phenomenon is observed for other N values. Therefore, the best accuracy performance is observed when the hash tables have the same size; i.e., $L_i = L_j$ for $1 \leq i, j \leq N$.



(a)



(b)

Fig. 10. Error rate performance: (a) $\mathbf{e_2}(\gamma)/\mathbf{e_2}(1)$ for various γ values ($N = 2$); (b) Error rates for HP-2, HP-3 and HP-4.

Our observation can be explained by the following simple derivation. Consider a sequence of packets arriving at the P4 switch. A packet has the flow id k with the probability q_k where $1 \leq k \leq K$. Therefore, we have

$$\sum_{k=1}^{K} q_k = 1$$

Suppose that we observe a sequence of M packets arriving at the switch. Among them, the number of packets with id k is M_k, then

$$q_k = \lim_{M \to \infty} \frac{M_k}{M}$$

We further assume that $K \gg L$ (if $K < L$, then in perfect hash, there will be no collisions and all flow ids will be counted in the hash tables without any error). In perfect hash, a flow id will be mapped to a slot with probability $1/L_n$. Suppose that in the steady state (i.e., when all slots in T_n are occupied) the flow id k of an arrived packet hits a perfect hash table T_n with the probability $x_{k,n}$. Therefore the probability of a collision $p_{n,k}$ for flow id k at T_n is

$$p_{n,k} = \left(\frac{1}{L_n}\right)(1 - x_{k,n})$$

and the probability p_n of a collision for an arbitrary flow id at T_n is

$$p_n = \sum_{k=1}^{K} q_k p_{n,k} = \left(\frac{1}{L_n}\right)\left[\sum_{k=1}^{K} q_k (1 - x_{k,n})\right]$$

In HP-2, a flow id in one of the hash tables is removed from T_2 if a collision occurs at T_1 and another collision occurs at T_2. The probability p for such situation is

$$\begin{aligned} p &= p_1 p_2 \\ &= \left(\frac{1}{L_1}\right)\left[\sum_{k=1}^{K} q_k (1 - x_{k,1})\right]\left(\frac{1}{L_2}\right)\left[\sum_{k=1}^{K} q_k (1 - x_{k,2})\right] \\ &= \left(\frac{1}{L_1 L_2}\right)\left[\sum_{k=1}^{K} q_k (1 - x_{k,1})\right]\left[\sum_{k=1}^{K} q_k (1 - x_{k,2})\right] \end{aligned} \quad (1)$$

Now, let us consider the effect of L_1. For fixed q_k and L, the table size L_1 results in minimum p of equation (1) when the differentiation $dp/dL_1 = 0$ or

$$\left(\frac{d}{dL_1}\right)\left[\frac{1}{L_1 (L - L_1)}\right] = \frac{2L_1 - L}{{L_1}^2 (L - L_1)^2} = 0$$

Finally, we have $L_1 = L/2$. This result is consistent with our observation in Fig. 10a, where the accuracy of HashPipe has the best performance when the hash tables are of the same size. Now we generalize the above result for HP-N by showing that for fixed N, L and q_k, the best collision performance for HP-N occurs when the sizes of the hash tables are the same; i.e., $L_1 = L_2 = \cdots = L_n = L/N$. We first note that a flow id in one of the hash tables is removed from T_N if a collision occurs at every hash table T_n. The probability p for such situation is

$$\begin{aligned} p = \prod_{n=1}^{N} p_n &= \prod_{n=1}^{N} \left\{\left(\frac{1}{L_n}\right)\left[\sum_{k=1}^{K} q_k (1 - x_{k,n})\right]\right\} \\ &= \left(\prod_{n=1}^{N} \frac{1}{L_n}\right)\left\{\prod_{n=1}^{N}\left[\sum_{k=1}^{K} q_k (1 - x_{k,n})\right]\right\} \end{aligned} \quad (2)$$

From the inequality of arithmetic and geometric means, we have

$$\prod_{n=1}^{N} L_n \leq \left(\frac{\sum_{n=1}^{N} L_n}{N}\right)^N = \left(\frac{L}{N}\right)^N \quad (3)$$

and substitute (3) into (2) to yield

$$p \geq \left(\frac{N}{L}\right)^N \prod_{n=1}^{N}\left[\sum_{k=1}^{K} q_k (1 - x_{k,n})\right]$$

The minimum p value of above inequaliby occurs when $L_1 = L_2 = \cdots = L_n = L/N$.

Now we show how q_k may affect p of equation (2). In the steady state, q_k is the probability that a packet finds that

the previously arrived packet also has the same id. Therefore, q_k is positively related to the probability $x_{k,n}$ that the id is found in a hash table. That is, if there are more packet arrivals of id k, then there is a larger count of id k in a hash table (higher probability of a hit at the table). Here we make a rough assumption to use q_k as the probability that the packet hits a hash table. That is, $x_{k,n} = q_k$. Then we have

$$p = \prod_{n=1}^{N} p_n = \prod_{n=1}^{N} \left[\left(\frac{1}{L_n} \right) \left(1 - \sum_{k=1}^{K} {q_k}^2 \right) \right] = \left(\prod_{n=1}^{N} \frac{1}{L_n} \right) \left(1 - \sum_{k=1}^{K} {q_k}^2 \right)^N \quad (4)$$

From the inequality of arithmetic and geometric means, we have

$$\sqrt[K]{\prod_{k=1}^{K} {q_k}^2} \leq \frac{\sum_{k=1}^{K} {q_k}^2}{K} \quad (5)$$

and the minimum of the right-hand side occurs when $q_1 = q_2 = \cdots = q_K = 1/K$. Therefore, for a fixed L_n value, substitute the minimum of (5) into (4) to yield

$$p \leq \left(\prod_{n=1}^{N} \frac{1}{L_n} \right) \left(1 - \frac{1}{K} \right)^N$$

and the maximum p of above inequality occurs when $q_1 = q_2 = \cdots = q_K = 1/K$. This result is consistent with our intuition. When $q_i \approx q_j$ for $1 \leq i, j \leq K$, the counts for all flow ids are almost identical and it is more difficult to identify the heaver flows.

Following the same assumption $x_{k,n} = q_k$, we consider the effect of N as follows. Consider a fixed L with equal-size hash tables, i.e., $L_1 = L_2 = \cdots = L_n = L/N$. Let $\alpha = (1 - \sum_{k=1}^{K} {q_k}^2)/L$. Clearly, $\alpha < 1/N$ since $L > N$. We express the probability p of equation (4) as a function of N

$$p\,(N) = (\alpha N)^N$$

and

$$p\,(N+1) = \alpha N \left(1 + \frac{1}{N} \right)^{N+1} p\,(N)$$

Under the same memory budget, it makes sense to increase the number of hash tables if

$$\alpha N \left(1 + \frac{1}{N} \right)^{N+1} < 1$$

If $\alpha < 1/(4N)$, it is clear that the above inequality always holds, and $p(N+1) < p(N)$. On the other hand, if α is only slightly smaller than $1/N$, then

$$\alpha N \left(1 + \frac{1}{N} \right)^{N+1} \approx \left(1 + \frac{1}{N} \right)^{N+1} > 1$$

and $p(N+1) > p(N)$. Since the time complexity of HP-N increases as N increases, it is not practical to select a large N. Fig. 10b illustrates the error rates for HP-2, HP-3 and HP-4 in our experiments. The figure indicates that it is appropriate to select $N = 3$. As we described in the previous section, we can implement HP-3 and EHP-3 in the P4 switch. However, due to the P4 pipeline architecture limitations and the time complexity, it is not practical to implement HP-N in the real P4 switch. Fortunately, for the same memory budget in the CPU architecture or the bmv2 model, Fig. 10b indicates that for $N = 4$, the improvement of the error rate is insignificant. In a P4 switch, increasing N has less benefit because some memory slots must be allocated for $T_2^*.c$. Therefore, it suffices to use HP-3 (EHP-3) in a P4 switch.

Fig. 11 shows the error rates for HP-2 vs EHP-2 and HP-3 vs EHP-3 with various memory budgets (L). Note that in both HP-3 and EHP-3, $((N-2)/(3N-2))L = L/7$ slots are used for $T_2^*.c$. The figure indicates that in terms of error rates, EHP-2 outperforms HP-2 by 40% $\sim$ 80%. Similarly, EHP-3 outperforms HP-3 by 25% $\sim$ 50%.

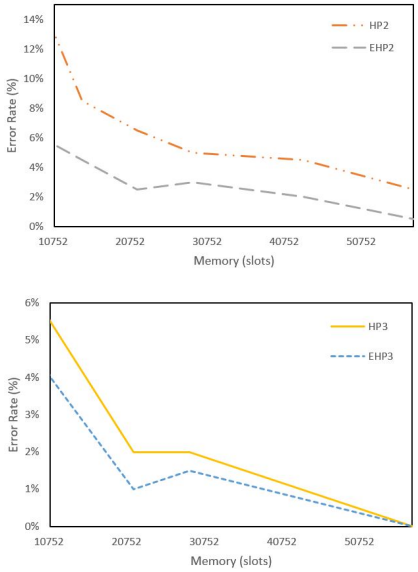


Fig. 11. HP-2 vs EHP-2 and HP-3 vs EHP-3.

Now we investigate the improvements of EHP-3 over HP-2, EHP-2 and HP-3, respectively. Let $\mathbf{e_2}, \mathbf{e_{E2}}, \mathbf{e_3}, \mathbf{e_{E3}}$ be the error rates of HP-2, EHP-2, HP-3 and EHP-3, where the sizes of the hash tables are the same. For $N = 2$ and 3, define the improvement of EHP-3 over (E)HP-N as

$$\mathbf{I_{E3}}\,(N) = \frac{\mathbf{e_N} - \mathbf{e_{E3}}}{\mathbf{e_{E3}}} \text{ for } N = 2,\ E2, \text{ and } 3$$

Fig. 12 illustrates $\mathbf{I_{E3}}(2)$, $\mathbf{I_{E3}}(E2)$ and $\mathbf{I_{E3}}(3)$ with various numbers of memory slots (L). The figure shows that $\mathbf{I_{E3}}(2)$ can be over 500%. In many cases, 80% improvement can be expected.

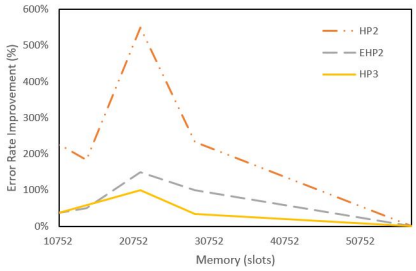


Fig. 12. Improvements of EHP-3 over other HashPipe algorithms.

VI. Conclusion

This paper developed HashPipe as a soft computing application in a real P4 switch. The HashPipe algorithm for heavy hitter detection was proposed for P4-based SDN to identify outliers in network traffic. The original HashPipe algorithm was implemented in bmv2 simulator, which cannot be directly executed in a real P4 switch. This paper showed that the implementation of HashPipe in a P4 switch is not trivial. We described how to smartly utilize the atoms of the Banzai machine to implement HashPipe in the P4 switch. The HashPipe algorithm manipulates multiple hash tables to store the heavy flows. Heavy hitter detection is more accurate if more hash tables are used at the cost of more computation time. Under the same memory budget, we showed that the best accuracy performance occurs when all hash tables are of the same size. We proposed the enhanced HashPipe algorithm that allows manipulation of one more hash table than the original HashPipe algorithm, which improve the accuracy of the detection by up to 80%. Our algorithm is executed at the line-rate of the P4 switch, which is at 6.4 Tera bits per second (with 64 ports). To our knowledge, this is fastest heavy hitter detection operation in the world.

References

[1] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 8.

[2] O. Rottenstreich and J. Tapolcai, "Optimal rule caching and lossy compression for longest prefix matching," *IEEE/ACM Transactions on Networking (TON)*, vol. 25, no. 2, pp. 864–878, 2017.

[3] Cisco ios netflow. [Online]. Available: http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html

[4] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers." in *Nsdi*, 2016, pp. 311–324.

[5] 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, 2005.

[6] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 101–114.

[7] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International Conference on Database Theory*. Springer, 2005, pp. 398–412.

[8] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.

[9] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss, "Space-optimal heavy hitters with strong error bounds," *ACM Transactions on Database Systems (TODS)*, vol. 35, no. 4, p. 26, 2010.

[10] Y.-B. Lin, S.-Y. Wang, C.-C. Huang, and C.-M. Wu, "The sdn approach for the aggregation/disaggregation of sensor data," *Sensors*, vol. 18, no. 7, p. 2025, 2018.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[12] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 15–28.

[13] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Symposium on SDN Research*. ACM, 2017, pp. 164–176.

[14] The caida ucsd anonymized internet traces 2016. [Online]. Available: https://data.caida.org/datasets/passive-2016

Yi-Bing Lin (M'96-SM'96-F'03) received Ph.D. from University of Washington, Seattle, in 1990. From 1990 to 1995 he was a Research Scientist with Bellcore. He then joined National Chiao Tung University (NCTU) in Taiwan, where he remains. In 2011, Lin became the Vice President of NCTU. During 2014 - 2016, Lin was Deputy Minister, Ministry of Science and Technology, Taiwan. Lin is a member of board of directors, Chunghwa Telecom. He serves on the editorial boards of IEEE Trans. on Vehicular Technology. Lin is the co-author of the books Wireless and Mobile Network Architecture (Wiley, 2001), Wireless and Mobile All-IP Networks (John Wiley, 2005), and Charging for Mobile All-IP Telecommunications (Wiley, 2008). Lin received Executive Yuen, 2011 National Chair Award, and TWAS Prize in Engineering Sciences, 2011 (the Academy of Sciences for the World). Lin is AAAS Fellow, ACM Fellow, and IET Fellow.

Ching-Chun Huang was born in New Taipei City, Taiwan, in 1989. She received the B.S. degree in computer science from National Chung Cheng University, Chiayi, Taiwan, in 2011, and is currently working toward the Master's degree at the Institute of Network Engineering, National Chiao Tung University. Her research interests include software-defined networks, heavy hitter detection, and P4.

Shi-Chun Tsai (M'-06, SM'-16) received the B.S. and M.S. in Computer Science and Information Engineering from National Taiwan University, Taipei, Taiwan in 1984 and 1988, respectively. He received the Ph.D. degree in Computer Science from the University of Chicago in 1996. From 1996 to 2001, he worked at National Chi-Nan University. Then he joined the Department of Computer Science of National Chiao Tung University and promoted to Professor in 2007. He has served as the Director of Information Technology Service Center of National Chiao Tung University, since 2010. His research interests include Computational Complexity, Algorithms Design and Analysis, Coding theory, Software Defined Networking and Machine learning.