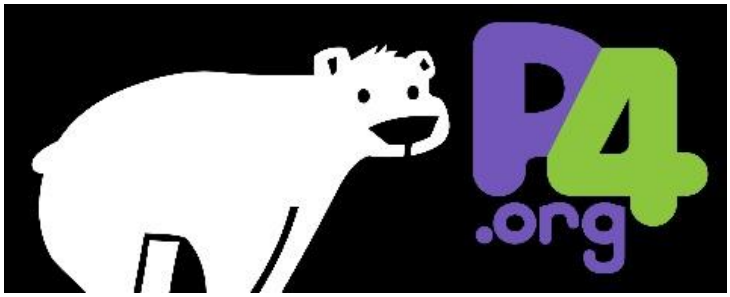
Technion – Israel Institute of Technology
Project in Computer Networks
Spring 2019



Heavy-Hitter Detection on SmartNIC - using P4

Team:
Yevhenii Liubchyk, Maria Shestakova

Supervised by:
Itzik Ashkenazi, Prof. Ori Rotenstreich

Contents

Introduction & Motivation

Many network management applications can benefit from finding the set of flows contributing significant amounts of traffic to a link. Such flows are called “Heavy Hitters”.

Monitoring heavy hitters is required for example, to relieve link congestion or to detect network anomalies and attacks. Furthermore, heavy-hitters identification is done in small time scales, it can enable dynamic routing of the heavy flows and also dynamic flow scheduling.

In order to respond quickly to short-term traffic variations, it is desirable to run heavy-hitter monitoring at all switches and (Network Interface Card) NICs in the network all the time. It should identify packets belonging to heavy-hitter flows in a reasonable accuracy and by meeting hardware constraints such as a limited number of accesses to the memory storing state and a limited amount of memory available.

The target of HashPipe algorithm is to track the K heaviest flows with high accuracy using limited available memory. The HashPipe can be implemented in programmable hardware such as Netronome SmartNic using P4 language.

Objectives

Goals

- Implement HashPipe algorithm on a programmable hardware Netronome SmartNic using P4 language.
- Evaluate the accuracy and quality of the algorithm

Estimated Timetable

- **April** – research Netronome SmartNIC and P4 environment.
- **May** – Implement HashPipe algorithm using P4 language.
- **June** – Find Algorithm that read and calculate real heavy-hitter flows. Treatment and counting heavy-hitter flows from HashPipe algorithm.
- **August** – Demo & Presentation delivery. Analyzing accuracy and quality of HashPipe algorithm.

Expected results

Inject traffic using real captured ISP backbone link traffic (from CAIDA) and produce the following statistics:

Overall Flow count

- When reporting xxx (TBD) heavy hitters: % of False Negative (Real heavy-hitter that was not detected)
- When reporting xxx (TBD) heavy hitters: % of False Positive (“fake” heavy-hitter detection)

Previous works

Among the previous works in this field, we must mention first and foremost the work of Ori Rottenstreich, Srinivas Narayana, Vibhaalakshmi Sivaraman, Jennifer Rexford, S. Muthukrishnan [1]. They proposed an algorithm to detect heavy traffic flows within the constraints of emerging programmable switches, and making this information available within the switch itself, as packets are processed.

Their solution, HashPipe, uses a pipeline of hash tables to track heavy flows preferentially, by evicting lighter flows from switch memory over time. They prototype HashPipe with P4, walking through the switch programming features used to implement their algorithm.

Through simulations on a real traffic trace, they showed that HashPipe achieves high accuracy in finding heavy flows within the memory constraints of switches today.

It is this algorithm that we have implemented and evaluated in our project.

Background

Netronome SmartNIC

Netronome SmartNIC is high-performance intelligent networking solutions that enable rapid innovation at lower cost and power through domain-specific, open and efficient programming models and ability to offload network and security processing.

Netronome SmartNIC environment that includes:

- Agilio CX SmartNIC
- Interfaces: 2x10Gb
- Processor: NFP-4000
- Memory: 2GB

There are two physical ports in SmartNIC. But in our project we don't use them. We work only with two virtual ports.

The Netronome SmartNIC has a five-core processor.

P4 Language

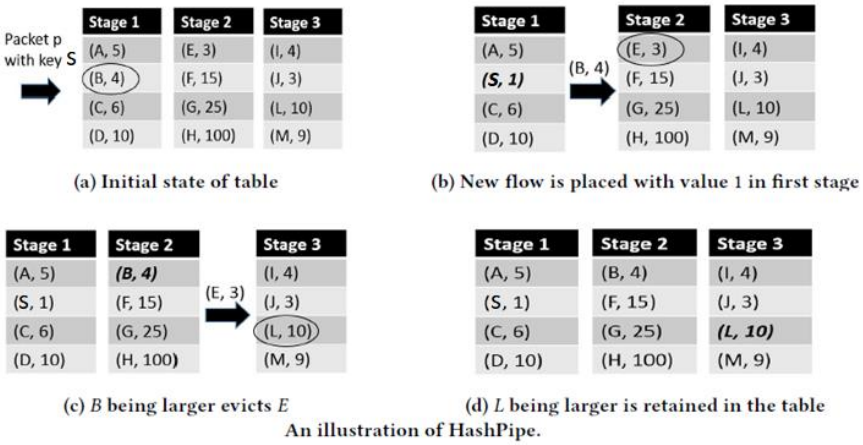
- P4 is a programming language designed to allow programming of packet forwarding planes.
- P4 is a domain-specific language with a number of constructs optimized around network data forwarding

Programmer Studio 6.0

The Programmer Studio in the Agilio P4C SDK 6.0 is the industry’s first P4 and C GUI-based Integrated Development Environment (IDE) for dynamically programming new and innovative networking capabilities on the AgilioCX

In work with Programmer Studio we use [4].

HashPipe Algorithm



An illustration of HashPipe.

Algorithm : HashPipe: Pipeline of d hash tables

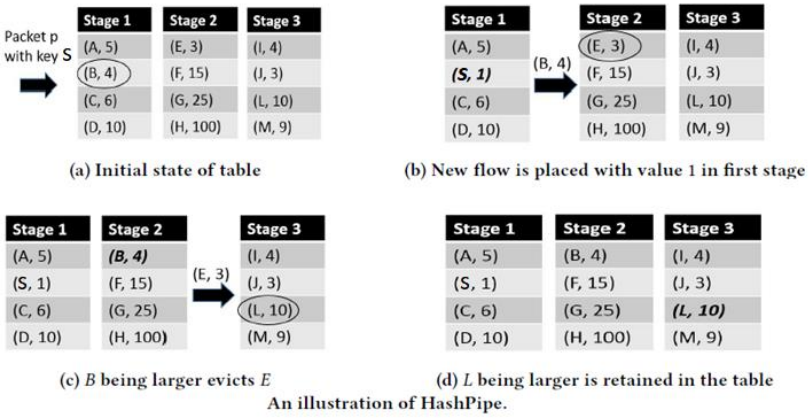
```
▷ Insert in the first stage
l_1 ← h_1(iKey)
if key_{l_1} = iKey then
    val_{l_1} ← val_{l_1} + 1
    end processing
end
else if l_1 is an empty slot then
    (key_{l_1}, val_{l_1}) ← (iKey, 1)
    end processing
end
else
    (cKey, cVal) ← (key_{l_1}, val_{l_1})
    (key_{l_1}, val_{l_1}) ← (iKey, 1)
end
▷ Track a rolling minimum
for i ← 2 to d do
    l ← h_i(cKey)
    if key_l = cKey then
        val_l ← val_l + cVal
        end processing
    end
    else if l is an empty slot then
        (key_l, val_l) ← (cKey, CVal)
        end processing
    end
    else if val_l < cVal then
        swap (cKey, cVal) with (key_l, val_l)
    end
end
```

HashPipe Algorithm tracks the **K** heaviest flows with high accuracy using limited available memory.

Example of processing a packet using HashPipe[1]:

A packet with a key S enters the switch pipeline (a), and since it isn't found in the first table, it is inserted there (b). Key B (that was in the slot currently occupied by S) is carried with the packet to the next stage, where it hashes to the slot containing key E. But since the count of B is larger than that of E, B is written to the table and E is carried out on the packet instead (c). Finally, since the count of L (that E hashes to) is larger than that of E, L stays in the table (d). The net effect is that the new key S is inserted in the table, and the minimum of the three keys B, E, and L—namely E—is evicted in its favor.

HashPipe Algorithm



(a) Initial state of table

(b) New flow is placed with value 1 in first stage

(c) B being larger evicts E

(d) L being larger is retained in the table

An illustration of HashPipe.

Algorithn : HashPipe: Pipeline of d hash tables

▷ Insert in the first stage

$l_1 \leftarrow h_1(iKey)$
if $key_{l_1} = iKey$ **then**
  $val_{l_1} \leftarrow val_{l_1} + 1$
  end processing
end
else if l_1 *is an empty slot* **then**
  $(key_{l_1}, val_{l_1}) \leftarrow (iKey, 1)$
  end processing
end
else
  $(cKey, cVal) \leftarrow (key_{l_1}, val_{l_1})$
  $(key_{l_1}, val_{l_1}) \leftarrow (iKey, 1)$
end

▷ Track a rolling minimum

16 **for** $i \leftarrow 2$ ***to*** d **do**
17 $l \leftarrow h_i(cKey)$
18 **if** $key_l = cKey$ **then**
19 $val_l \leftarrow val_l + cVal$
20 end processing
21 **end**
22 **else if** l *is an empty slot* **then**
23 $(key_l, val_l) \leftarrow (cKey, CVal)$
24 end processing
25 **end**
26 **else if** $val_l < cVal$ **then**
27 swap $(cKey, cVal)$ with (key_l, val_l)
28 **end**
29 **end**

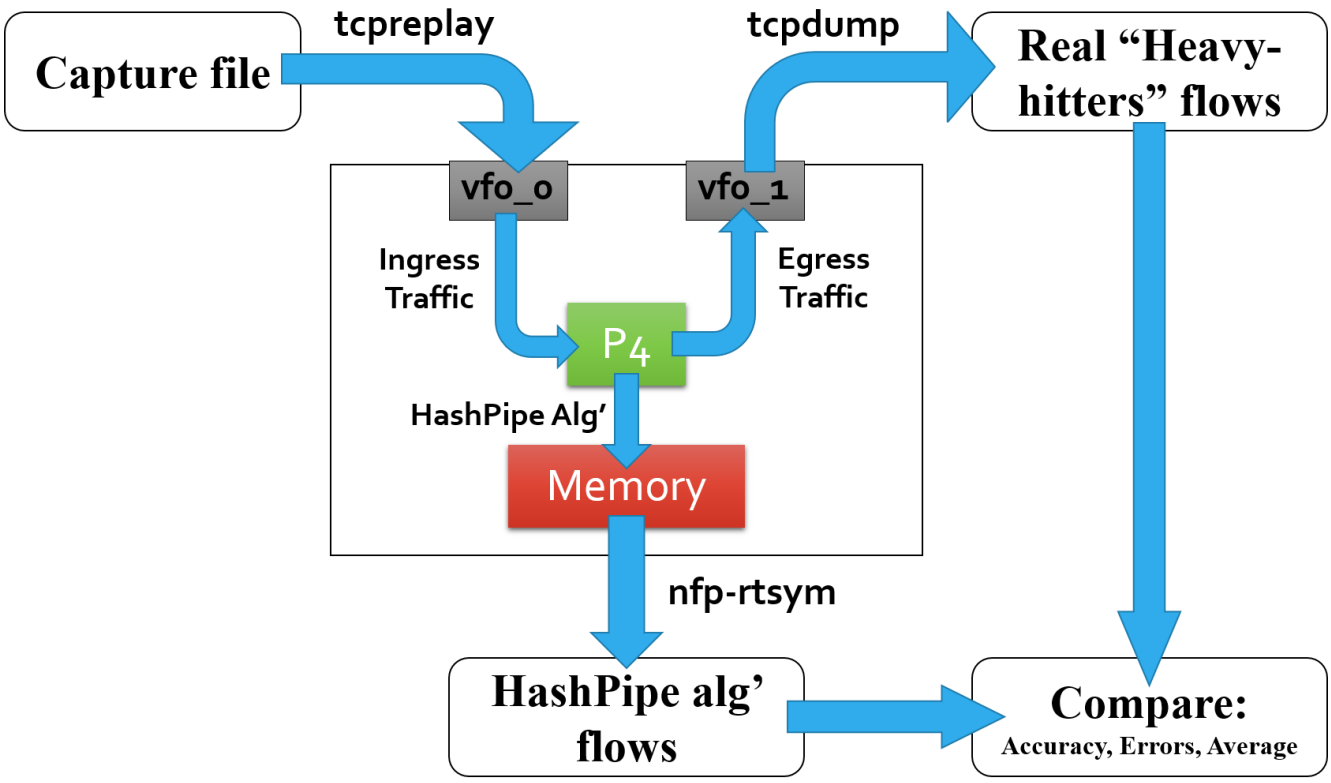
HashPipe Algorithm tracks the **K** heaviest flows with high accuracy using limited available memory.

Example of processing a packet using HashPipe[1]:

A packet with a key S enters the switch pipeline (a), and since it isn't found in the first table, it is inserted there (b). Key B (that was in the slot currently occupied by S) is carried with the packet to the next stage, where it hashes to the slot containing key E. But since the count of B is larger than that of E, B is written to the table and E is carried out on the packet instead (c). Finally, since the count of L (that E hashes to) is larger than that of E, L stays in the table (d). The net effect is that the new key S is inserted in the table, and the minimum of the three keys B, E, and L—namely E—is evicted in its favor.

Program outline and plan

Outline



Plan

1. Get the capture file as the input of our program.
2. Using the **tcpreplay** command, reproduce network traffic from the pcap file.
   Packets from pcap file arrive to the Netronome SmartNIC through virtual port vf0_0.
3. The Netronome SmartNIC accepts packages from virtual port vf0_0, checks that their protocol is TCP or UDP (With the P4 code) and drop packet if not.
4. All TCP and UDP packets processed using HashPipe Algorithm and are saved in the Memory of the Netronome SmartNIC and after that sent to a virtual port vf0_1.
5. Using the **nfp_rtsym** command, read from Netronome SmartNIC memory and save it in table "**HashPipe alg' flows"** for further processing.
6. Using the **tcpdump** command, save all data from virtual port vf0_1 to the table "**Real "Heavy-hitters" flows"**.
7. As a result, we have two tables: "**HashPipe alg' flows"** and "**Real "Heavy-hitters" flows"**.
   So, we evaluate the HashPipe Algorithm by comparing two tables and derive its accuracy and average place of packets from "**HashPipe alg' flows"** in sorted list of "**Real "Heavy-hitters" flows"** start from heaviest.

Commands

- **tcpdump** – is a type of packet analyzer software utility that monitors and logs TCP/IP traffic passing between a network and the computer on which it is executed.
- **tcpreplay** – is a tool for replaying network traffic from files saved with **tcpdump** or other tools which write pcap files
- **nfp_rtsym** – Read/write contents of runtime symbol

Packet processing code – hh.p4 (using P4 language)

The **hh.p4** file processes packets received by the Netronome SmartNIC through virtual port vf0_0.

1. Parse the network IPv4 packets.

```
header_type ipv4_t {
    fields {
        version : 4;
        ihl : 4;
        diffserv : 8;
        totalLen : 16;
        identification : 16;
        flags : 3;
        fragOffset : 13;
        ttl : 8;
        protocol : 8;
        hdrChecksum : 16;
        srcAddr : 32;
        dstAddr : 32;
    }
}
```

2. Check their protocol.

```
parser parse_ipv4 {
    extract(ipv4);
    return select(latest.protocol) {
        TCP_PROTO : parse_tcp;
        UDP_PROTO : parse_udp;
        //No default, so drop it if not tcp or udp
    }
}
```

   If the packet protocol is neither TCP nor UDP, then drop it.

```
action do_drop()
{
    drop();
}
```

3. For all valid packets (whose protocol is TCP and UDP):
   a. process packets using the HashPipe Algorithm which is implemented by the function in plugin.c file and save results to the Netronome SmartNIC memory;
   b. send packets to virtual port vf0_1.

```
primitive_action hashpipe_algorithm();
action do_forward(port) {
    hashpipe_algorithm();
    modify_field(standard_metadata.egress_spec, port);
}
```

Packet processing code was written using [2] and [3].

Packet processing code – plugin.c (using C language)

The **plugin.c** file implements HashPipe Algorithm:

1. Variables **NUM_SKETCH** and **SKETCH_COLUMN_COUNT** there are the number of hash-tables and hash-table size respectively.
```
#define SKETCH_COLUMN_COUNT 128
#define SKETCH_COLUMN_COUNT_MASK (SKETCH_COLUMN_COUNT-1)
#define NUM_SKETCH 12
```

Size of Netronome SmartNIC memory that we use represented by the formula:

Size = (number of cores) x (number of hash-tables) x (hash-table size)

Where number of cores for our Netronome SmartNIC equal **5.**

2. Struct **Heavy_Hitter** stores flow identifier (for our project it is **5-tuple**) and its counter.
```
struct Heavy_Hitter {
    uint32_t srcAddr;
    uint32_t dstAddr;
    uint16_t srcPort;
    uint16_t dstPort;
    uint8_t protocol;
    uint32_t count;
};
```

3. The variable **sketch** is used to save hash-tables obtained as a result of HashPipe Algorithm.
```
__export __mem static struct Heavy_Hitter sketch[NUM_SKETCH][SKETCH_COLUMN_COUNT];
```

4. Function **hash_func** describes the hash-functions used in HashPipe Algorithm. Function get hash-table number from 0 to **NUM_SKETCH** and return the number from 0 to **SKETCH_COLUMN_COUNT** that defines the place in this hash-table.
```
uint32_t hash_func(uint32_t function_num, struct Heavy_Hitter heavy_hitter)
```

5. Function **pif_plugin_hashpipe_algorithm** implements HashPipe Algorithm.
```
int pif_plugin_hashpipe_algorithm(EXTRACTED_HEADERS_T *headers, MATCH_DATA_T *match_data)
```

Packet processing code was written using [2] and [3].

Estimation code – getResult.py (using Python language)

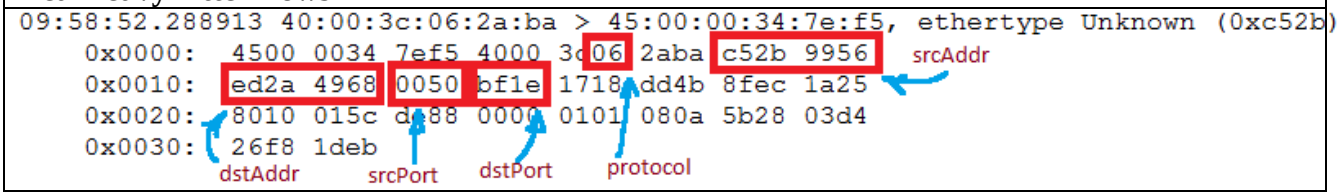
The **getResult.py** file processes and compares files **RealHeavyHitterFlows** and **HashPipeFlows**, which were obtained as a result of the execution of the packet processing code on Netronome SmartNIC, and calculates results.

After processing all packets using Netronome SmartNIC there are two files:

1. **HashPipeFlows** which is obtained by reading the variable **sketch** from Netronome SmartNIC memory using command **nfp-rtsym**;
2. **RealHeavyHitterFlows** which is the result of a command **tcpdump.**

HashPipeFlows

RealHeavyHitterFlows



For each file we create a specialized container with a data structure where **Key** is **5-tuple** of packet and Value is counter of packet (number of packet with similar **Key** in corresponding file):

1. Container for **HashPipeFlows** is
   ```
   HashPipeFlows = collections.Counter()
   ```
2. Container for **RealHeavyHitterFlows** is
   ```
   RealHeavyHitterFlows = collections.Counter()
   ```

File **getResult.py** gets a variable **K** as an input, which defines a number of "heavy-hitter" flows which we want to check. If the value of **K** is greater than the number of packets in one of the structures **HashPipeFlows** or **RealHeavyHitterFlows** then we reduce the value of **K** to the size of the smaller of the structures, since we cannot check the larger number of packets:

```
numOfHH = int(sys.argv[1])# K is number of heavy-hitter flows which we want to check
if (numOfHH > len(HashPipeFlows)):
    numOfHH = len(HashPipeFlows)
if (numOfHH > len(RealHeavyHitterFlows)):
    numOfHH = len(RealHeavyHitterFlows)
```

In the graphs, this value is denoted as **Optimal K.**

Then we calculate the results:

1. **Accuracy** of the HashPipe algorithm:

```
######################## Accuracy of HashPipe Algorithm ########################
countOfHH = 0
for hh in HashPipeFlows.most_common(numOfHH):
    iterator = 0
    for item in RealHeavyHitterFlows.most_common(numOfHH):
        if hh[0] in item:
            countOfHH += 1
            break;
        iterator += 1

print("Accuracy: " + str(float(countOfHH*100)/numOfHH) + "%")
```

2. **Average** place of packets from HashPipe Algorithm flows in sorted list of Real Heavy-hitter flows start from the heaviest:

```
######### Average place of packets HashPipe Algorithm in sorted list #########
############ of Real "Heavy-Hitter" flows start from the heaviest ############
averageCount = 0
for hh in HashPipeFlows.most_common(numOfHH):
    iterator = 0
    for item in RealHeavyHitterFlows.most_common(len(RealHeavyHitterFlows)):
        if hh[0] in item:
            averageCount += iterator
            break;
        iterator += 1

print("Average: " + str(float(averageCount)/numOfHH))
print("Optimal average: " + str(float(numOfHH + 1)/2))
```

Below is an example of calculation:

- **Accuracy**: in the first **K=3** packets in the table "Real Heavy-Hitter Flows" there are **2** packages from the first **K=3** packets of the table "HashPipe Alg' Flows"
- **Average**: first **K=3** packets of the table "HashPipe Alg' Flows" are in places **1, 3, 5** in the table "Real Heavy-Hitter Flows"

Example

Input: K = 3

Output:

- **Accuracy** = $\frac{2}{3} \cdot 100\% = 66.7\%$
- **Average** = $\frac{1+3+5}{3} = 3$
- **Optimal Average** = $\frac{1+2+3}{3} = 2$

HashPipe Alg' FLows	Real Heavy-Hitter Flows
1	1
3	2
5	3
2	4
4	5

Console Code – Makefile

```
.PHONY: tcpdump tcpreplay result

tcpdump:
    sudo tcpdump -B 262144 -l -xx -n -v -i vf0_1 > RealHeavyHitterFlows

tcpreplay:
    sudo tcpreplay -L$(NUM) -i vf0_0 $(CAPTURE)
    sudo /opt/netronome/bin/nfp-rtsym i32._sketch.108 > HashPipeFlows
    sudo /opt/netronome/bin/nfp-rtsym i33._sketch.108 >> HashPipeFlows
    sudo /opt/netronome/bin/nfp-rtsym i34._sketch.108 >> HashPipeFlows
    sudo /opt/netronome/bin/nfp-rtsym i35._sketch.108 >> HashPipeFlows
    sudo /opt/netronome/bin/nfp-rtsym i36._sketch.108 >> HashPipeFlows

result:
    python getResult.py $(K)
```

The **Makefile** is designed to run the program through the console.

The **Makefile** includes 3 commands:

1. **"make tcpdump"** – runs command **tcpdump** to read information from the virtual port vf0_1 and writes the result to file **RealHeavyHitterFlows.**

2. **"make tcpreplay NUM=... CAPTURE=..."** – runs command **tcpreplay** to replay traffic from Capture file to Netronome SmartNIC through the virtual port vf0_0 and after that runs command **nfp-rtsym** to save information from the variable **sketch** of all five cores of the Netronome SmartNIC memory to file **HashPipeFlows.**
   Arguments:
   a. The variable **NUM** is the number of packets that we send.
   b. The variable **CAPTURE** is the name of pcap file which we send.

3. **"make result K=..."** – run python file **getResult.py** for processing and comparing files **RealHeavyHitterFlows** and **HashPipeFlows** and calculating results.
   Argument:
   a. The variable **K** is number of heavy-hitter flows which we want to check.

Launch program through the console

Before starting the program:

1. Create a project in Programmer Studio 6.0 with files **hh.p4**, **plugin.c** and **user_config.p4cfg**.
2. Run Build.
3. Put files **getResult.py** and **Makefile** on the server.

For each test:

1. Run Start Debugging in the project which was created in Programmer Studio 6.0.
2. Open 2 consoles.
3. In the first console, run the command **"make tcpdump":**

```
iashken@lccntarshish:~$ make tcpdump
sudo tcpdump -B 262144 -l -xx -n -v -i vf0_1 > RealHeavyHitterFlows
tcpdump: listening on vf0_1, link-type EN10MB (Ethernet), capture size 262144 by
tes
```

4. In the second console, run the command **"make tcpreplay NUM=... CAPTURE=...":**

```
iashken@lccntarshish:~$ make tcpreplay NUM=100000 CAPTURE=test1.pcap
sudo tcpreplay -L100000 -i vf0_0 test1.pcap
sending out vf0_0
processing file: test1.pcap
Warning in tcpreplay.c:replay_file() line 279:
test1.pcap DLT (RAW) does not match that of the outbound interface: vf0_0 (EN10MB)
Actual: 100000 packets (4462280 bytes) sent in 0.36 seconds.           Rated: 12395222.0 bps, 94.57 Mbps, 277777.78 pp
s
Statistics for network device: vf0_0
        Attempted packets:         100000
        Successful packets:        100000
        Failed packets:            0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0
sudo /opt/netronome/bin/nfp-rtsym i32._sketch.108 > HashPipeFlows
sudo /opt/netronome/bin/nfp-rtsym i33._sketch.108 >> HashPipeFlows
sudo /opt/netronome/bin/nfp-rtsym i34._sketch.108 >> HashPipeFlows
sudo /opt/netronome/bin/nfp-rtsym i35._sketch.108 >> HashPipeFlows
sudo /opt/netronome/bin/nfp-rtsym i36._sketch.108 >> HashPipeFlows
iashken@lccntarshish:~$
```

5. Waiting for the end of writing to the file **RealHeavyHitterFlows**. After that in the second console, run the command **"make result K=...":**

```
iashken@lccntarshish:~$ make result K=2000
python getResult.py 2000
Accuracy: 80.05%
Average: 1259.6095
Optimal average: 999.5
iashken@lccntarshish:~$
```

6. Run End Debugging

Graph Description

Output↓ Input→	K	12xC	5xBxC	NUM
Accuracy	Graph 1.1	Graph 2.1	Graph 3.1	Graph 4.1
Average	Graph 1.2	Graph 2.2	Graph 3.2	Graph 4.2

To evaluate the HashPipe Algorithm, we ran tests with different parameters and built graphs of the dependence of **Accuracy** and **Average** on several arguments:

1. **K –** number of heavy-hitter flows which we want to check.
2. **12xC – memory size** through resizing hash-tables with a constant number of hash-tables
3. **BxC** – the ratio of the hash-table size to number of hash-tables with a constant **memory size**, i.e. a constant product of number of hash-tables by hash-table size
4. **NUM –** number of packets that we send.

For each argument we build to graphs:

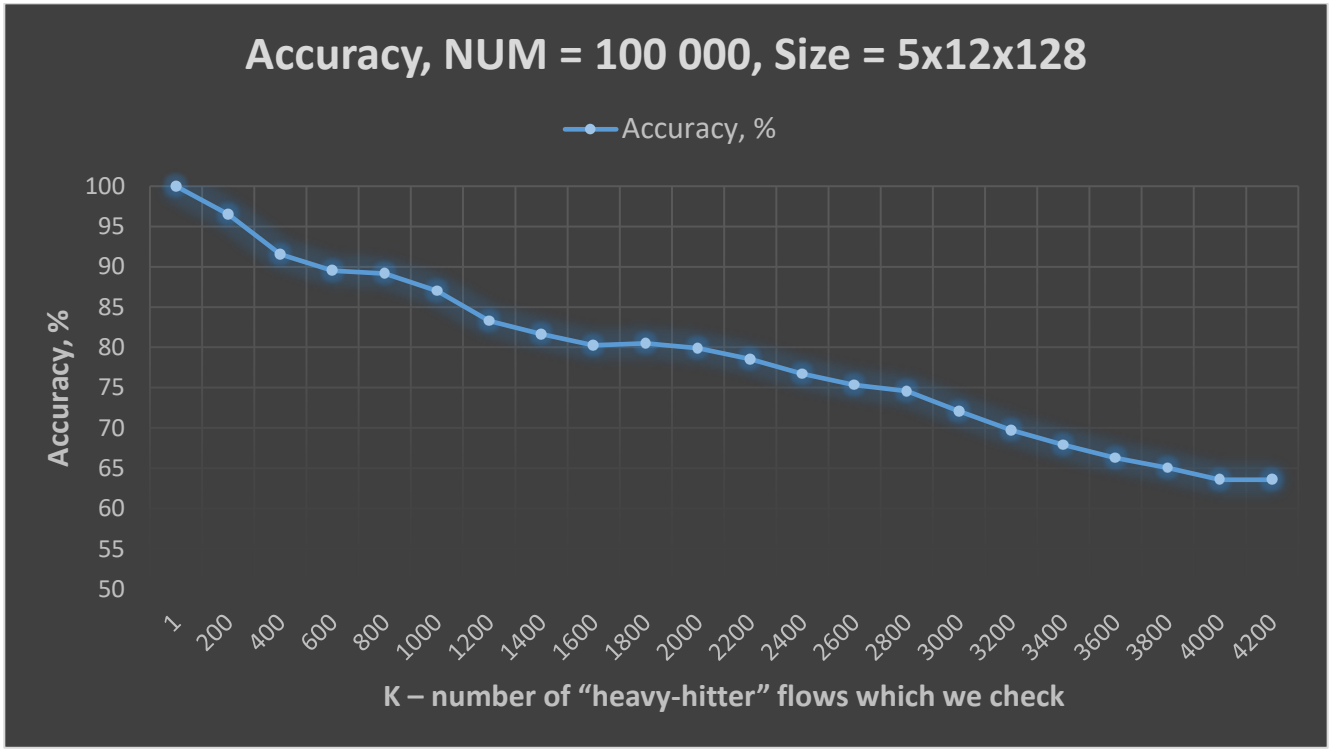
1. **Accuracy** of HashPipe Algorithm (Errors False Negative and False Positive)
2. **Average** place of packets from HashPipe Alg' flows in sorted list of Real "Heavy-Hitter" flows start from heaviest

In addition to **Average** we also display two parameters allowing better analysis of graphs:

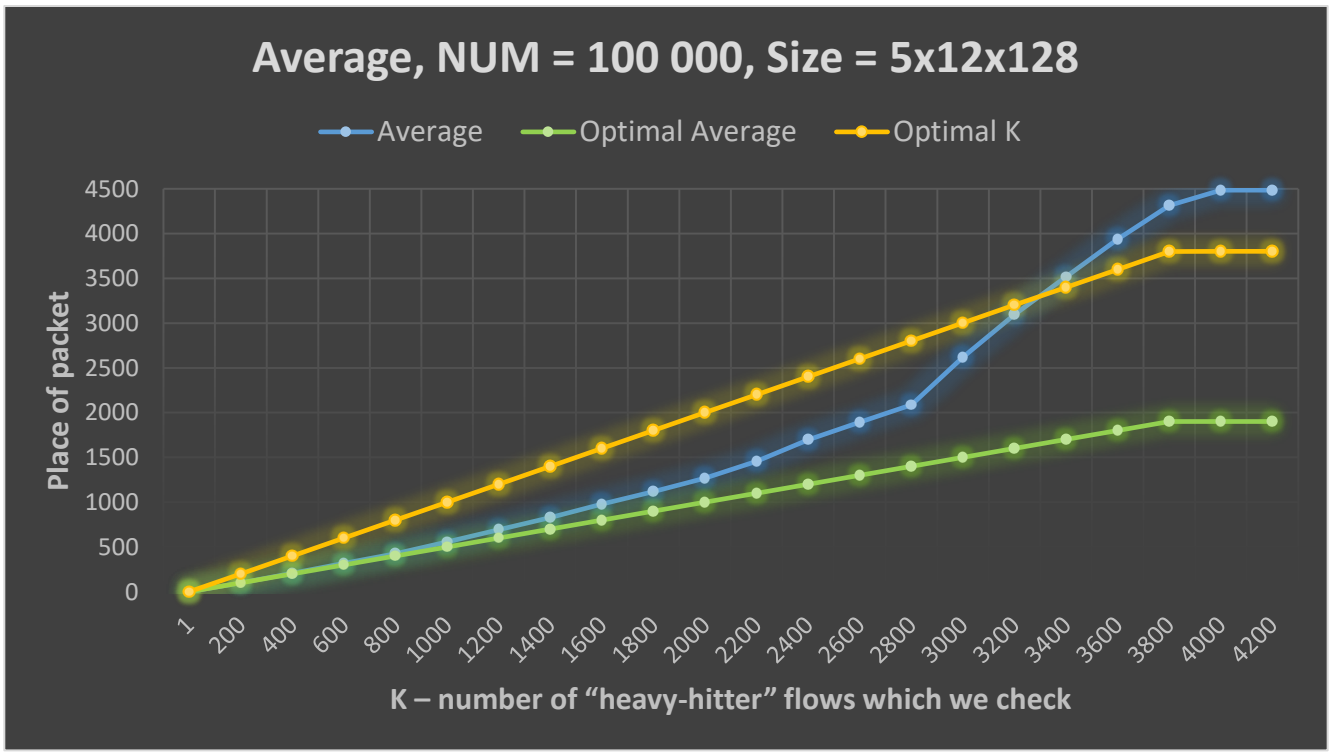
1. **Optimal K** = min(**K**, Number of **HashPipeFlows**, Number of **RealHeavyHitterFlows**) – biggest number of "heavy-hitter" flows which we can check (not bigger than **K**)
2. **Optimal Average** = (Optimal K + 1) / 2 – Average which we get for **Accuracy=100%**

Tests depending on the parameter K

Graph 1.1 - Accuracy

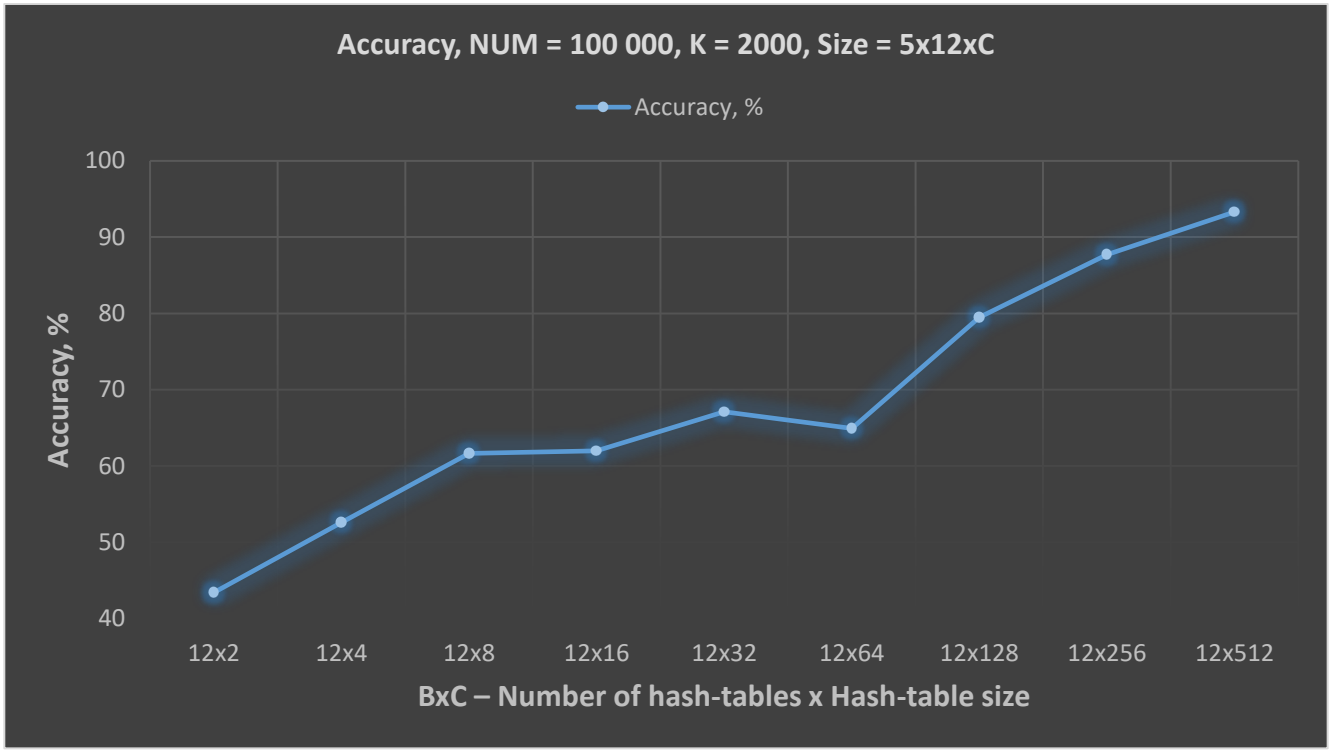


Graph 1.2 - Average

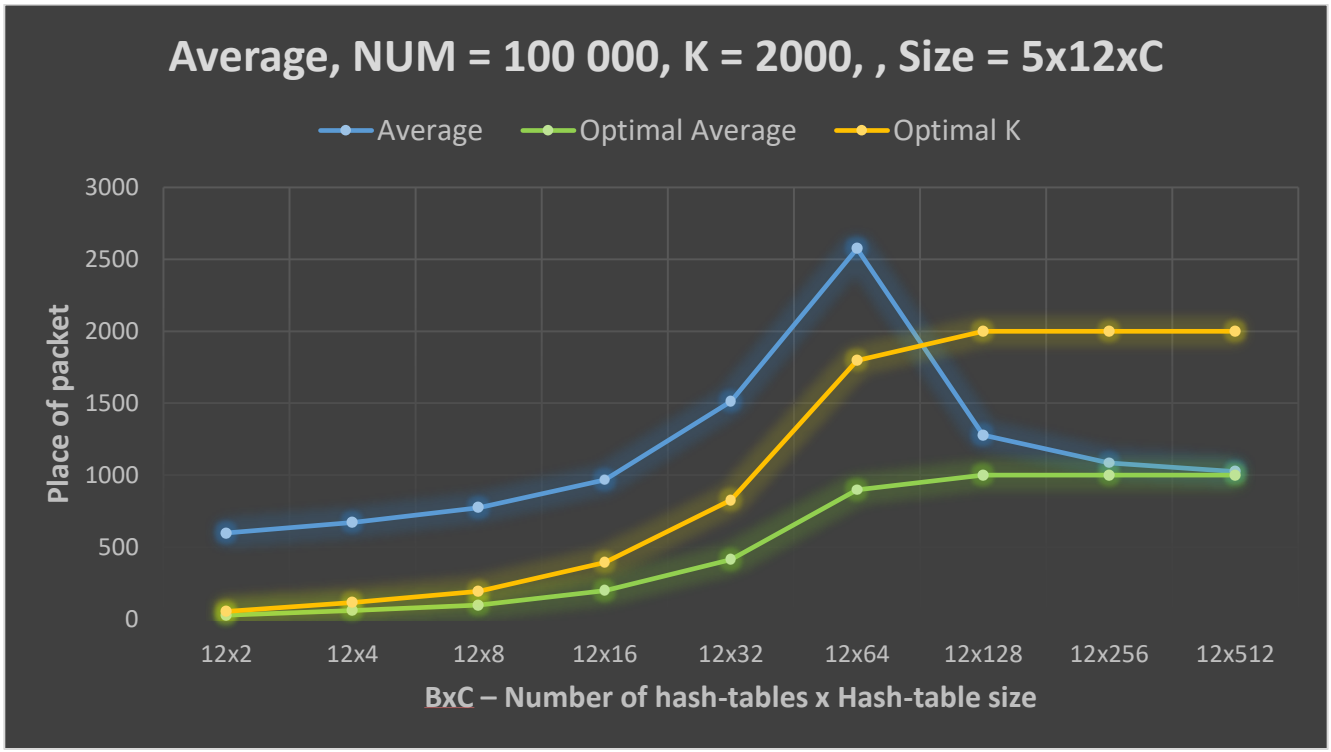


Tests depending on the hash-table size with constant number of hash-tables

Graph 2.1 - Accuracy

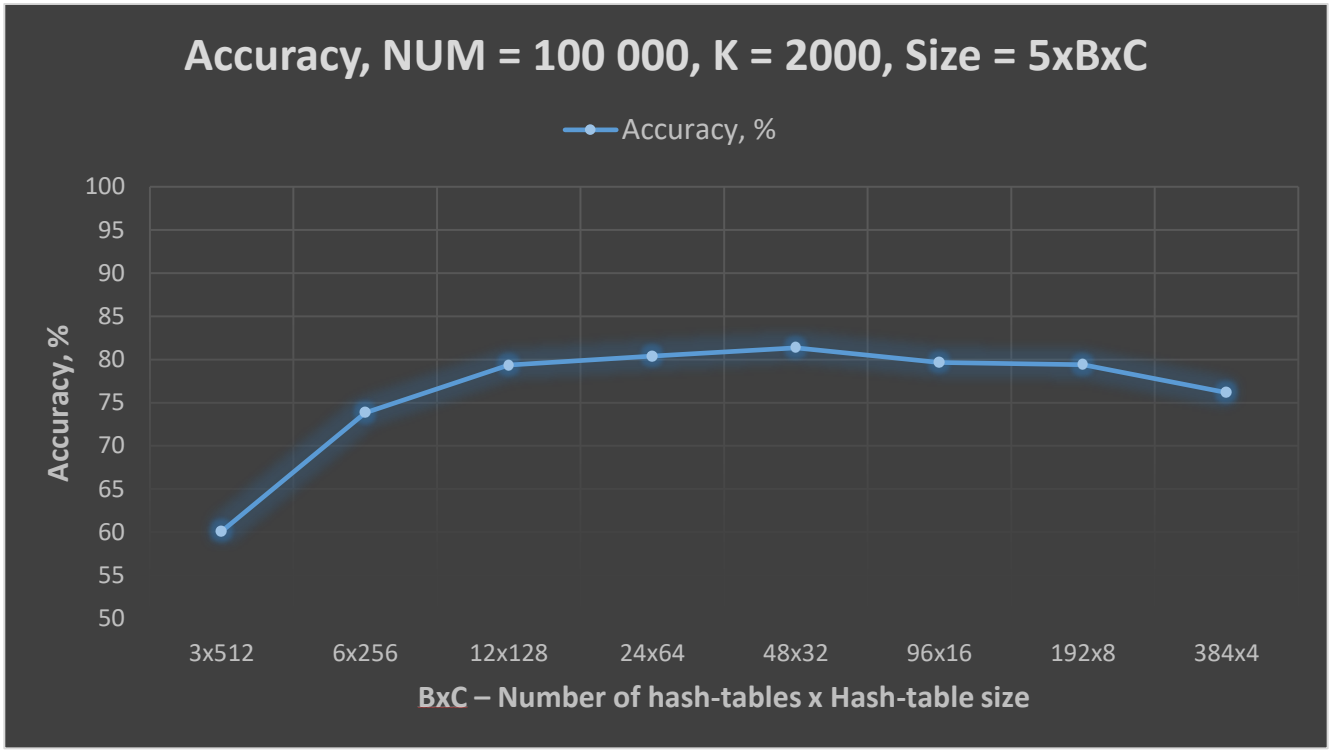


Graph 2.2 - Average

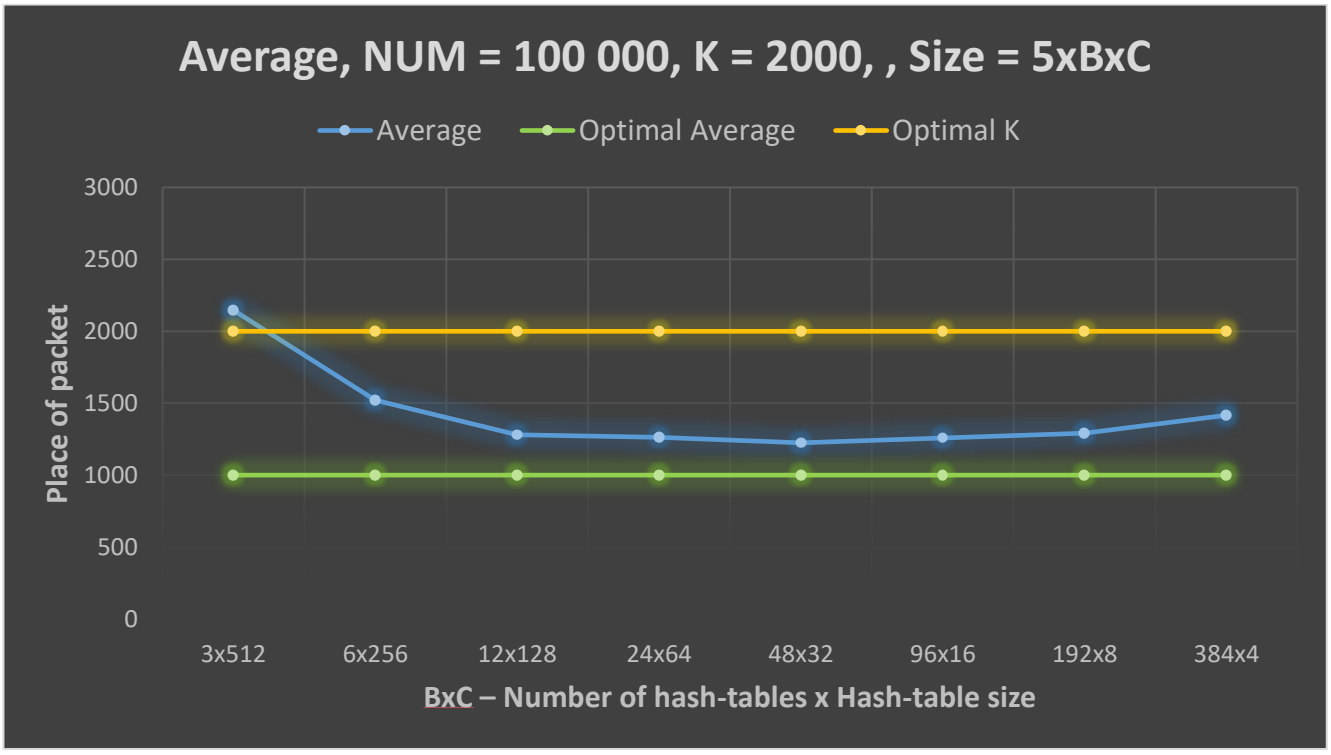


Tests depending on the number of hash-tables and hash-table size with constant memory size

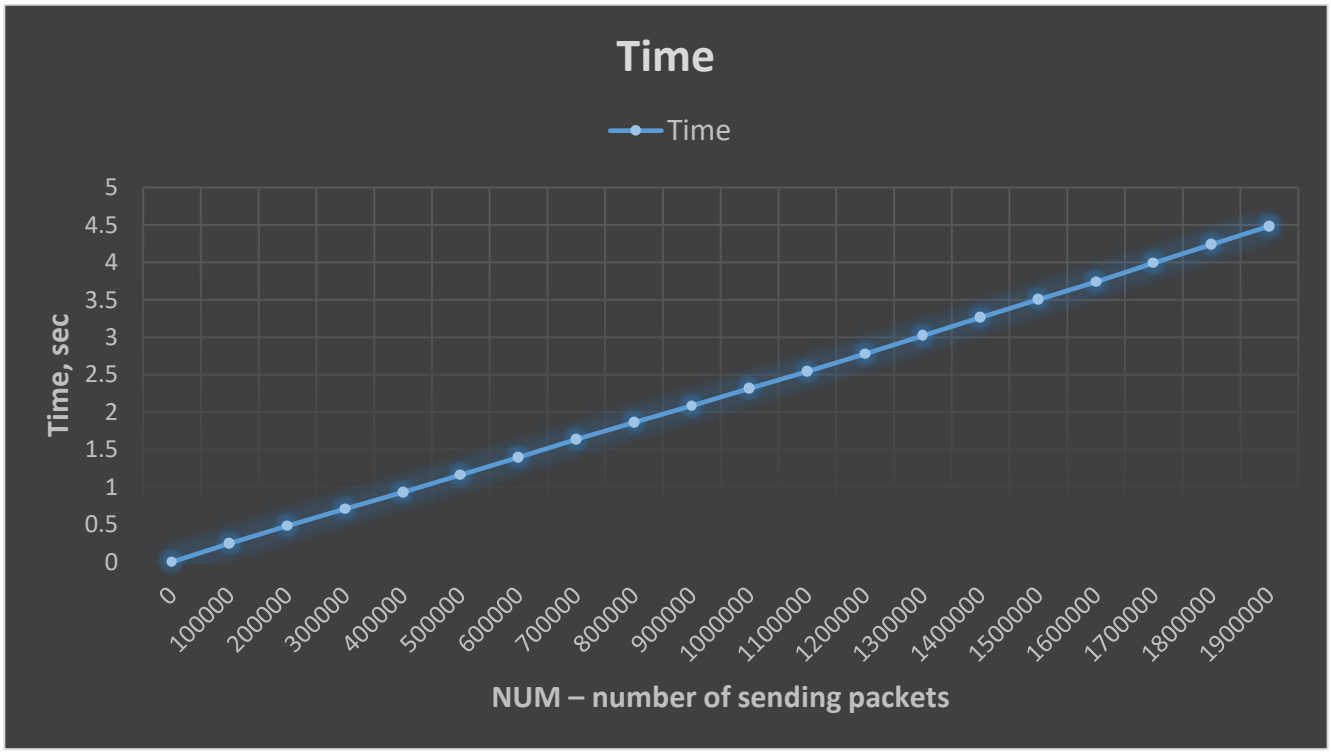
Graph 3.1 - Accuracy



Graph 3.2 - Average



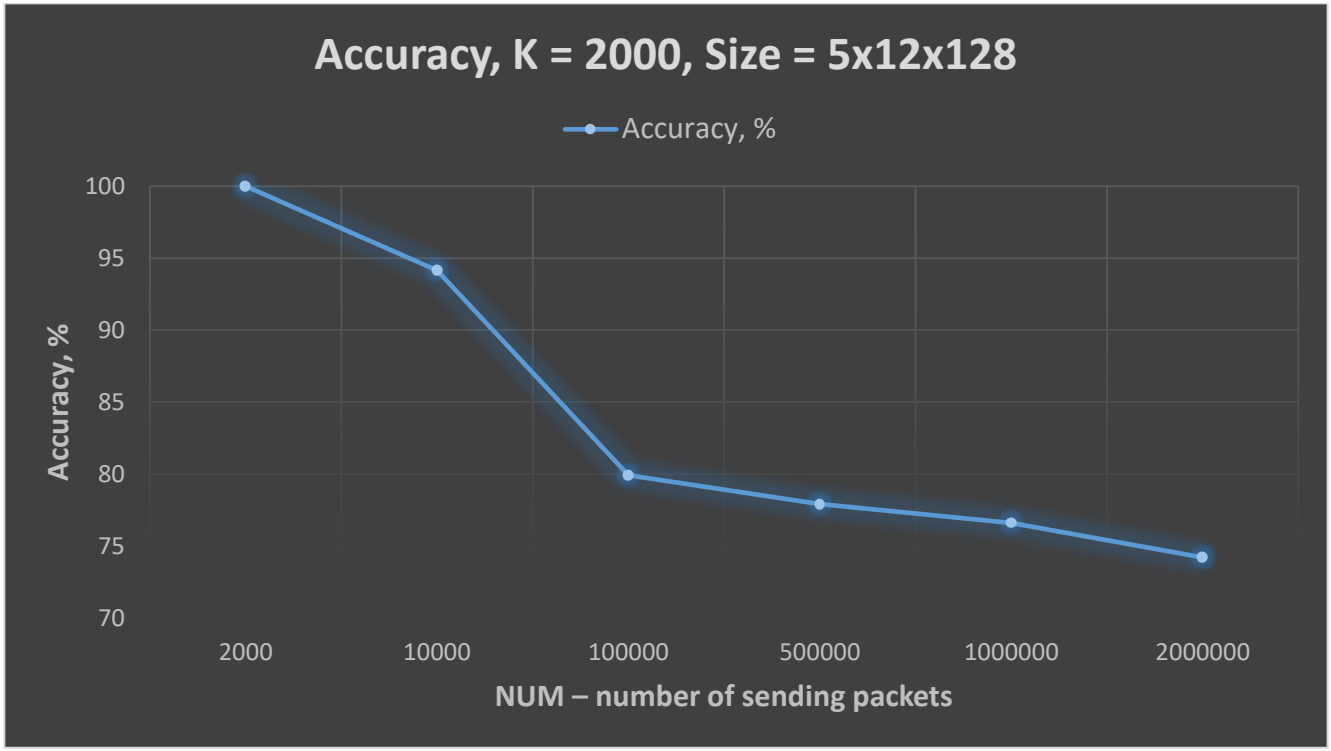
Packet processing time depending on the number of packets



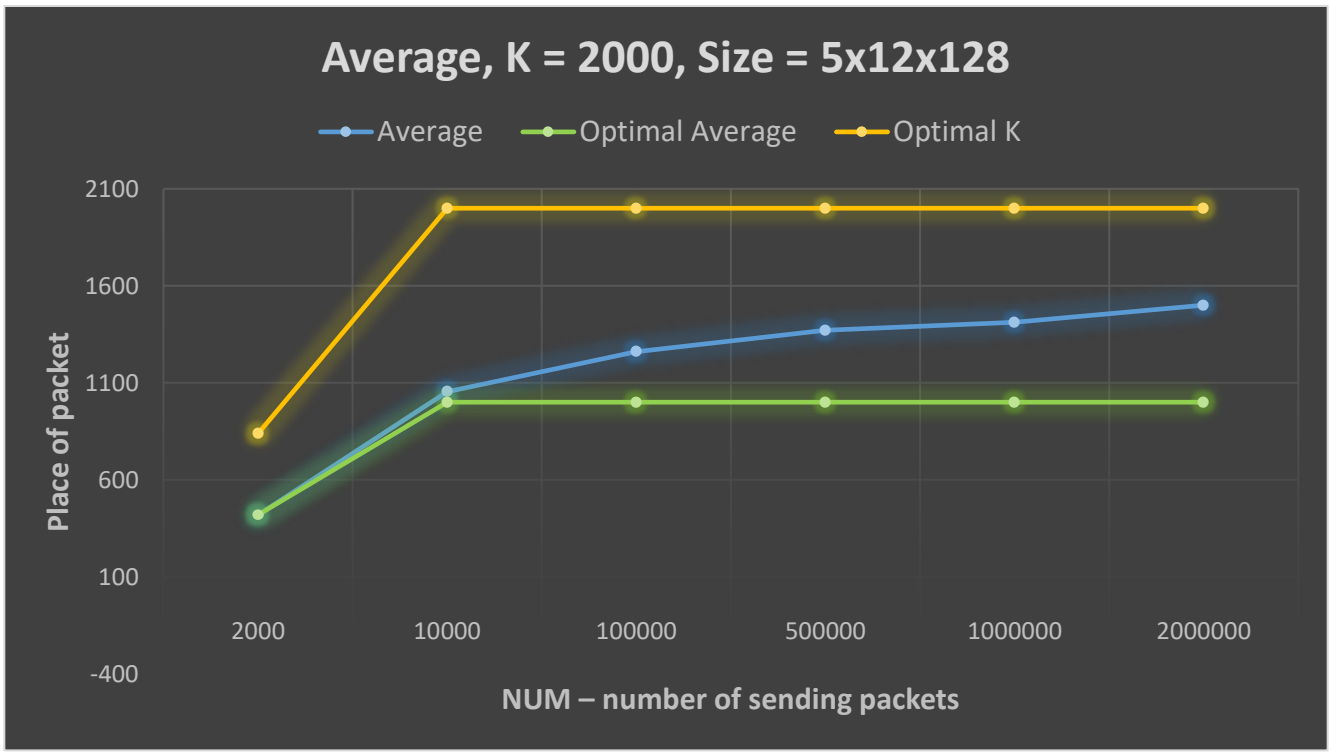
As we can see for a large number of packets, packet processing time is linearly dependent on the number of packets.
Then in spite of the possibly unequal time spent processing different packets, we can consider the packet processing time and packet number parameters as equivalent. And also the graphs that will depend on these parameters will be similar to each other

Tests depending on the parameter NUM

Graph 4.1 - Accuracy



Graph 4.2 - Average



Conclusions

From the constructed graphs we made the following conclusions:

Graphs 1.1 and 1.2

- **Accuracy** is directly proportional to **K** and **Average** is inversely proportional to **K.**
- If value of **K** more than the number of different packets received using the HashPipe Algorithm, then the results become permanent, since we cannot check a larger number of packets than there are in the file.
- The HashPipe Algorithm shows good results for many parameters.
    - For example, for **NUM = 100 000, Memory size = 5x12x128, K = 1000** we got **Accuracy ≈87%.**
    - Even for largest value of **K** such as **4000** we got **Accuracy** not less than **60%.**

Graphs 2.1 and 2.2:

- Accuracy is highly dependent on **memory size**. For example, for standard parameters **NUM=100 000** and **K=2000**:
    - for **memory size = 5x12x4** we got **Accuracy ≈52%**
    - for **memory size = 5x12x128** we got **Accuracy ≈80%**
    - for **memory size = 5x12x512** we got **Accuracy ≈93%**
- For good results, it is important that the **memory size** (number of places for different packets) be significantly larger than **K.** Otherwise, we will not have enough space to save the required number of packets and we will get bad results.
    - As we can see in the graphs from **12x2** to **12x64** we have a fast-growing value of Optimal K (i.e. Optimal K depend on **memory size** and we use all available memory). As a result, we get a low value of **Accuracy** and a fast-growing value of **Average** similarly to Optimal K and **memory size.**
    - In the same time from **12x64** to **12x512** value of Optimal K reaches the value of real **K** and stabilizes (i.e. there is enough space to save all necessary packets). As a result, we get a fast-growing value of **Accuracy** and decreasing value of **Average** (and approaching the value of Optimal Average).

Graphs 3.1 and 3.2:

- For constant memory size best results are achieved when the number of hash-tables and the hash-table size are approximately equal. For example, for **memory size = 5x48x32** we got **Accuracy ≈81%**
- On the other hand, if one of the parameters is greatly reduced, then the results will deteriorate. For example:
    - for **memory size = 5x3x512** we got **Accuracy ≈60%**
    - for **memory size = 5x384x4** we got **Accuracy ≈76%**

Graphs 4.1 and 4.2:

- For a large number of packets, we can assume that the graphs, depending on the number of packets, are identical to the graphs of the processing time of these packets
- The number of sent (**NUM**) packets affects the results much weaker than **K** or **memory size**
- But **NUM** strongly affects the execution time of the program.

Learning and Future Directions

Learning

- Learned to work with **Netronome SmartNIC**
- Learned to use IDE **Programmer Studio 6.0**
- Practiced writing code in the programming languages **P4**, **C** and **Python**
- Practiced using console commands to work with traffic and with SmartNIC (**tcpreplay**, **tcpdump**, **nfp-rtsym**)

Future Directions

For HashPipe Algorithm:

- In the future, should compare the HashPipe Algorithm with older and known algorithms, check in which cases it gives a better result and in which not, find weaknesses of the Algorithm
- It is also worth analyzing the dependence of the accuracy of the HashPipe Algorithm on the choice of hash-functions

For Program:

- Develop a more universal way of processing data obtained from Netronome SmartNIC for applying the program to different types of packets
- Find a way to zero Netronome SmartNIC memory and restart recording via the command **tcpdump** for more convenient, quick and automatic use of the program

References

1. Ori Rottenstreich, Srinivas Narayana, Vibhaalakshmi Sivaraman, Jennifer Rexford, S. Muthukrishnan, “Heavy-Hitter Detection Entirely in the Data Plane”, 2017
2. https://github.com/open-nfpsw/M-Sketch
3. Xiaoban Wu, Yan Luo, “Network Measurement with P4 and C on Netronome NFP”
4. https://github.com/open-nfpsw/p4_basic_lb_metering_nic

For P4 language learning:

5. Noa Zilberman, “P4 Tutorial Welcome”
6. Mihai Budiu, Chris Dodd, “The P4 Programming Language”, 2017
7. https://github.com/p4lang/tutorials