PRECISION- Final report

Shir Sason(207548231), Gal Markovich(322217456)

Introduction

Programmable network switches must stand some strict limitations, and therefore promise flexibility and high throughput, which are very important when working with large data traffic. These limitations, such as restricted branching, limited capability for memory access and a limited number of processing stages, restrict the types of measurement algorithms that can run on programmable switches.
This paper introduce PRECISION, an algorithm that aims to find top flows on a programmable switch, by probabilistically recirculating packets in order to simplify the access to stateful memory, stand the RMT limitations and achieves higher accuracy than previous heavy hitter detection algorithms that avoid recirculation.

Algorithms- Implementation Details

Utils

In order to Implement the algorithms described in the paper, first we implemented the following structures:

- Packet:
  Each packet contains the flowID of the flow that it belongs to, as well as a flag that determine if a packet is being recirculated or not. In addition, each recirculated packet should know which stage contains the min counter, and should return to this stage to affect the relevant counter in the Precision algorithm.

```
type Packet struct {  10 usages
    flowID          int
    isRecirculated  bool
    currStageNum    int
    newVal          int
}
```

- Stage:
  Each stage has its own keys and values arrays (separated RAM sections). In addition, each stage has its own hash function. The matched flag specifies if a packet was matched in the current stage, this is used in the Precision algorithm when deciding if a packet should be recirculated.

```
type stage struct {  8 usages
    keys     []int
    values   []int
    matched  bool
    index    int
    hashFunc func(flowID int) int
}
```

- Pipeline:
  The pipeline contains an array of stages and an IncomingPacket channel for receiving packets. It has arrays of keys and values which include the stages' memory arrays. In addition, it has a matched flag for declaring if a packet was matched during its processing.

```
type pipeline struct {  7 usages
    keys           []int
    values         []int
    matched        bool
    stages         []stage
    incomingPacket chan Packet
}
```

HashPipe

A previous Heavy Hitter algorithm. Each packet goes through the d pipeline stages once. The algorithm has High throughput, as it passes a large packet traffic, but it has lower accuracy.
Additionally, the HashPipe algorithm can't stand RMT limitations, and therefore is not a good solution for solving the Heavy hitters problem using a programmable switch.

The counters are distributed in the d stages. When a new packet arrives it is always inserted to the first stage. The flow with the bigger counter stays, the lowest is moved to the next stage. If we couldn't match the packet flowID to a flow we already seen- The flow with the lowest counter will be evicted after the last stage.

Algorithm 1: HashPipe [35] heavy hitter algorithm

$l_1 \leftarrow h_1(iKey)$ ▷ Always insert in the first stage;
if $key_1[l_1] = iKey$ **then**
  $val_1[l_1] \leftarrow val_1[l_1] + 1$;
  end processing;
else if l_1 *is an empty slot* **then**
  $(key_1[l_1], val_1[l_1]) \leftarrow (iKey, 1)$;
  end processing;
else
  $(cKey, cVal) \leftarrow (key_1[l_1], val_1[l_1])$;
  $(key_1[l_1], val_1[l_1]) \leftarrow (iKey, 1)$;
  ▷ Track a rolling minimum;
for $i \leftarrow 2$ ***to*** d **do**
  $l_i \leftarrow h_i(cKey)$;
  if $key_i[l_i] = cKey$ **then**
    ▷ Read key_i ;
    $val_i[l_i] \leftarrow val_i[l_i] + cVal$;
    ▷ R/W val_i ;
    end processing;
  else if $[l_i]$ *is an empty slot* **then**
    $(key_i[l_i], val_i[l_i]) \leftarrow (cKey, CVal)$;
    ▷ Read key_i, val ;
    end processing;
  else if $val_i[l_i] < cVal$ **then**
    ▷ Condition on val_i; Violating Restriction I;
    swap $(cKey, cVal) \Leftrightarrow (key_i[l_i], val_i[l_i])$;
    ▷ R/W key_i ;

Explanation of our Implementation based on the pseudo-code in the paper - (functions *runHashPipe(), processHashPipe()*):
- First, we initialized the pipeline and sent packets to through the incomingPacket channel.
- Then we processed the packet using *processHashPipe()* function:

We inserted the packet to the first stage; if the slot is empty or matches the packet's flowID, we simply increment the flow's counter. If there is no match, We propagate the flow with the lowest counter until we can insert it in the next stages, or the flow with the lowest counter will be evicted.

PRECISION

An algorithm that uses Probabilistic Recirculation to find top flows on a programmable switch. By recirculating a small fraction of packets, PRECISION simplifies the access to stateful memory to conform with RMT limitations and achieves higher accuracy than previous heavy hitter detection algorithms that avoid recirculation.

Algorithm 2: PRECISION heavy hitter algorithm

for $i \leftarrow 1$ ***to*** d **do**
  $l_i \leftarrow h_i(iKey)$;
  $*$ Hardware stage i_A: access key_i register;
  if $key_i[l_i] = iKey$ **then**
    $matched_i \leftarrow true$;
  $*$ Hardware stage i_B: access val_i register;
  if $matched_i$ **then**
    $val_i[l_i] \leftarrow val_i[l_i] + 1$;
  else
    $oval_i = val_i[l_i]$
  $*$ Hardware stage i_C: maintain carry minimum;
  if $(\neg matched_i) \wedge (oval_i < carry\_min)$ **then**
    $carry\_min \leftarrow oval_i$;
    $min\_stage \leftarrow i$
if $\bigwedge_{i=1}^{d}(\neg matched_i)$ **then**
  ▷ $iKey$ not in cache; do Probabilistic Recirculation.
  $new\_val = 2^{\lceil \log_2(carry\_min) \rceil}$;
  Generate random integer $R \in [0, new\_val - 1]$, by assembling $\lceil \log_2(carry\_min) \rceil$ random bits;
  if $R = 0$ **then**
    clone and recirculate packet;
if *packet is recirculated* **then**
  $i \leftarrow min\_stage$;
  $l_i \leftarrow h_i(iKey)$;
  $key_i[l_i] \leftarrow iKey$ ▷ Hardware stage i_A: key_i ;
  $val_i[l_i] \leftarrow new\_val$ ▷ Hardware stage i_B: val_i;
  Drop the cloned copy;

Explanation of our Implementation based on the pseudo-code in the paper- (functions *runPrecision(), process()*):
- First, we initialized the pipeline and sent packets to through the incomingPacket channel.
- Then we process the packet using process() function:

Packets go through the pipeline for the first time, trying to match its flowID to an existing flow's ID. If matched, we increment the flow's counter. If the pipeline finished(= no match) - maybe pass the packet in the pipeline again, to claim an entry with the new flow's ID.
If the packet was not matched in any stage, then we generate a random number R and the packet will do a second round in the pipeline with probability of 1/carry_min+1.

Evaluation

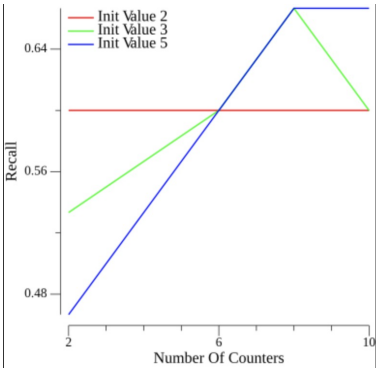
Top-k

In this task we want to see how different **init values affect the Recall score** for solving the Top-15 task. In our experiment we used the following setup:

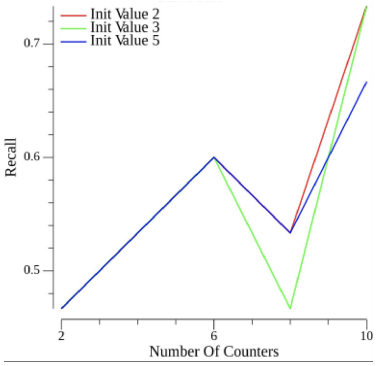
- init values - we experimented with the numbers [2, 3, 5]
- number of stages in the pipeline (d) = 2
- Flow frequency: [800, 700, 3, 803, 800, 400, 804, 7, 90000, 350, 900, 720, 17000, 300, 905, 40000, 6, 250, 600, 850, 750, 5, 71, 42, 800, 40000, 6, 250, 600, 850, 750, 5, 71, 42, 800, 40000, 6, 250, 600, 850, 750, 5, 71, 42, 800] - so we worked with 45 flows, 5 of them where the main Heavy hitters, and 17 flows were of medium size.
- number of top flows we wish to find (k) = 15

Note - as we did not have an actual switch simulator, we built our own simulation environment, and therefore we did not use the same testing environment as the one used in the paper experiment. In addition, we experimented with a relatively low amount of packets, compared to the paper results. These two changes affect our results, and thus we can expect to receive results that will not be the same as the results shown in the paper. However, we did expect to see a similar trend in our graphs.

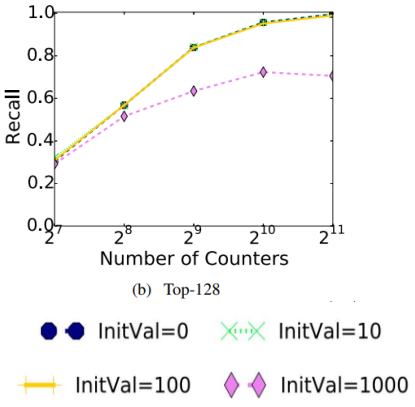
(a) HashPipe



(b) Precision



(c) Precision- from the paper

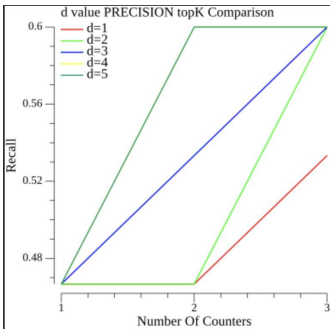


In the 3 graphs above we can see the effect of the init values on the recall score, when experimenting with different numbers of counters.
Graph (a) shows how different init values affect the recall score using the hash pipe algorithm. We chose to Include this graph out of curiosity, we wanted to see if the trend would be similar between hash pipe and PRECISION.
By comparing Graph (a) and Graph (b), we can see that even though the graphs are different, both explain the same result- 5 is the best init value for our experiment.
We can also see in general that the recall score increases as the number of counters grows, both in the paper experiment and in ours.

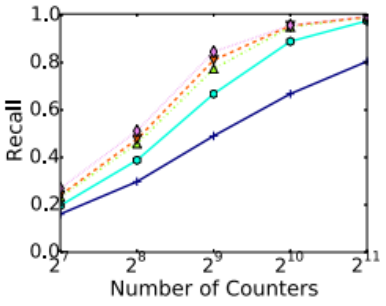
In addition, we want to see how different **d values affect the Recall score** for solving the Top-15 task. We used the following setup:

- d - we experimented with the numbers [1,2,3,4,5]
- init values - 5
- k - 15
- Flow frequency: [9, 70000, 1, 2, 12000, 38, 2, 25670, 9, 19, 10, 8] - so we worked with 12 flows, 3 of them were the main Heavy hitters, and 3 flows were of medium size.

a) Precision



b) Precision- from the paper



(b) Top-128

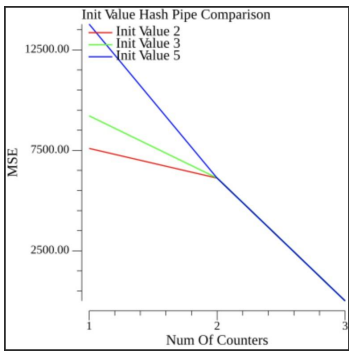
As we expect, there is an increasing trend like in the original graph from the paper.
In both graphs we can see that as the number of counters grows, the Recall increases. Furthermore, Using d = 5 is a right balance between achieving good accuracy and saving pipeline stages usage.

Frequency Estimation

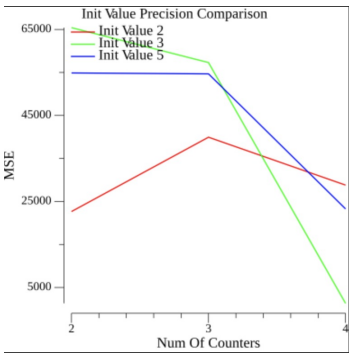
In this task we want to see how different **init values affect the MSE score** for solving the frequency estimator task. In our experiment we used the following setup:

- init values - we experimented with the numbers [2, 3, 4]
- number of stages in the pipeline (d) = 2
- Flow frequency: [9, 70000, 1, 2, 12000, 38, 2, 25670, 9, 19, 10, 8] - so we worked with 12 flows, 3 of them were the main Heavy hitters, and 3 flows were of medium size.

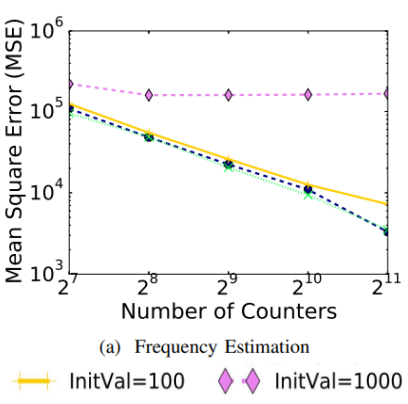
(a) HashPipe-



(b) Precision-



(c) Precision- from the paper

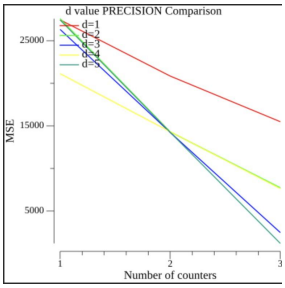


In the 3 graphs above we can see the effect of the init values on the MSE, when experimenting with different numbers of counters.
We can see in general that the MSE is getting lower as the number of counters is growing both in the HashPipe and Precision, similar to the graph of the paper.

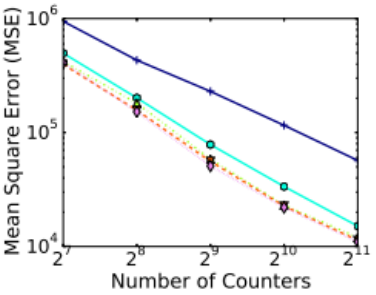
In addition, we want to see how different **d values affect the MSE score** for solving the frequency estimator task. We used the following setup:

- d - we experimented with the numbers [1,2,3,4,5]
- init values - 5
- Flow frequency: [9, 70000, 1, 2, 12000, 38, 2, 25670, 9, 19, 10, 8] - so we worked with 12 flows, 3 of them were the main Heavy hitters, and 3 flows were of medium size.

a) Precision



b)Precision-from the paper



(a) Frequency Estimation

As we expect, there is a decreasing trend like in the original graph from the article. In both graphs we can see that as the number of counters grows, the MSE decreases.
Furthermore, Using d = 3 is a right balance between achieving good accuracy and saving pipeline stages usage.

Research Chapter- AI & Programmable Switches

Artificial Intelligence models can be very large, and most likely training such models using only one GPU is not feasible. There is a crucial need for thousands of GPUs that will be working simultaneously and in synchronization with each other, as large amounts of data is being passed from one GPU to another non - stop.

In order to be able to transfer data efficiently, we think that knowing which “data flows” have large data traffic, as well as knowing the GPU workload status, can be beneficial for directing the data to the most available GPUs, and prevent bottlenecks or inefficient data processing.

We think that using a traffic analysis algorithm, such as a Heavy Hitters detection algorithm, can be beneficial for efficient traffic direction between GPUs.

As we saw in the paper, since algorithms working on programmable switches must stand in the RMT strict limitations, they promise high performance, and therefore enabling applications such as load balancing and traffic engineering. As the PRECISION solution allows us to solve the Heavy Hitters detection problem using a programmable switch, we believe that a combination of the PRECISION algorithm and the advantages of programmable switches can be a game changer in controlling traffic between GPUs for complex AI applications.

Conclusions

In our report we performed experiments trying to reproduce the graphs from the original paper. Since we did not have a switch simulator, most of our work was developing a simulation environment for packet streaming and processing while standing the strict RMT limitations. During this task we faced the challenges of working with the RMT limitations.

In this report we compared the results we obtained using the PRECISION algorithm to the results of Hash Pipe, and as we expected - PRECISION performed very well and received similar accuracy.

The need for this study is emphasized since other algorithms for solving the Heavy Hitter detection problem do not stand the RNT limitations and therefore could not be implemented in a programmable switch. To the best of our knowledge, PRECISION is the first heavy hitter detection algorithm tailored for the RMT architecture, that gets accuracy scores compatible with the scores of current algorithms that could not be implemented on a programmable switch.

While working on this project we learned and researched a lot about programmable switches, which is an un-familier field for us. As we dived deeper and learned more we got more excited to know that this work is an important step forward for measurements on high-performance programmable switches and it is a big technological progress.

.