Heavy-Hitter Detection
Entirely in the Data Plane

VIBHAALAKSHMI SIVARAMAN
SRINIVAS NARAYANA, ORI ROTTENSTREICH, MUTHU MUTHUKRSISHNAN, JENNIFER REXFORD
Heavy Hitter Flows

Flows above a certain threshold of total packets

“Top-$k$” flows by size
Why detect heavy hitters?

Trouble-shooting and anomaly detection

Dynamic routing or scheduling of heavy flows

<table>
<thead>
<tr>
<th>Flow</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>100</td>
</tr>
<tr>
<td>f2</td>
<td>75</td>
</tr>
<tr>
<td>f3</td>
<td>5</td>
</tr>
</tbody>
</table>
Problem Statement

Restrict processing to data plane

Low data plane state

High accuracy

Line-rate packet processing
Emerging Programmable Switches

Programmable switches with stateful memory
Basic arithmetic on stored state
Pipelined operations over multiple stages
State carried in packets across stages
Constraints

Small, deterministic time budget for packet processing at each stage

Limited number of accesses to stateful memory per stage

Limited amount of memory per stage

No packet recirculation
## Existing Work

<table>
<thead>
<tr>
<th>Technique</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling-based (Netflow, sflow, Sample &amp; Hold)</td>
<td>Small “flow memory” to track heavy flows</td>
<td>Underestimates counts for heavy flows</td>
</tr>
<tr>
<td>Sketching-based (Count, Count-Min, Reversible)</td>
<td>Statistics for <em>all</em> flows in single data structure</td>
<td>No flow identifier to count association</td>
</tr>
<tr>
<td>Counting-based (<em>Space Saving</em>, Misra-Gries)</td>
<td>Summary structure with heavy flow ids and counters</td>
<td>Occasional updates to multiple counters</td>
</tr>
</tbody>
</table>
Motivation: Space-Saving Algorithm\textsuperscript{1}

\(O(k)\) space to store heavy flows

Provable guarantees on accuracy

Evict the minimum to insert new flow

Multiple reads but exactly one write per packet

**Space Saving Algorithm**

<table>
<thead>
<tr>
<th>Flow Id</th>
<th>Packet Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>4</td>
</tr>
<tr>
<td>K2</td>
<td>2</td>
</tr>
<tr>
<td>K3</td>
<td>7</td>
</tr>
<tr>
<td>K4</td>
<td>10</td>
</tr>
<tr>
<td>K5</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flow Id</th>
<th>Packet Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>4</td>
</tr>
<tr>
<td>K2</td>
<td>2</td>
</tr>
<tr>
<td>K3</td>
<td>7</td>
</tr>
<tr>
<td>K4</td>
<td>10</td>
</tr>
<tr>
<td>K6</td>
<td>2</td>
</tr>
</tbody>
</table>

*High accuracy
Exactly one write*

*Entire table scan
Complex data structures*
Towards HashPipe

<table>
<thead>
<tr>
<th>Technique</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space-Saving</td>
<td>High accuracy; Exactly one write-back</td>
<td>Entire table scan; Complex data structures</td>
</tr>
<tr>
<td>HashParallel</td>
<td>Sample fixed number of locations; Approximate minimum</td>
<td>Multiple reads per stage; Dependent write-back</td>
</tr>
<tr>
<td>Sequential Minimum Computation</td>
<td>Hash table spread across multiple stages; Sample one location per stage</td>
<td>Multiple passes through the pipeline</td>
</tr>
</tbody>
</table>
Our Solution - HashPipe

Always insert new key in the first stage
Hash to index to a location
 Carry evicted key to the next stage

Stage 1
- A: 5
- B: 6
- C: 10
- K1: 4

Stage 2
- K2: 3
- D: 15
- E: 25
- F: 100

Stage 3
- G: 4
- K3: 3
- H: 10
- I: 9

New key K

$h_1(K) \rightarrow K1$
Our Solution - HashPipe

At each later stage, carry current minimum key
Hash on carried key to index to a location
Compare against key in location for local minimum

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 5</td>
<td>D 3</td>
<td>G 4</td>
</tr>
<tr>
<td>K 1</td>
<td>E 15</td>
<td>K3 3</td>
</tr>
<tr>
<td>B 6</td>
<td>K2 25</td>
<td>H 10</td>
</tr>
<tr>
<td>C 10</td>
<td>F 100</td>
<td>I 9</td>
</tr>
</tbody>
</table>

(K1, 4)
At any table stage, retain the heavier hitter

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 5</td>
<td>D 3</td>
<td>G 4</td>
</tr>
<tr>
<td>K 1</td>
<td>E 15</td>
<td>K3 3</td>
</tr>
<tr>
<td>B 6</td>
<td>F 100</td>
<td>H 10</td>
</tr>
<tr>
<td>C 10</td>
<td></td>
<td>I 9</td>
</tr>
</tbody>
</table>

\( h_2(K1) \rightarrow K2 \)

\( \max(K1, K2) \rightarrow K2 \)
HashPipe

At any table stage, retain the heavier hitter

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>G</td>
</tr>
<tr>
<td>K</td>
<td>E</td>
<td>K3</td>
</tr>
<tr>
<td>B</td>
<td>K2</td>
<td>H</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>I</td>
</tr>
</tbody>
</table>

5 | 3 | 4
1 | 15 | K3
6 | 25 | 10
10 | 100 | 9

\(h_3(K1) \rightarrow K3\)
\(Max(K1, K3) \rightarrow K1\)
HashPipe

At any table stage, retain the heavier hitter
Eventually evict a relatively small flow

Stage 1
A  5
K  1
B  6
C 10

Stage 2
D  3
E 15
K2 25
F 100

Stage 3
G  4
K1  4
H 10
I  9

High accuracy
Single pass
One read/write per stage

Duplicates
HashPipe Summary

Split hash table into $d$ stages

<table>
<thead>
<tr>
<th>Condition</th>
<th>Stage 1</th>
<th>Stages 2 - $d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>Insert with value 1</td>
<td>Insert key and value carried</td>
</tr>
<tr>
<td>Match</td>
<td>Increment value by 1</td>
<td>Coalesce value carried with value in table</td>
</tr>
<tr>
<td>Mismatch</td>
<td>Insert new key with value 1, evict and carry key in table</td>
<td>Keep key with higher value and carry the other</td>
</tr>
</tbody>
</table>
## Implementation

### Prototyped on P4

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>K2</td>
<td>G</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>K1</td>
<td>D</td>
<td>H</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>K3</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>9</td>
</tr>
</tbody>
</table>

- **Hash on packet header**
- **Packet metadata**
- **Register arrays**

New key K

Conditional updates to compute minimum

Register arrays

(The P4 Register arrays are used to store the metadata of the packets, which is hashed on the packet header. This allows for efficient conditional updates to compute the minimum value.)
Evaluation Setup

Top-$k$ 5 tuples on CAIDA traffic traces with 500M packets

50 trials, each 20 s long with 10M packets and 400,000 flows

Memory allocated: 10 KB to 100 KB; $k$ value: 60 to 300

Metrics: false negatives, false positives, count estimation error
Tuning HashPipe

-$k = 210$

5040 flowids maintained in table
HashPipe Accuracy

5-10% false negatives for detecting heavy hitters
HashPipe Accuracy

5-10% false negatives for the detecting heavy hitters

4500 flow counters on traces with 400,000 flows
5-10% false negatives for the detecting heavy hitters
4500 flow counters on traces with 400,000 flows
Competing Schemes

Sample and Hold
- Sample packets of new flows
- Increment counters for all packets of a flow once sampled

Count-Min Sketch
- Increment counters for every packet at $d$ hashed locations
- Estimate using minimum among $d$ location
- Track heavy hitters in cache
HashPipe vs. Existing Solutions

![Graph showing False Negative % vs. Memory (in KB)]
HashPipe vs Existing Solutions
HashPipe vs Existing Solutions

![Graph showing comparison of False Negative percentage vs Memory (in KB) for HashPipe, Count-Min + Cache, and Sample and Hold solutions.]
Contributions and Future Work

Contributions:
- Heavy hitter detection on programmable data planes
- Pipelined hash table with preferential eviction of smaller flows
- P4 prototype - https://github.com/vibhaa/iw15-heavyhitters

Future Work:
- Analytical results and theoretical bounds
- Controlled experiments on synthetic traces
THANK YOU

vibhaa@princeton.edu
Backup Slides
P4 prototype – Stage 1

```c
action doStage1()
{
  mKeyCarried = ipv4.srcAddr;
  mCountCarried = 0;
  modify_field_with_hash_based_offset (mIndex, 0,
    stage1Hash, 32);

  // read the key and value at that location
  mKeyTable = flowTracker[mIndex];
  mCountTable = packetCount[mIndex];
  mValid = validBit [mIndex];

  // check for empty location or different key
  mKeyTable = (mValid == 0) ? mKeyCarried : mKeyTable;
  mDiff = (mValid == 0) ? 0 : mKeyTable - mKeyCarried;

  // update hash table
  flowTracker[mIndex] = ipv4.srcAddr;
  packetCount[mIndex] = (mDiff == 0) ? mCountTable+1 : 1;
  validBit [mIndex] = 1;

  // update metadata carried to the next table stage
  mKeyCarried = (mDiff == 0) ? 0 : mKeyTable;
  mCountCarried = (mDiff == 0) ? 0 : mCountTable;
}
```
action doStage2 {
    ....
    mKeyToWrite = (mCountInTable < mCountCarried) ?
        mKeyCarried : mKeyTable);
    flowTracker[mIndex] = (mDiff == 0) ? mKeyTable :
        mKeyToWrite;
    mCountToWrite = (mCountTable < mCountCarried) ?
        mCountCarried : mCountTable;
    packetCount[mIndex] = (mDiff == 0) ? (mCountTable +
        mCountCarried) : mCountToWrite;
    mBitToWrite = (mKeyCarried == 0) ? 0 : 1);
    validBit[mIndex] = (mValid == 0) ? mBitToWrite : 1);
    ....
}
HashPipe vs Idealized Schemes

Performance of three schemes is comparable
HashPipe may outperform SpaceSaving
Programmable Switches

New switches that allow us to run novel algorithms

Barefoot Tofino, RMT, Xilinx, Netronome, etc.

Languages like P4 to program the switches