Heavy-Hitter Detection Entirely in the Data Plane

VIBHAALAKSHMI SIVARAMAN

SRINIVAS NARAYANA, ORI ROTTENSTREICH, MUTHU MUTHUKRSISHNAN, JENNIFER REXFORD

PRINCETON UNIVERSITY

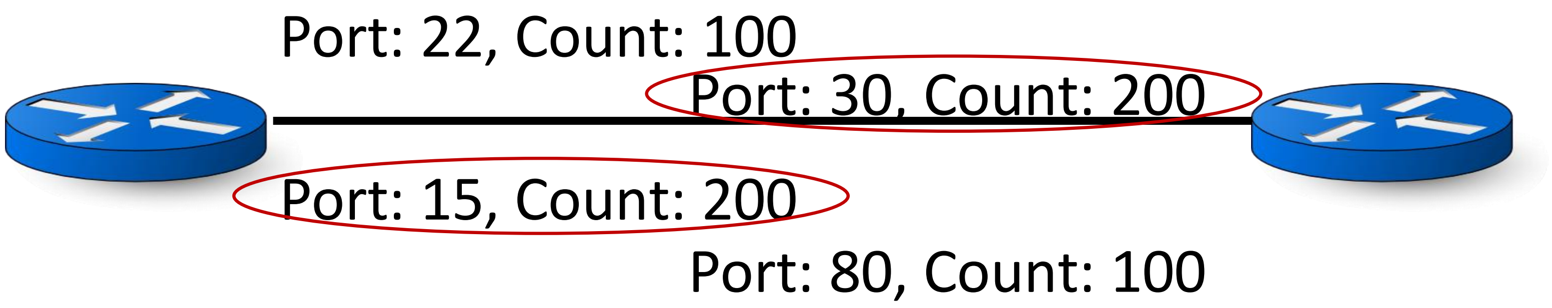
Massachusetts Institute of Technology

RUTGERS

Heavy Hitter Flows

Flows above a certain threshold of total packets

“Top-*k*” flows by size

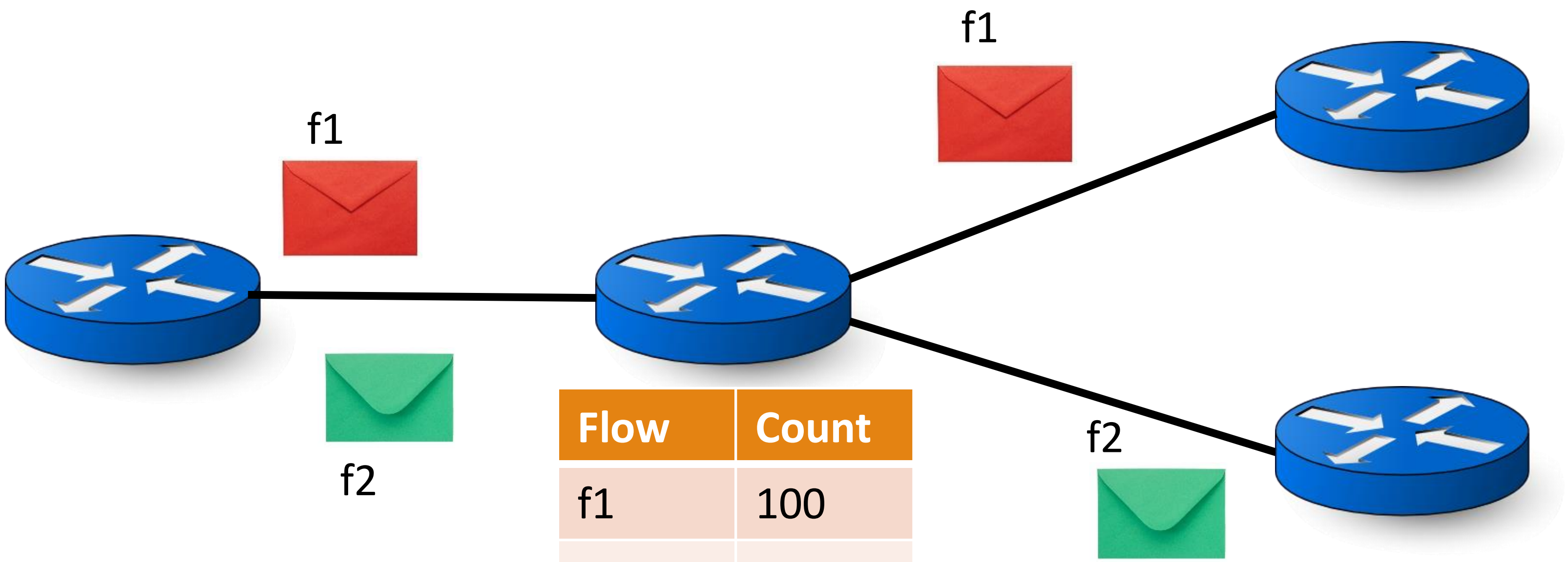


$k = 2$

Why detect heavy hitters?

Trouble-shooting and anomaly detection

Dynamic routing or scheduling of heavy flows



Flow	Count
f1	100
f2	75
f3	5

Problem Statement

Restrict processing to data plane

Low data plane state

High accuracy

Line-rate packet processing

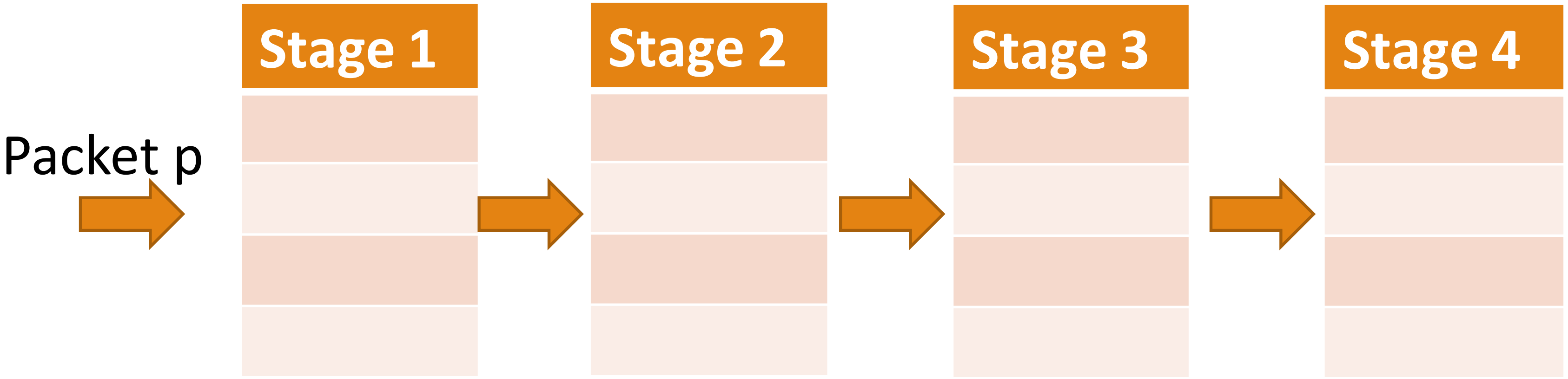
Emerging Programmable Switches

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

Technique	Pros	Cons
Sampling-based (Netflow, sflow, Sample & Hold)	Small “flow memory” to track heavy flows	Underestimates counts for heavy flows
Sketching-based (Count, Count-Min, Reversible)	Statistics for *all* flows in single data structure	No flow identifier to count association
Counting-based (***Space Saving***, Misra-Gries)	Summary structure with heavy flow ids and counters	Occasional updates to multiple counters

Motivation: Space-Saving Algorithm[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

[1]Metwally, Ahmed, Divyakant Agrawal, and Amr El Abbadi. "Efficient computation of frequent and top-k elements in data streams." *International Conference on Database Theory*. Springer Berlin Heidelberg, 2005.

Space Saving Algorithm

New
Key K6

Flow Id	Packet Count
K1	4
K2	2
K3	7
K4	10
K5	1

Flow Id	Packet Count
K1	4
K2	2
K3	7
K4	10
K6	***2***

High accuracy
Exactly one write

Entire table scan
Complex data structures

Towards HashPipe

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

Our Solution - HashPipe

Always insert new key in the first stage

Hash to index to a location

Carry evicted key to the next stage

New key K

h_1 (K) -> K1

Stage 1	
A	5
K1	4
B	6
C	10

Stage 2	
K2	3
D	15
E	25
F	100

Stage 3	
G	4
K3	3
H	10
I	9

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

Stage 1	
A	5
K	**1**
B	6
C	10

(K1, 4) →

Stage 2	
D	3
E	15
K2	25
F	100

Stage 3	
G	4
K3	3
H	10
I	9

HashPipe

At any table stage, retain the heavier hitter

h_2(K1) -> K2
Max(K1, K2) -> K2

Stage 1	
A	5
K	1
B	6
C	10

(K1, 4)

Stage 2	
D	3
E	15
K2	25
F	100

Stage 3	
G	4
K3	3
H	10
I	9

HashPipe

At any table stage, retain the heavier hitter

h_3(K1) -> K3
Max(K1, K3) -> K1

Stage 1	
A	5
K	1
B	6
C	10

Stage 2	
D	3
E	15
K2	25
F	100

(K1, 4)

Stage 3	
G	4
K3	3
H	10
I	9

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

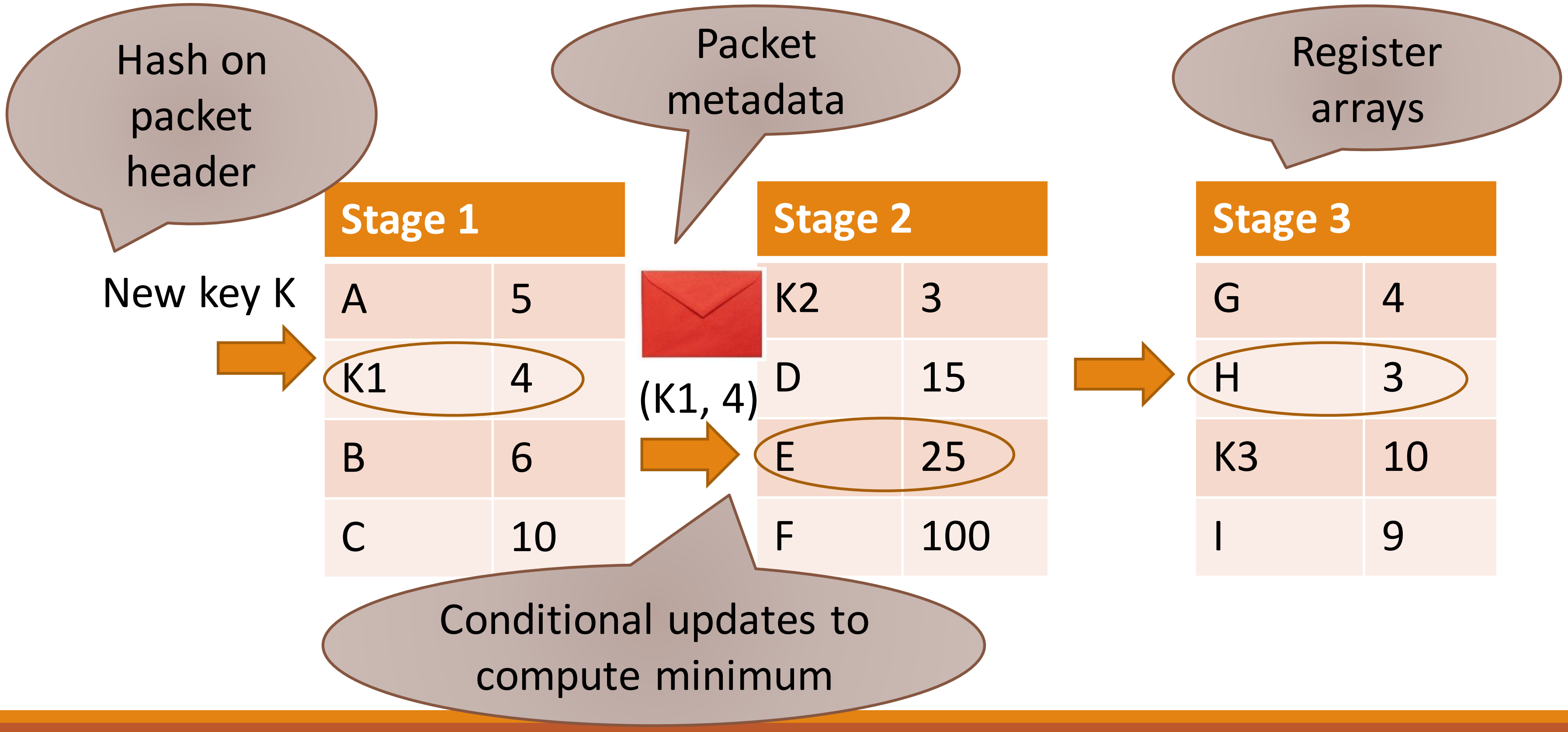
HashPipe Summary

Split hash table into *d* stages

Condition	Stage 1	Stages 2 - *d*
Empty	Insert with value 1	Insert key and value carried
Match	Increment value by 1	Coalesce value carried with value in table
Mismatch	Insert new key with value 1, evict and carry key in table	Keep key with higher value and carry the other

Implementation

Prototyped on P4



Evaluation Setup

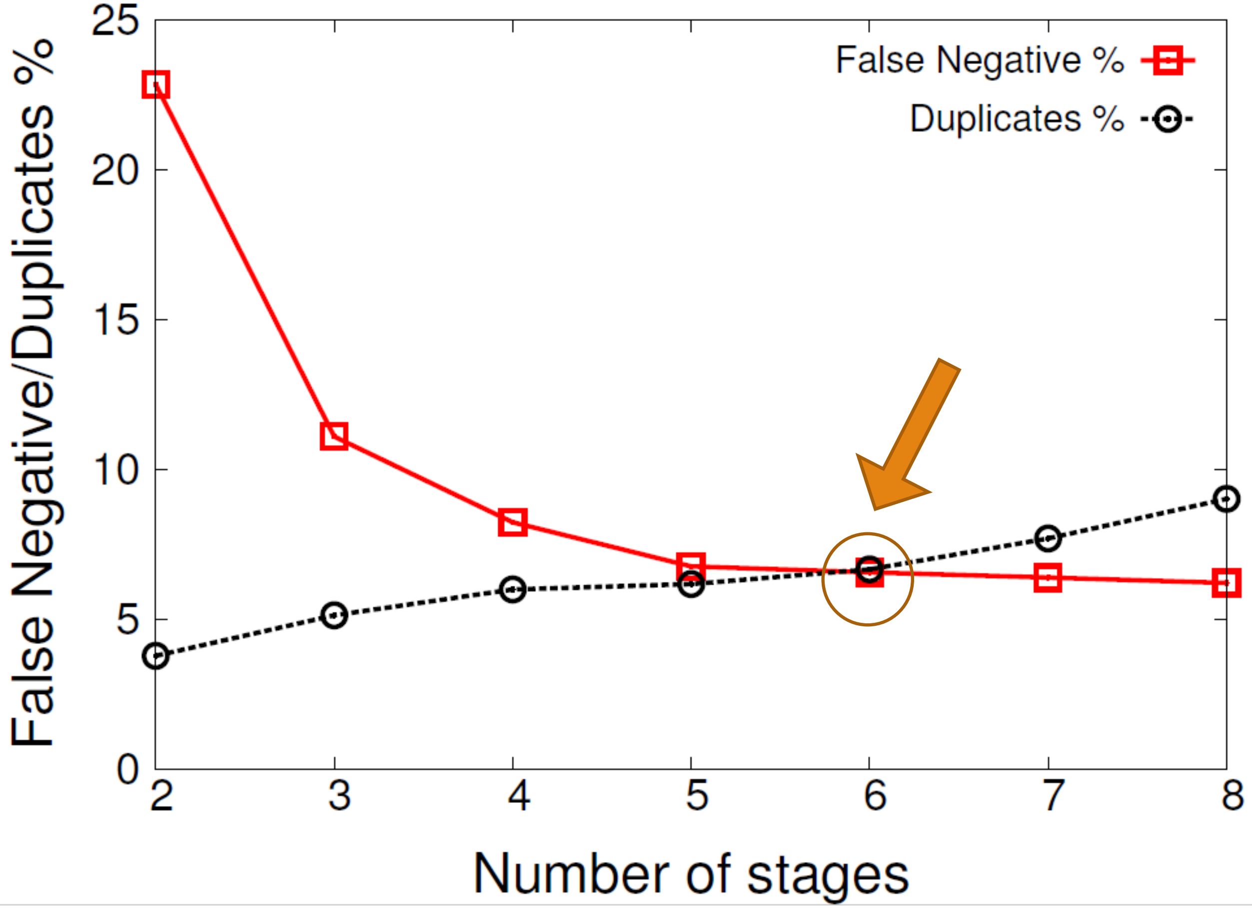
Top-*k* 5 tuples on CAIDA traffic traces with 500M packets

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

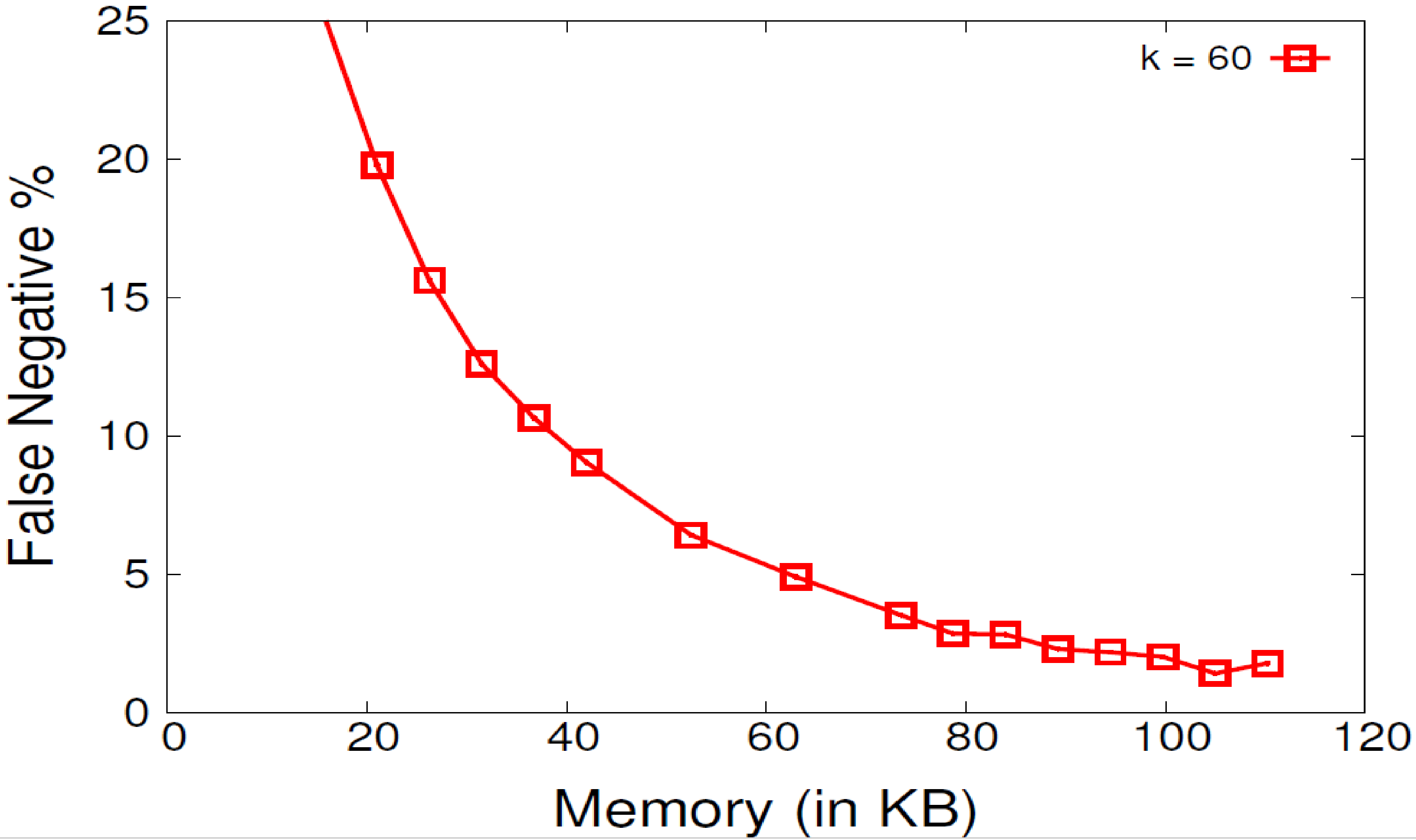


k = 210

5040 flowids maintained in table

HashPipe Accuracy

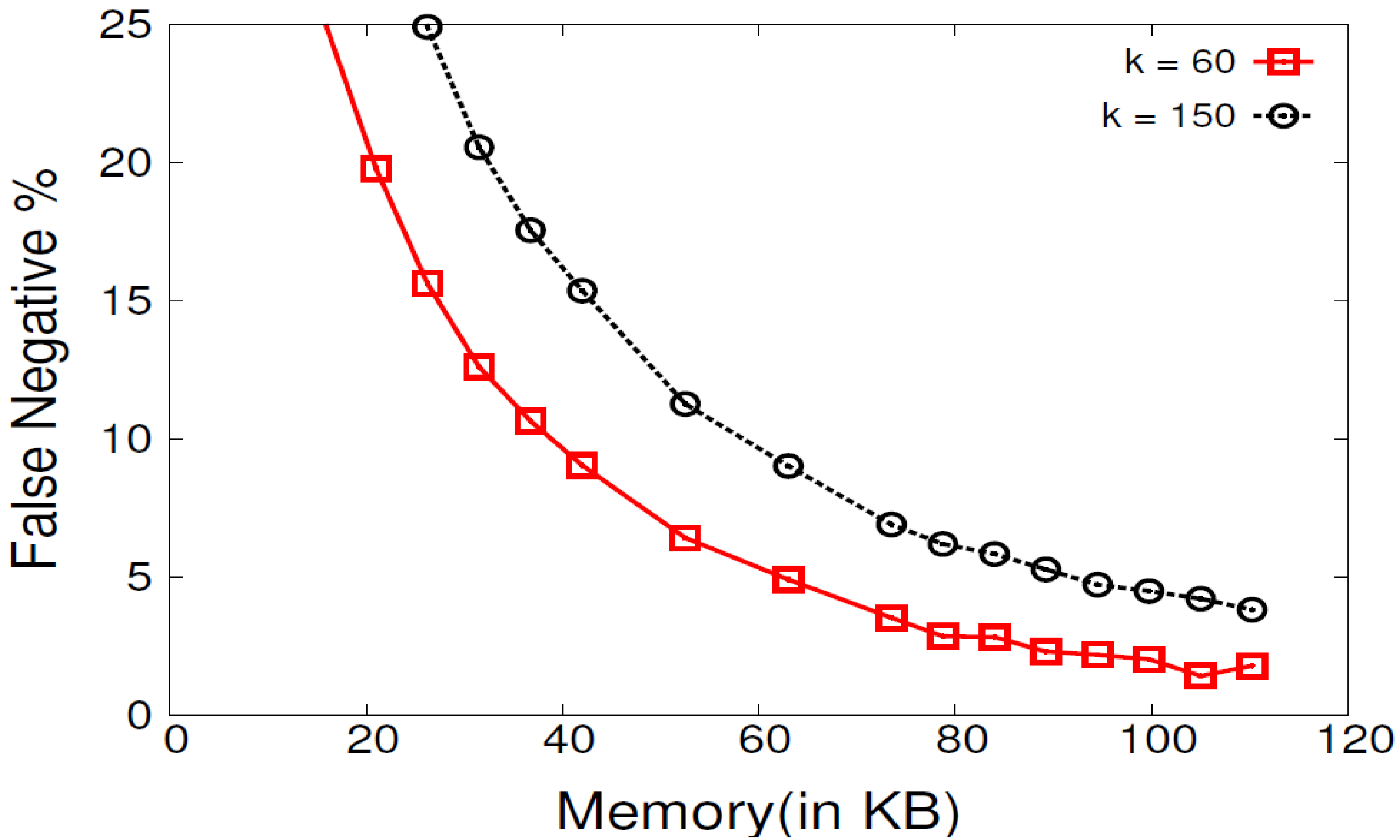
5-10% false negatives for detecting heavy hitters



HashPipe Accuracy

5-10% false negatives for the detecting heavy hitters

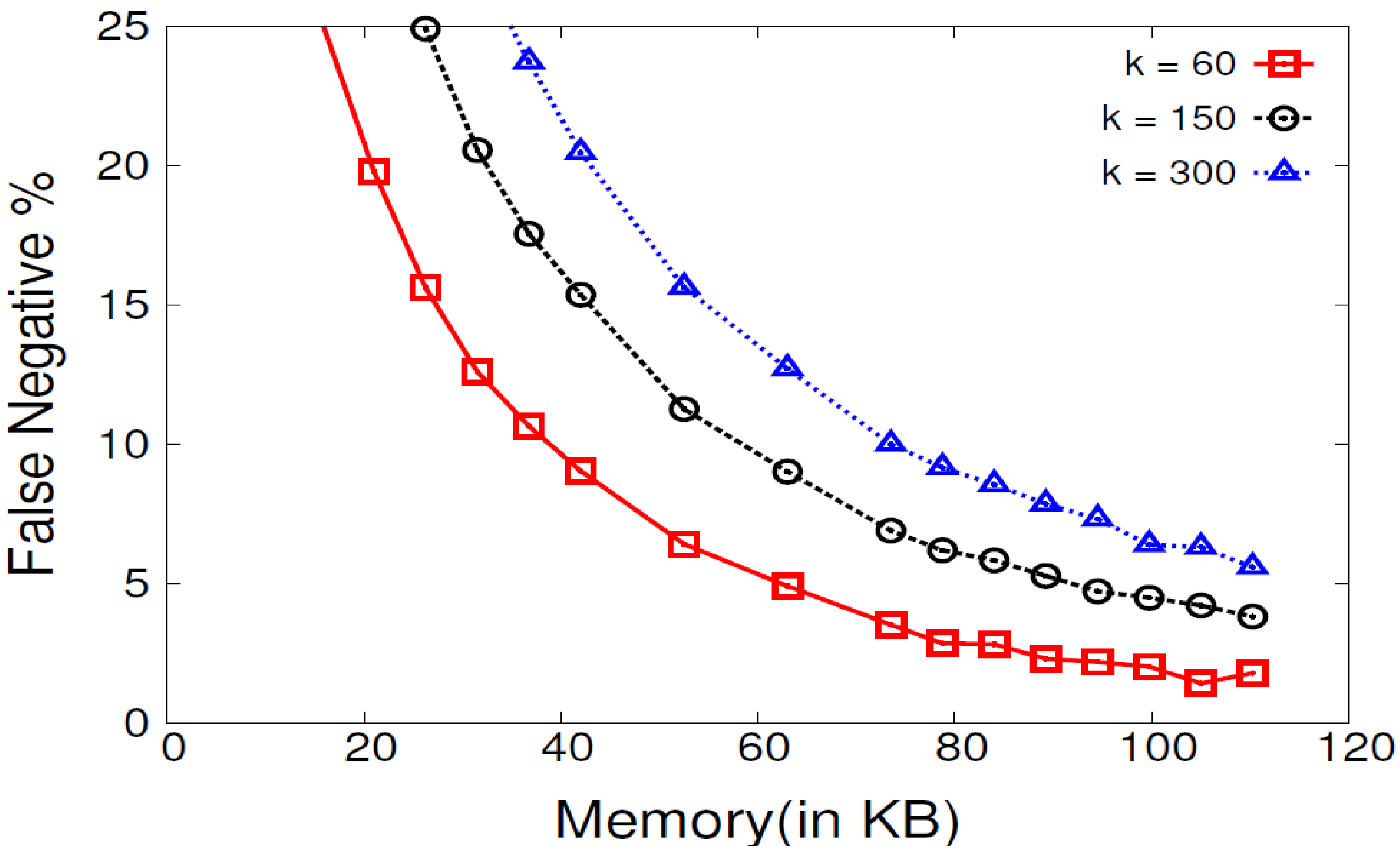
4500 flow counters on traces with 400,000 flows



HashPipe Accuracy

5-10% false negatives for the detecting heavy hitters

4500 flow counters on traces with 400,000 flows



Competing Schemes

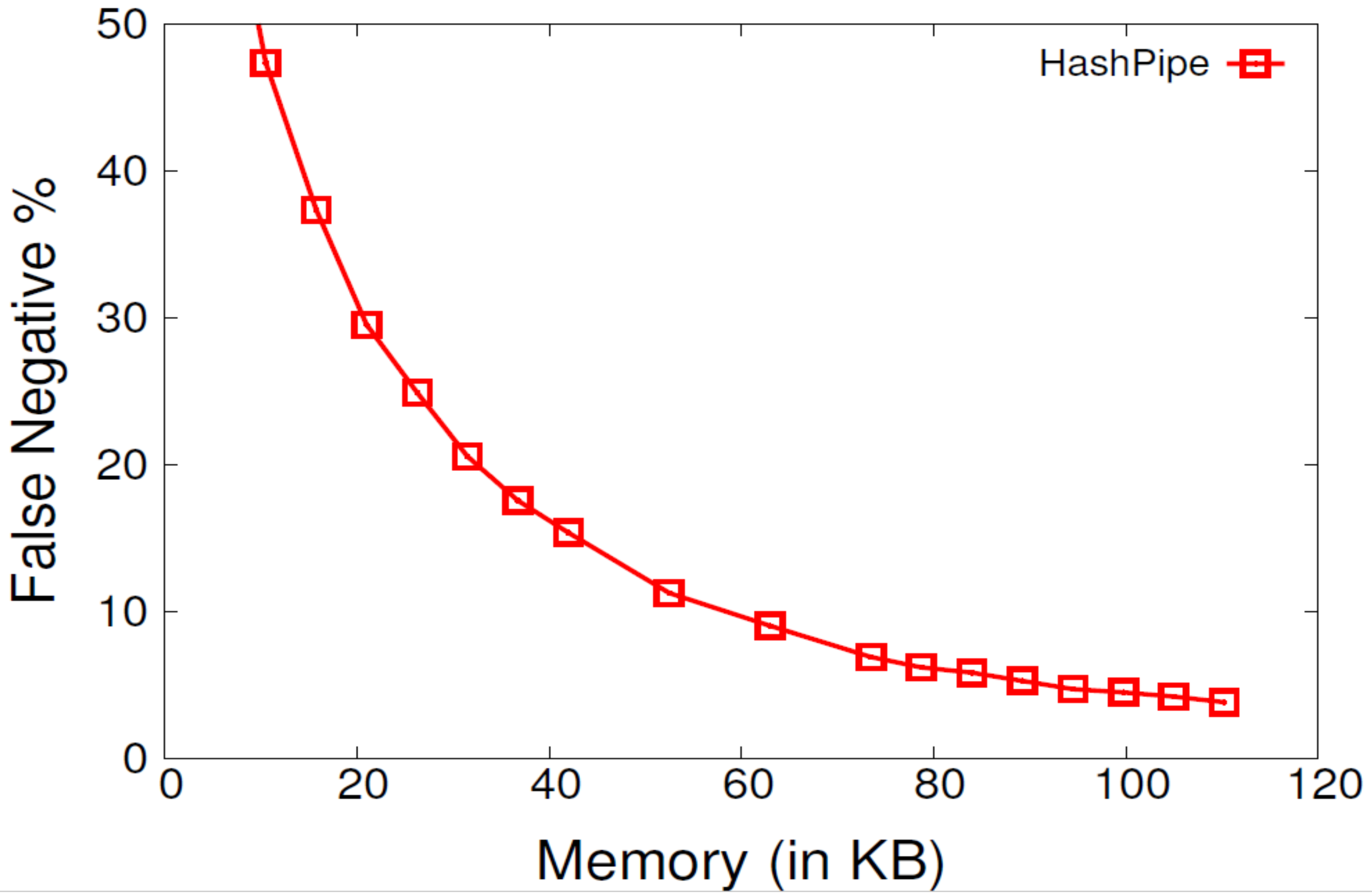
Sample and Hold

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

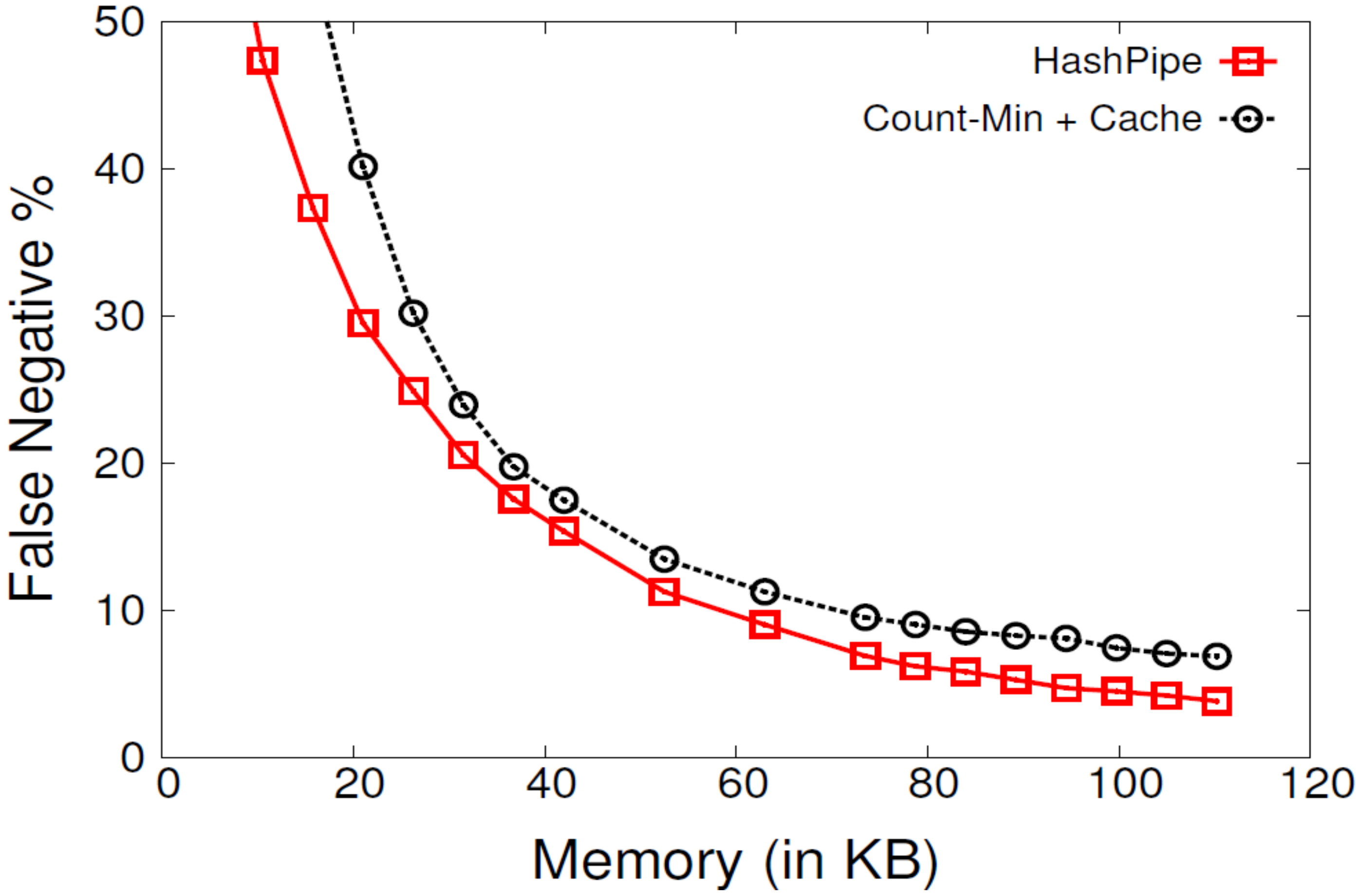
Count-Min Sketch

- Increment counters for every packet at *d* hashed locations
- Estimate using minimum among *d* location
- Track heavy hitters in cache

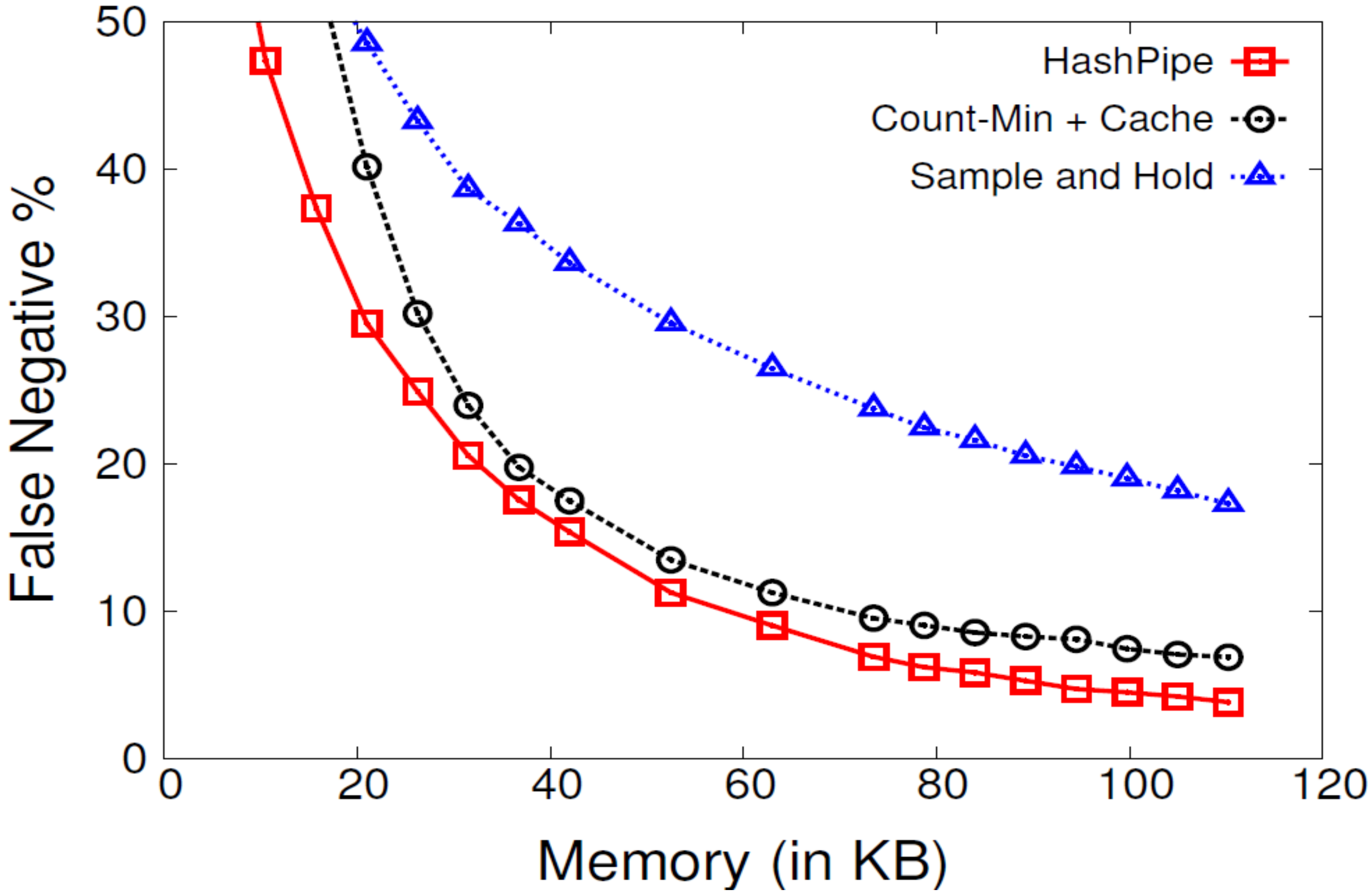
HashPipe vs. Existing Solutions



HashPipe vs Existing Solutions



HashPipe vs Existing 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

```
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;
  mDif = (mValid == 0) ? 0 : mKeyTable - mKeyCarried;

  // update hash table
  flowTracker[mIndex] = ipv4.srcAddr;
  packetCount[mIndex] = (mDif == 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;
}
```

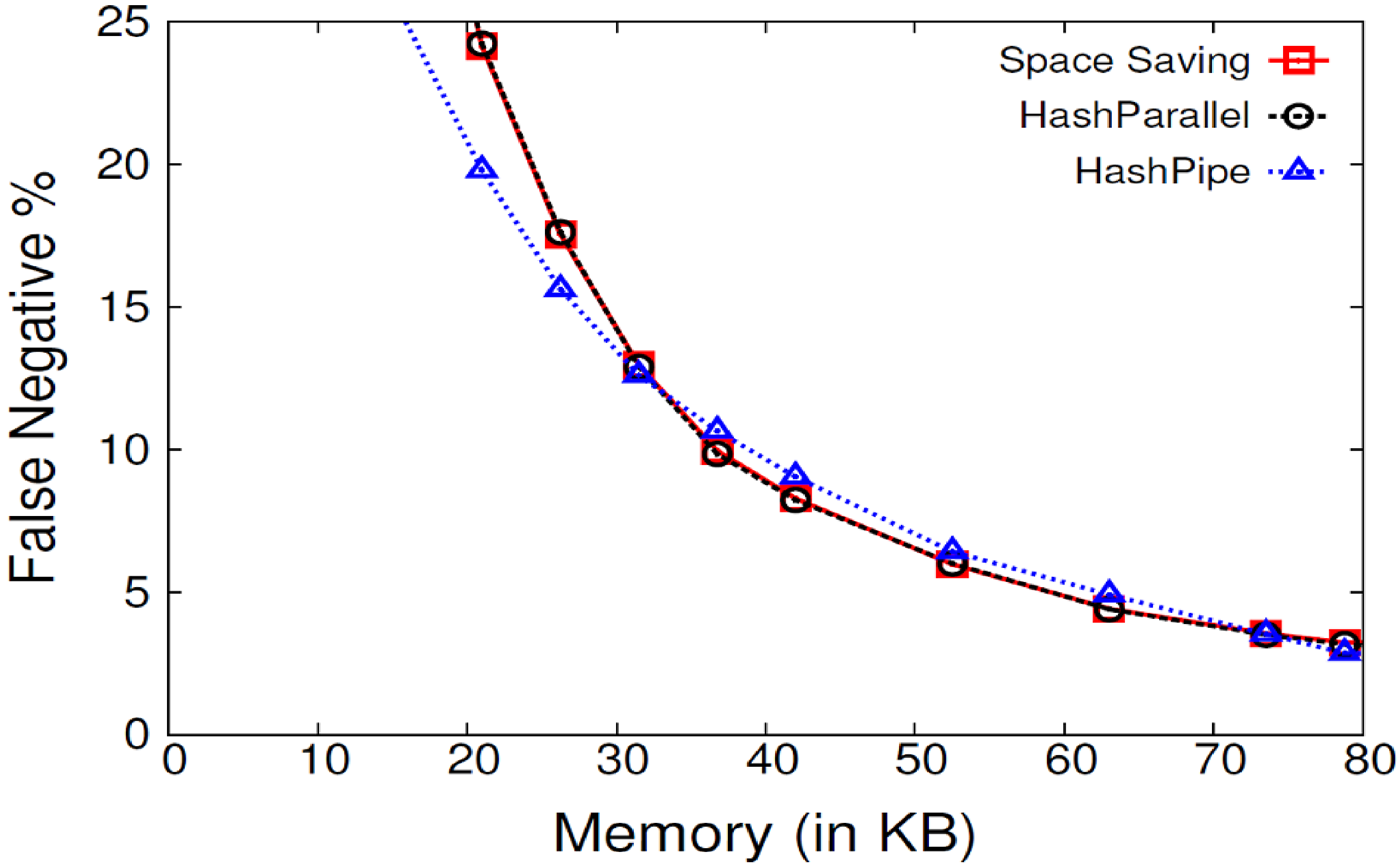
P4 prototype – Stage 2 onwards

```
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