Heavy Hitter Detection

Guest lecture for CS344

Jeongkeun Lee, Georgios Nikolaidis

Certain slides by courtesy of folks at Barefoot Networks

What are Heavy Hitters?

- **a.k.a elephant flows**
  - Major source of network congestion
  - Increase flow completion time for (delay-sensitive) mice flows
    - Bit-torrent can kill your Facetime
    - Hadoop traffic can penalize online banking transactions

- **Small number of flows**
  that constitute most of the network traffic
  over a certain amount of time (to differentiate from micro bursts)

- **Definite numbers depend on various factors**
  - Port speed, network RTT, traffic distribution, application policy, etc.
  - E.g., more than 50KB within the last 1 ms
    ➔ at least 400Mbits/s ➔ up to 250 concurrent heavy hitters @ 100G port

Why Detect Heavy Hitters?

- **To treat them differently**
  - Queuing: put them in low-priority queue while mice go to high-priority
  - Flow rate control: allocate fair 1/n bandwidth
  - Traffic engineering: route them through high-bandwidth paths
  - Visibility: know the culprits of a congestion

  - Or drop them!

How to Detect Heavy Hitters?

- **HHD is basically ‘rate’ estimation**
  - Number of bytes (or packets) *per unit time*

- **Can I install match-action rules to track per-flow rates?**
  - No, expensive: there could be millions of flows active in the network
  - And slow: heavy hitting can be transient, shorter than rule-insertion time from control plane

- **Detection needs to be *fast***
  - Immediate detection enables immediate reaction
  - Sub-millisecond detection & reaction is critical in datacenters w/ small RTT and shallow buffer
  - Hence, we want rate estimation, HHD, reaction all in H/W data-plane

Detection needs to be efficient to fit in H/W

- **On-chip SRAMs are fast, but small**
  - Need to find heavy hitters out of *millions* of flows
  - Storing 5tuple flow key is too expensive
- **ALU operations on high-speed H/W may be limited**
  - E.g., no floating-point arithmetic
  - Note: programmable data-plane design principle
    - Programmability should not sacrifice speed
    - Rule of thumb: process 1 packet per cycle

HHD approaches

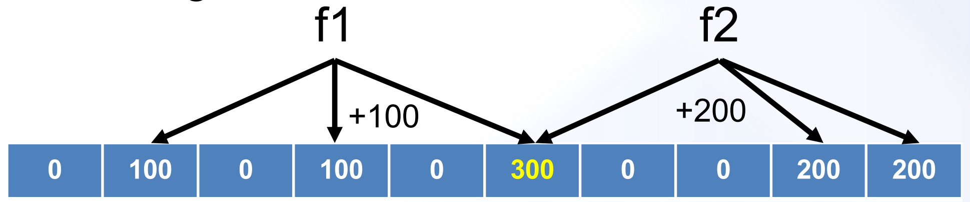
- **Top-N flow selection**
  - Detect N number of flows with highest rate
  - E.g., space saving algorithm and its variants (HashPipe @ SOSR’17)
  - Less sensitive to rate threshold
  - Algorithm complexity and memory requirements can be high
  - (Need separate rate estimation for the top-N flows)

- **Threshold-based**
  - Estimate rate and compare it to a threshold
  - Ways to estimate flow rates in data-plane
    - Time-decaying low-path filter (may need H/W support)
    - Count-min sketch: counting Bloom filter (data structure) + periodic reset

- **This lecture uses count-min sketch**

HHD via count-min sketch

- **Counting**
  - Each flow computes multiple hash indices using flow 5 tuple as hash key
    - hash-1(flow), hash-2(flow), ..., hash-k(flow)
  - Add packet size to the indexed locations of counter array
  - Flows can hash-collide, adding to a common counter instance



  - Take *minimum* of the counter values. count(f1) = min(100, 100, 300)

HHD via count-min sketch

- **Rate estimation**
  - Rate = min_value / measurement_time
- **Detection**
  - Compare the rate to a threshold
- **Periodic reset for timely detection**
  - Reset counter values and measurement time to zero

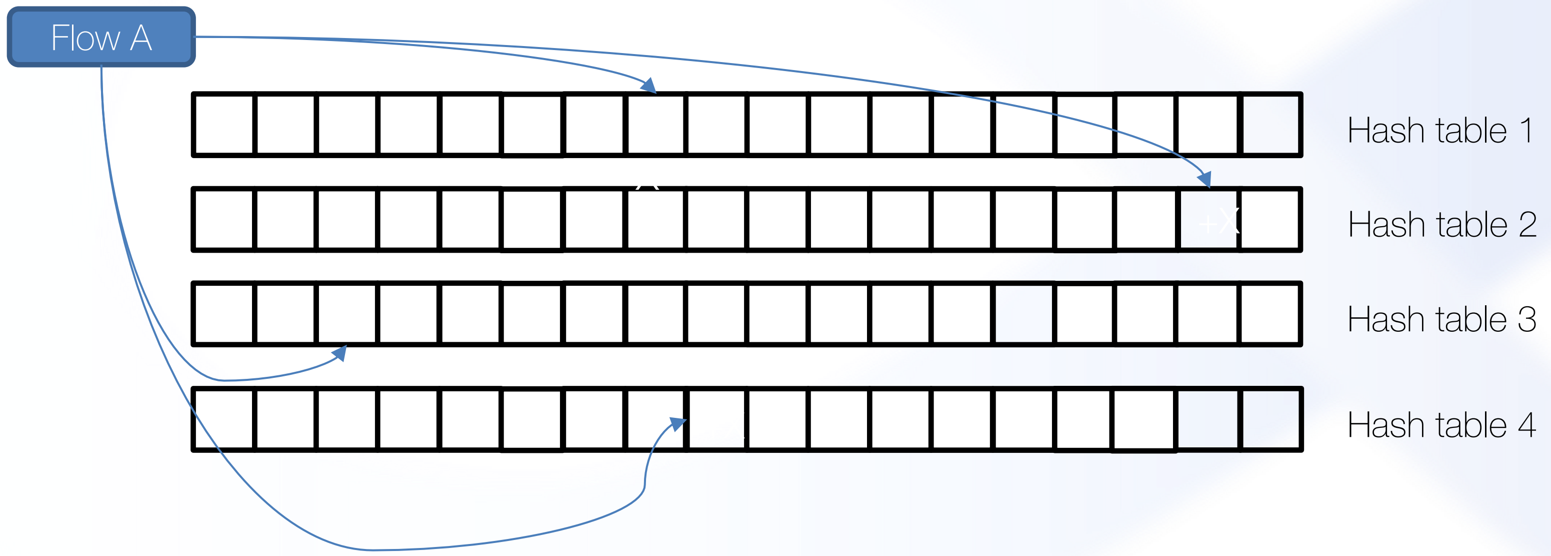
- We will detail design of each step, starting from counting

Counting: H/W-friendly counting bloom filter

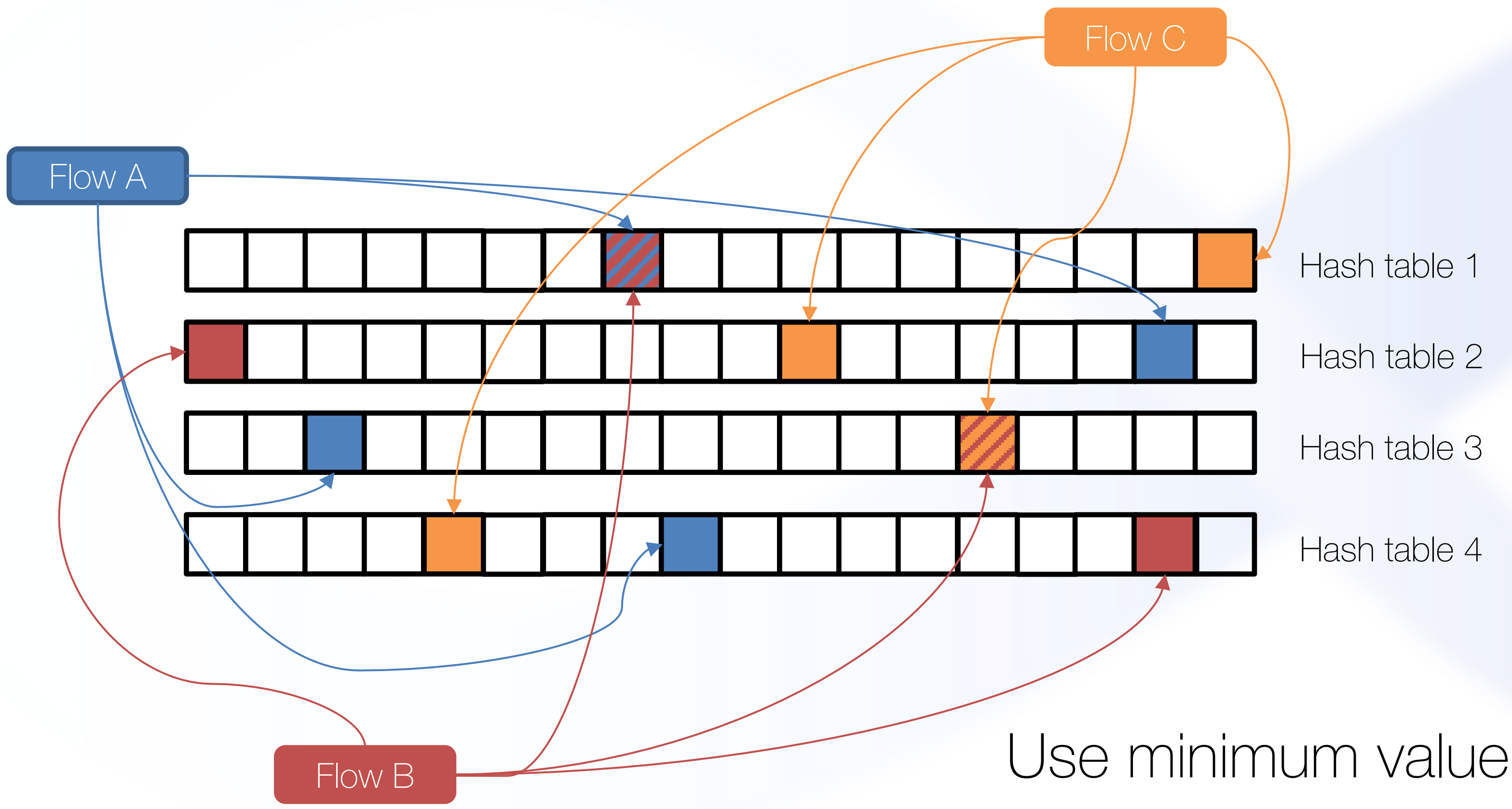
- **Using multiple hash functions is key for estimation accuracy**
  - If all hash locations collide with other flows ➔ over-estimation
  - (count-min sketch never underestimates)
  - Optimal # of hash functions can be larger than one

- **Multiple, concurrent access on a share memory can slow down speed**

- **Solution: create K counting arrays for K hash functions**
  - Each hash function accesses a dedicated array exclusively
  - Note: parallel bloom filter is known to yield comparable accuracy
    - “Implementing Signatures for Transactional Memory”, MICRO 2007.
  - Using parallel arrays has another benefit (later slide).



New packet: X bytes



Use minimum value

Rate estimation and thresholding

- Rate estimation and thresholding require floating point arithmetic (division or multiplication)

- If your target system doesn't support it

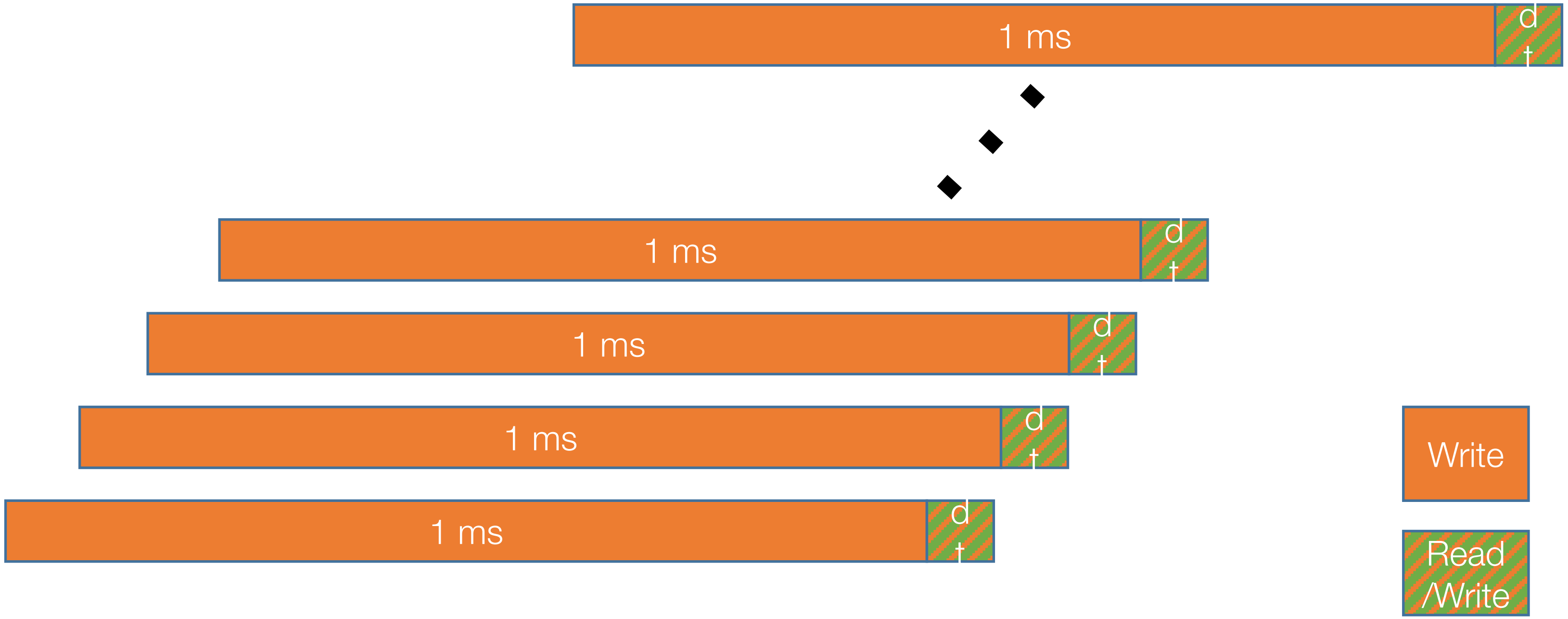
  you can memoize threshold values for different measurement time ranges

Periodic Reset

- **Use values to calculate the average rate of a flow since the last reset**
- **When the counter is too “young” it gives us observations over too short of a duration**
  - Too sensitive, cannot filter out micro-burst of mice flows
- **When the sketch is too “old”, identifying recent heavy hitters becomes hard (since we average over the duration since last reset)**

Idea: Stacked count-min sketches

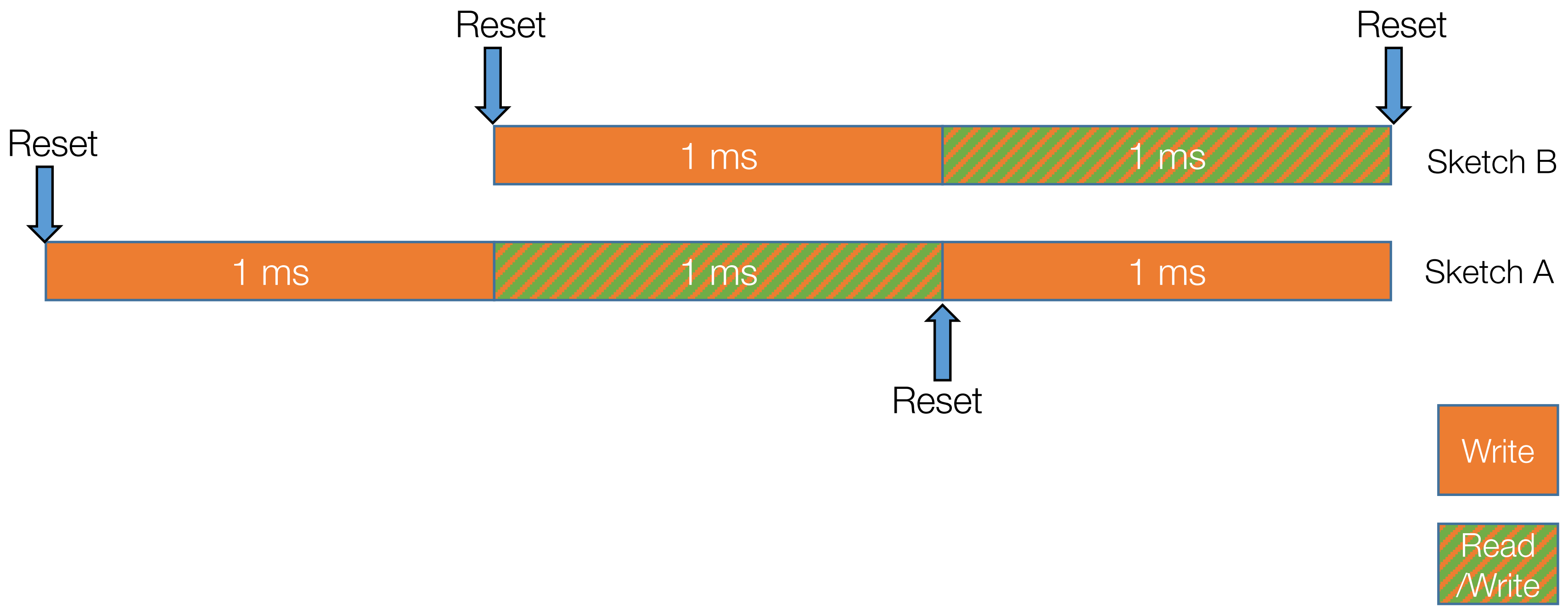
Each stack is a set of K parallel filters



Idea: Stacked count-min sketches

- Shorter dt leads to better accuracy
- Shorter dt also means more resources
- Compromise: let’s use two sketches

“Ping-pong” two count-min sketches



Ping-pong and reset must be synchronous

- **From control plane**
  - Switch sketch A to B and then reset A
  - Reset entire arrays can take time
  - Using K parallel arrays help
    - Reduce reset time by 1/K, maximize write time

- **H/W assisted approach**
  - Packet generation triggered by h/w clock
  - P4 program acts on the special h/w-generated pkt

Backups

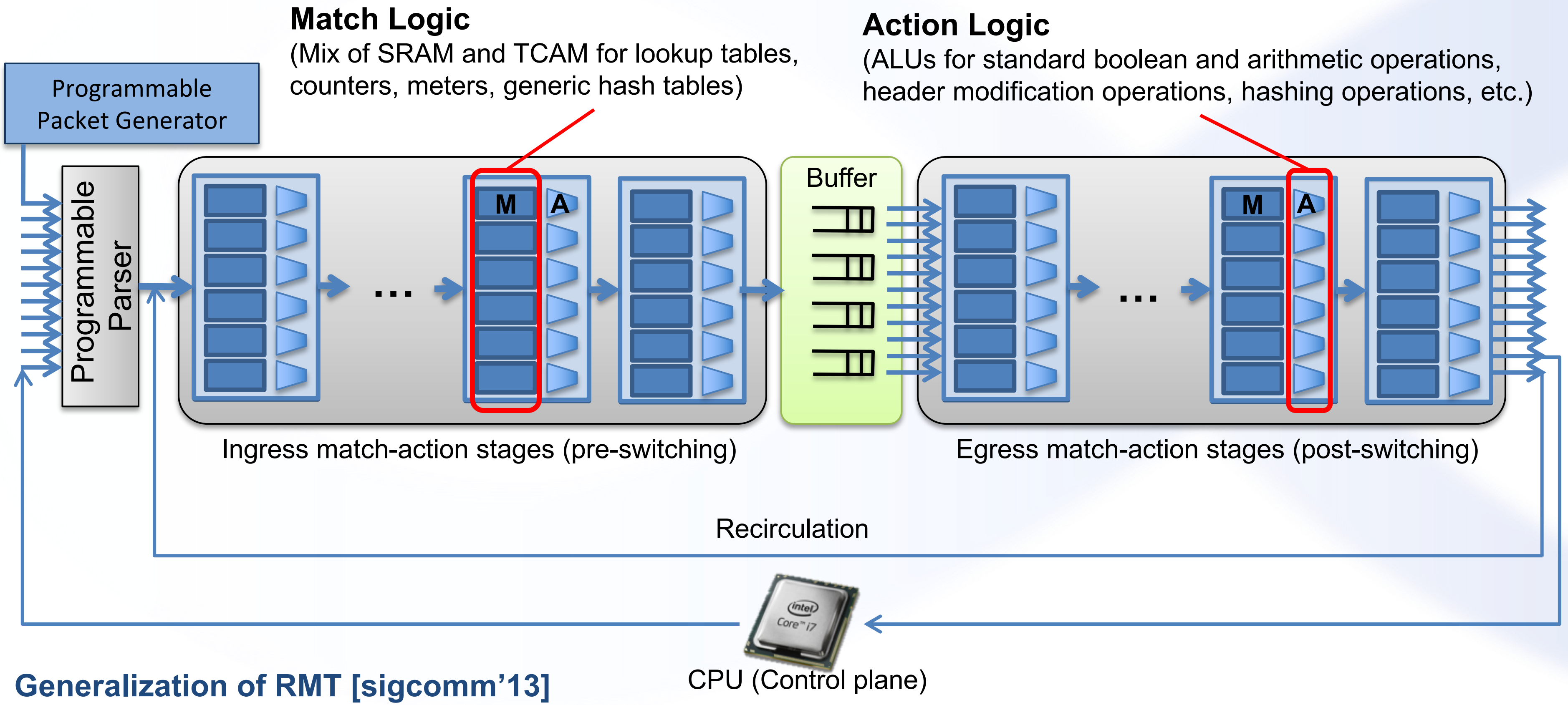
Non-networking usecase of HHD

- In-network caching (NetCache @ SOSP’17)
  - ToR switch works as a cache for a rack of memcached nodes
  - Using HHD, switch itself can detect ***hot*** key-value items for caching w/o relying on a separate controller
    - Hot item = a key with lots of *read* request

H/W-friendly count-min sketch

- **Optimal number of hash functions to minimize the number of overestimated flows**
  - $k = \frac{m}{n} \ln 2$
  - k : # of hash functions
  - m: array size
  - n: number of flows
- **If consider the size of overestimations**
  - "Summarizing and Mining Skewed Data Streams" https://www.cs.rutgers.edu/~muthu/cmz-sdm.pdf

PISA: Protocol Independent Switch Architecture



Match Logic
(Mix of SRAM and TCAM for lookup tables, counters, meters, generic hash tables)
Action Logic
(ALUs for standard boolean and arithmetic operations, header modification operations, hashing operations, etc.)
Programmable Packet Generator
Programmable Parser
M
A
...
Buffer
Ingress match-action stages (pre-switching)
Egress match-action stages (post-switching)
Recirculation
CPU (Control plane)

Generalization of RMT [sigcomm'13]

What we have seen so far: Adding new networking features

1. New encapsulations and tunnels
2. New ways to tag packets for special treatment
3. New approaches to congestion control
4. New ways to manipulate and forward packets: e.g. splitting ticker symbols for high-frequency trading
5. News ways to speed-up memory cluster

What we have seen so far: World’s fastest middle boxes

1. Layer-4 load connection balancing at Tb/s
   – Replace 100s of servers or 10s of dedicated appliances with one PISA switch
   – Track and maintain mappings for 5 ~ 10 million HTTP connections

2. Stateless firewall or DDoS detector
   – Add/delete and track 100s of thousands of new connections per second
   – Include other stateless line-rate functions (e.g., TCP SYN authentication, sketches, or Bloomfilter-based whitelisting)

What we have seen so far: Offloading part of computing to network

1. DNS cache
2. Key-value cache [ACM SOSP’17]
3. Chain replication
4. Paxos [ACM CCR’16] and RAFT consensus protocols
5. Parameter service for DNN training

Which app P4 will succeed (or not)?

- Top question at P4D2’17 Fall Panel session
- Data being packetized and forwarded between distributed system components
- Apps that can be implemented without (much) sacrificing speed, power, cost
  - Tradeoff point can change based on deployment model: in-switch vs. dedicated appliance vs. smartNIC

PISA/P4 design choices

- No loop in match-action stages
  - (Loop with a max limit is allowed in parser)
  - HW and language both modeled as a DAG
  - Guarantee bandwidth and latency
- Tables are read-only in data-plane
  - Table update is computationally expensive
  - Control plane only can add/delete/update table entries

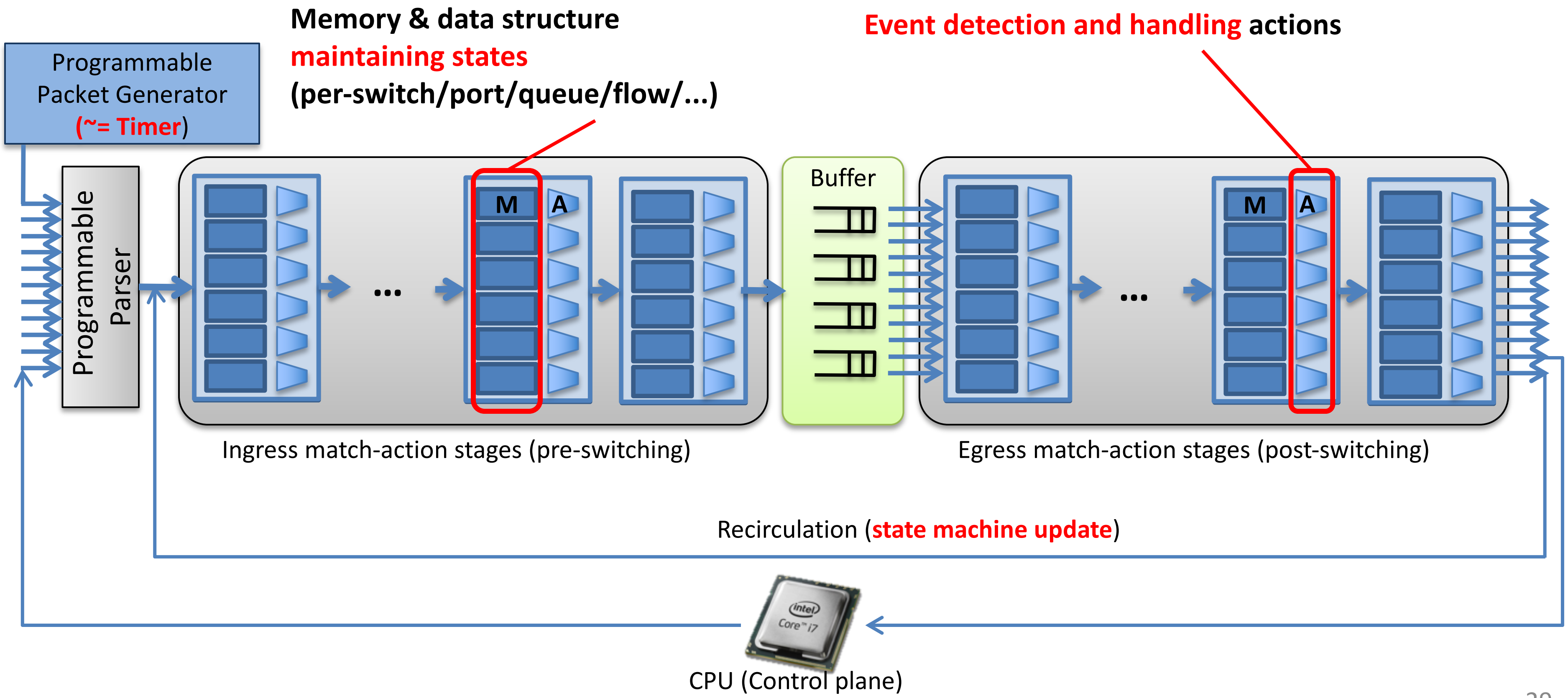
App design challenges

- “Learning”
  - Learning (table write/update) is done by CPU
    - E.g., L2 MAC learning, connection learning
  - CPU-HW speed mismatch may break consistency
  - App-specific solution may exist (e.g., Bloomfilter in L4 connection LB)
  - Some device may support h/w learning
- State is stage local
  - May need recirculation and/or replication for fast state propagation
  - Consistency model and techniques from distributed systems can help

PISA: An architecture for high-speed programmable ~~packet forwarding~~ event processing

- I/O events
- Timer events
- State events

PISA as a event processing engine



Memory & data structure
maintaining states
(per-switch/port/queue/flow/...)
Event detection and handling actions
Programmable Packet Generator (~= Timer)
Programmable Parser
M
A
...
Buffer
Ingress match-action stages (pre-switching)
Egress match-action stages (post-switching)
Recirculation (state machine update)
intel Core i7
CPU (Control plane)

Observations

- PISA and P4: The first attempt to define a machine architecture and programming models for networking in a disciplined way
- Network is becoming yet another programmable platform
- E.g., “network storage”
  - Yesterday, it meant storage system **connected** by network
  - Tomorrow, storage system gets **faster and more reliable** by the network