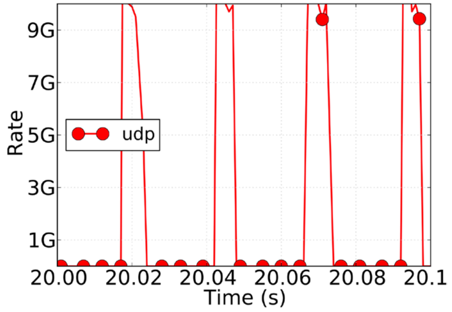
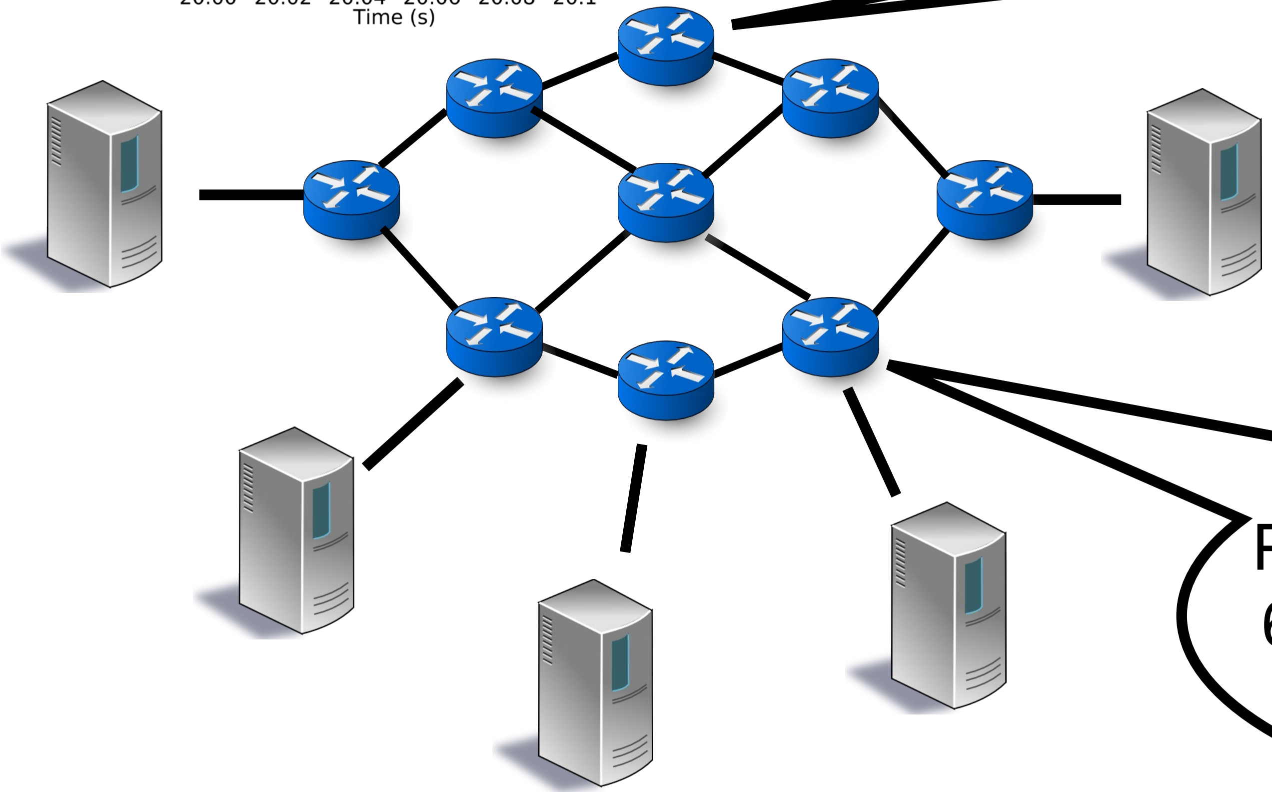
Language-Directed Hardware Design for Network Performance Monitoring

Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimal Jeyakumar, and Changhoon Kim

Example: Who caused a microburst?





Queue build-up deep in the network

- End-to-end probes
- Sampling
- Counters & Sketches
- Mirror packets

Per-pkt info: challenging in software
6.4Tbit/s switch: Need 100M recs/s
COTS: 100K-1M recs/s/core

Switches should be first-class citizens in performance monitoring.

Why monitor from switches?

- Already see the queues & concurrent connections
- Infeasible to stream all the data out for external processing
- Can we filter and aggregate performance on switches directly?

We want to build “future-proof” hardware:

Language-directed hardware design

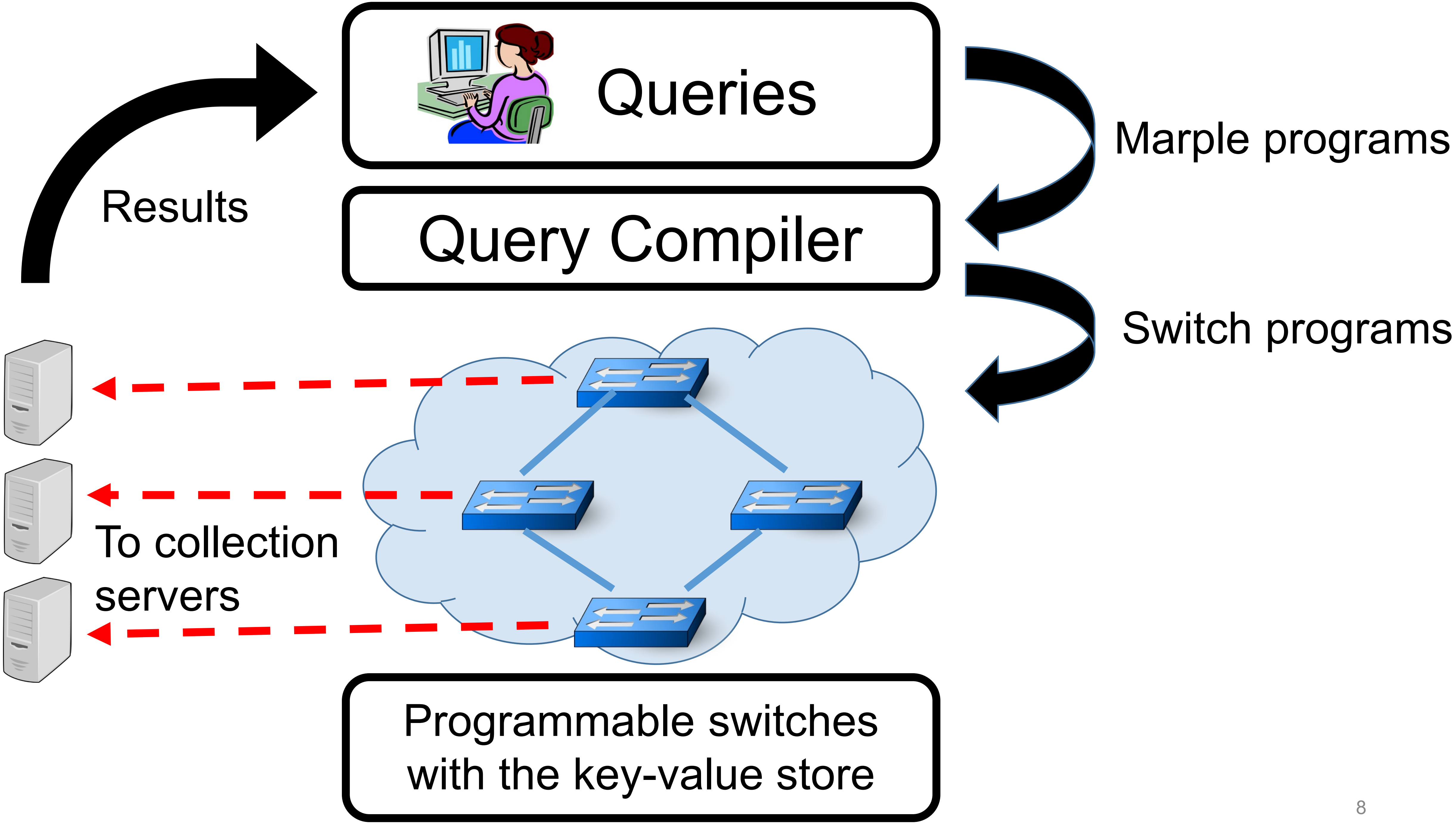
Performance monitoring use cases

Expressive query language

Line-rate switch hardware primitives

Contributions

- Marple, a performance query language
- Line-rate switch hardware design
  - Aggregation: Programmable key-value store
- Query compiler

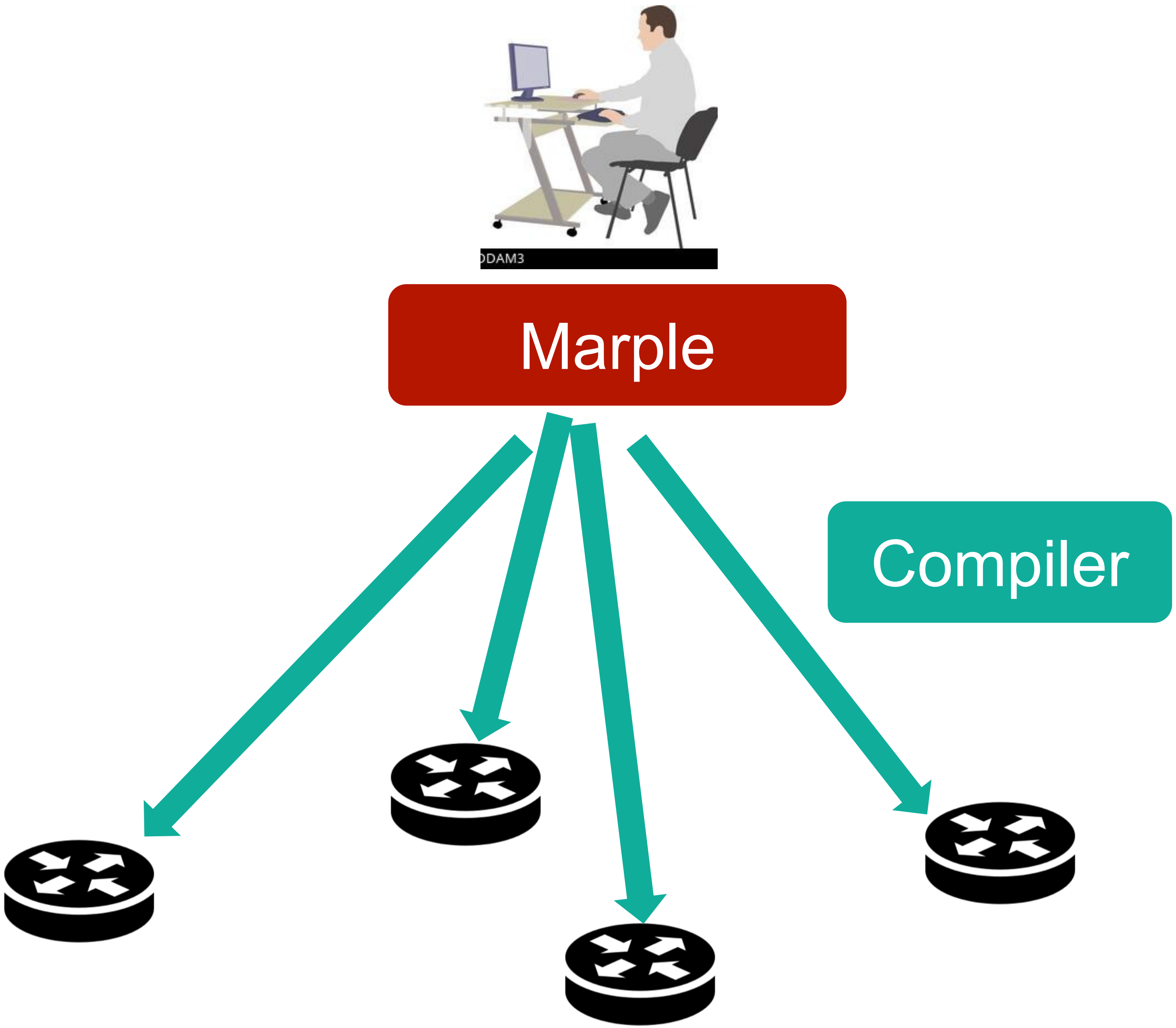

Queries
Results
Marple programs
Query Compiler
Switch programs
To collection servers
Programmable switches with the key-value store

Marple: Performance Query Language

To express telemetry queries

A high-level language
e.g. Marple

Hardware supports the language


Marple
Compiler

pktstream

Stream: For *each* packet at *each* queue,

$$S := (\underbrace{\texttt{switch}, \texttt{qid}}_{\text{Location}}, \underbrace{\texttt{hdrs}, \texttt{uid}}_{\text{Packet identification}}, \underbrace{\texttt{tin}, \texttt{tout}}_{\text{Queue entry and exit timestamps}}, \underbrace{\texttt{qsize}}_{\text{Queue depth seen by packet}})$$

pktstream

Stream: For *each* packet at *each* queue,

`S:= (switch, qid, hdrs, uid, tin, tout, qsize)`

Familiar functional operators

filter | map | zip | groupby

Filter: restrict packet data of interest

```
result = filter(pktstream, qid == Q and switch == S and
tout – tin > 1 ms)
```

Tracking packets that experience high queueing latencies (>1ms) at switch S, queue Q

Filter: restrict packet data of interest

```
result = filter(pktstream, qid == Q and switch == S and
                tout – tin > 1 ms)
```

```
R_output = filter(R_input, predicate)
```

Map: compute stateless functions over packets

```
result = map(pktstream, [tin/epoch_size], [epoch])
```

Rounding packet timestamps to an “epoch”

Map: compute stateless functions over packets

```
result = map(pktstream, [tin/epoch_size], [epoch])

R_output = map(R_input, [expression], [field])
```

Groupby: aggregate *statefully* over multiple packets

```
result = groupby(pktstream, [5tuple], count)

def count([num],[]):
    num = num+1
```

Counting packets belonging to each transport-level flow (i.e. 5-tuple)

Groupby: aggregate *statefully* over multiple packets

```
result = groupby(pktstream, [5tuple, switch], ewma)

def ewma([avg], [tin, tout]):
    avg = ((1-alpha)*avg) + (alpha*(tout-tin))
```

Maintaining an exponentially weighted moving average (EWMA) of queueing latencies

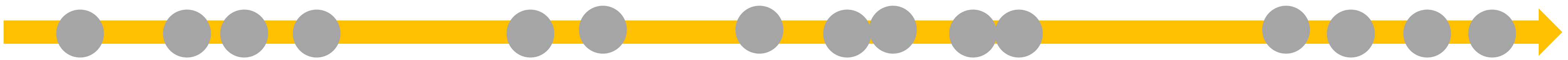
Tracking latency spikes for each connection

Groupby: aggregate *statefully* over multiple packets

```
result = groupby(pktstream, [5tuple], count)

result = groupby(pktstream, [5tuple, switch], ewma)


R_output= groupby(R_input, [aggFields], fun)
```

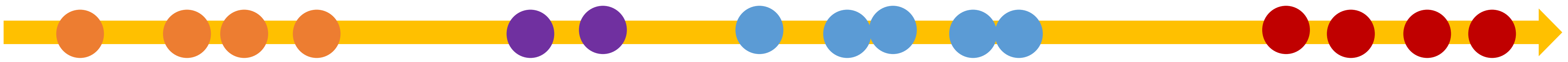
Emit()

Tracking the size distribution of flowlets



Emit()

Tracking the size distribution of flowlets



Emit()

Tracking the size distribution of flowlets

```
fl_track = groupby(pktstream, [5tuple], fl_detect);

def fl_detect([last_time, size], [tin]):
    if (tin - last_time > delta):
        emit() #stream out [last_time, size]
        size = 1
    else:
        size = size + 1
        last_time = tin
```

Chaining together multiple queries

- All Marple constructs have streams as their inputs and outputs
- Write queries that take results of previous queries as inputs

Chaining together multiple queries

Tracking the size distribution of flowlets

```
fl_track = groupby(pktstream, [5tuple], fl_detect);
fl_bkts  = map(fl_track, [size/16], [bucket]);
fl_hist  = groupby(fl_bkts, [bucket], count);
```

Chaining together multiple queries

```
fl_track = groupby(pktstream, [5tuple], fl_detect);
fl_bkts  = map(fl_track, [size/16], [bucket]);
fl_hist  = groupby(fl_bkts, [bucket], count);
```

src_addr	dst_addr	src_port	dst_port	protocol id	tin	...
1	2	0	0	0	2	...
5	1	1	2	0	0	...
1	2	0	0	0	3	...
...	...	...	...	...	...	...

Chaining together multiple queries

```
fl_track = groupby(pktstream, [5tuple], fl_detect);
fl_bkts  = map(fl_track, [size/16], [bucket]);
fl_hist  = groupby(fl_bkts, [bucket], count);
```

src_addr	dst_addr	src_port	dst_port	protocol id	last_time	size
1	2	0	0	0	10	11
5	1	1	2	0	22	30
…	…	…	…	…		…

Chaining together multiple queries

```
fl_track = groupby(pktstream, [5tuple], fl_detect);
fl_bkts  = map(fl_track, [size/16], [bucket]);
fl_hist  = groupby(fl_bkts, [bucket], count);
```

src_addr	dst_addr	src_port	dst_port	protocol id	last_time	size	bucket
1	2	0	0	0	10	11	0
5	1	1	2	0	22	30	1

Discussion: given fl_bkts, how should fl_hist look like?

```
fl_track = groupby(pktstream, [5tuple], fl_detect);
fl_bkts  = map(fl_track, [size/16], [bucket]);
fl_hist  = groupby(fl_bkts, [bucket], count);
```

src_addr	dst_addr	src_port	dst_port	protocol id	last_time	size	bucket
1	2	0	0	0	10	11	0
5	1	1	2	0	22	30	1
1	3	0	4	0	10	6	0
3	2	1	3	0	26	26	1
1	1	2	8	0	10	13	1
2	5	1	1	1	35	41	2

Chaining together multiple queries

```
fl_track = groupby(pktstream, [5tuple], fl_detect);
fl_bkts  = map(fl_track, [size/16], [bucket]);
fl_hist  = groupby(fl_bkts, [bucket], count);
```

bucket	count
0	2
1	3
2	1

Zip: join results across queries

Example: detecting TCP incast

TCP incast: fan-in of packets from many connections into a single queue

1. The number of active flows in a queue over a short interval of time is high
2. The queue occupancy is large

Zip: join results across queries

Example: detecting TCP incast

1. Compute the number of active flows over the current epoch

```
R1 = map(pktstream, [tin/epoch_size], [epoch]);
R2 = groupby(R1, [5tuple, epoch], new_flow);
R3 = groupby(R2, [epoch], count);
```

2. Combine with the queue occupancy information in the original pktstream

```
R4 = zip(R3, pktstream);
result = filter(R4, qsize > 100 and count > 25);
```

What Marple cannot do

Example

- EWMA over some packet field across all packets seen anywhere in the entire network, while processing packets in the order of their tout values.
- Challenges:
  - Coordinate between switches
  - OR stream all packets to a central location.

What Marple cannot do

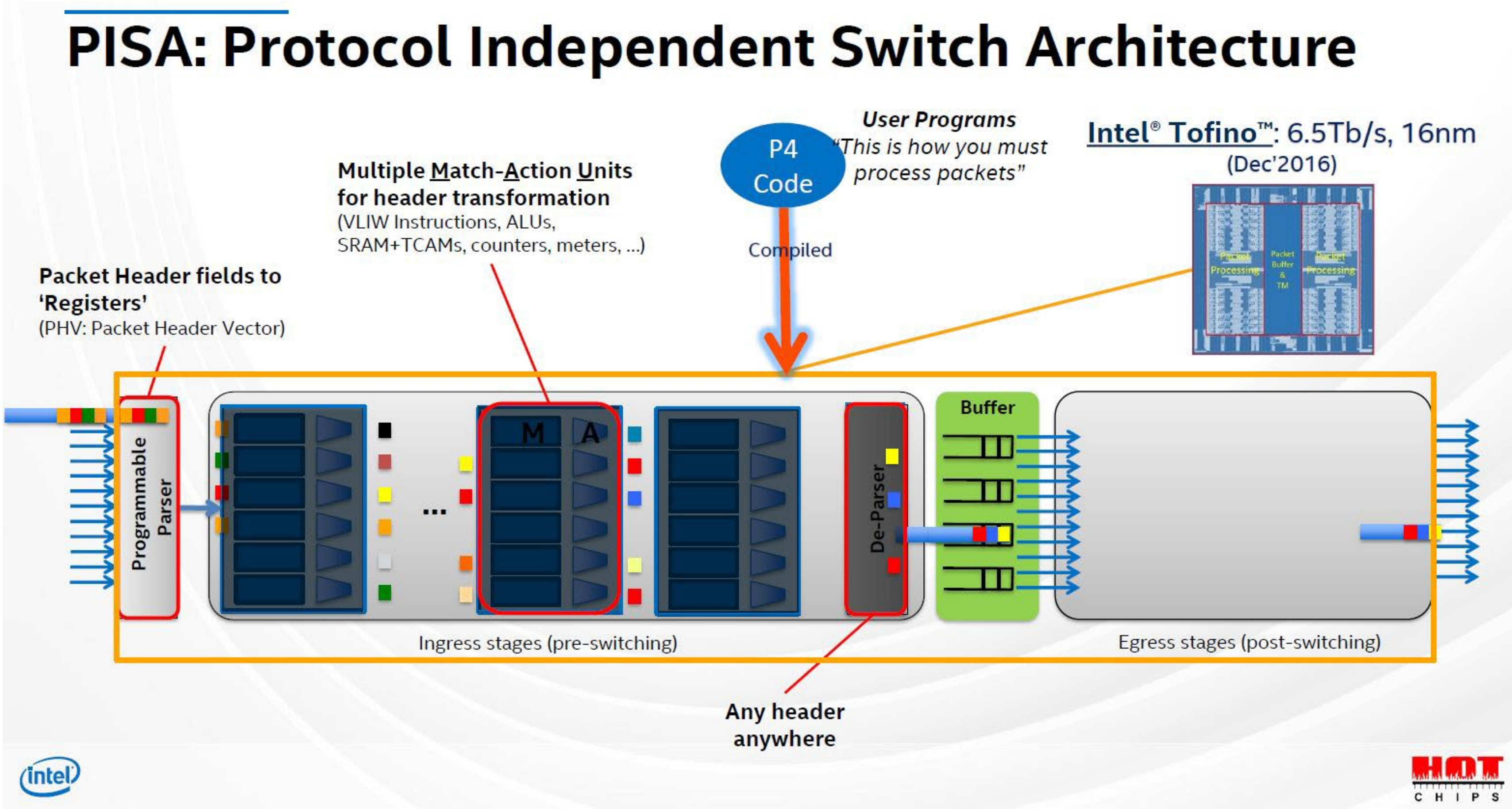
- Aggregations that need to process *multiple packets* at *multiple switches* in *order of their* tout *values*.

What Marple can do

1. Operate independently on each switch
2. Operate independently on each packet
3. Associative and commutative

Hardware Implementation

Architecture of a PISA switch



The Banzai Machine

```
if (counter < 99)
  counter++;
else
  counter = 0;
```

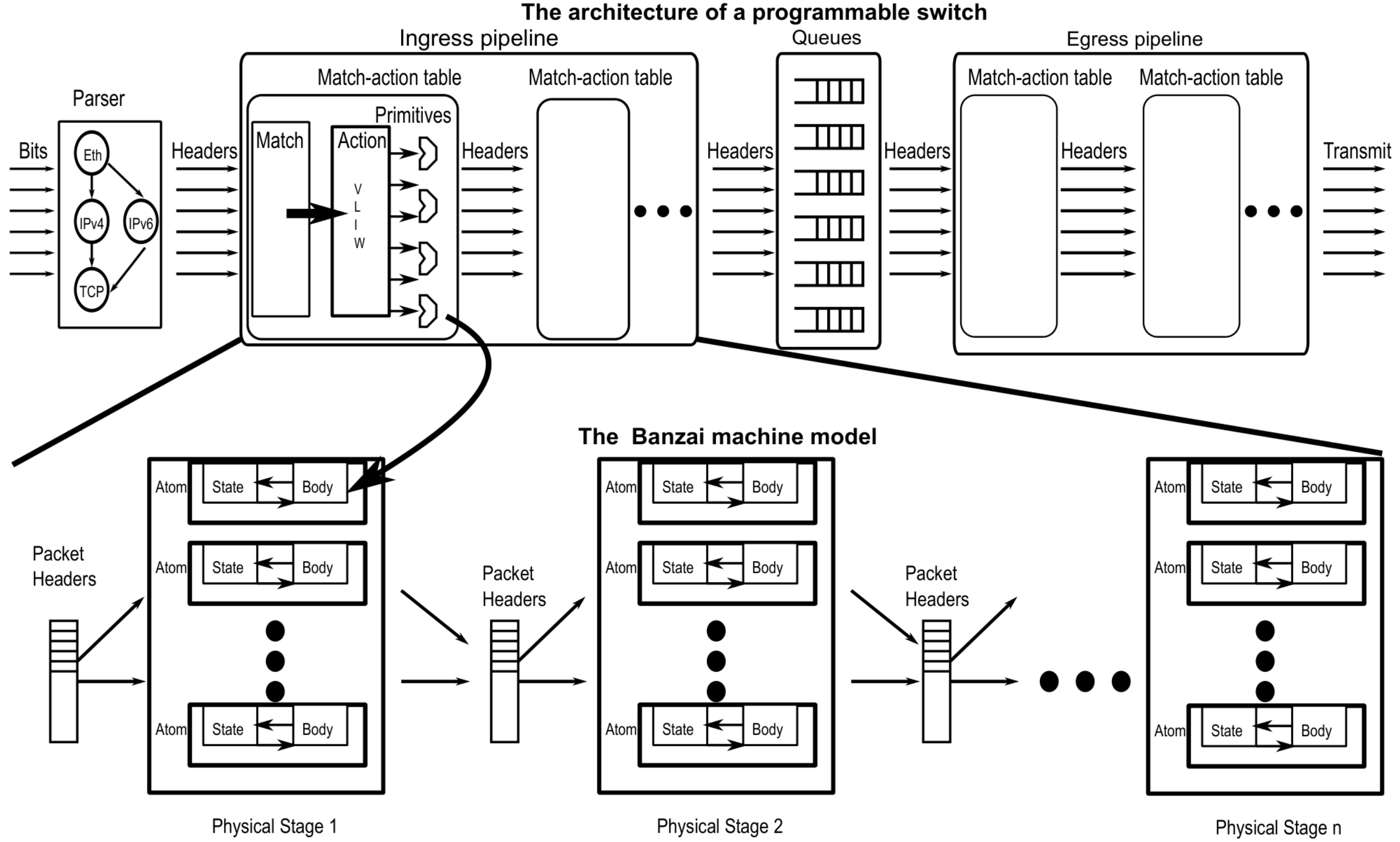
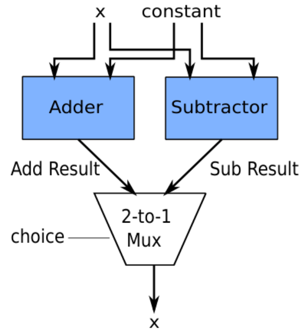


Figure 1: The Banzai machine model and its relationship to programmable switch architectures.



(a) Circuit for an atom that can add or subtract a constant from a state variable.

```
bit choice = ??;
int constant = ??;
if (choice) {
  x = x + constant;
} else {
  x = x - constant;
}
```

(b) Circuit representation as an atom template.

Figure 2: Atoms and atom templates

In practice, atom templates will be designed by an ASIC engineer and exposed as a machine's instruction set

Implementing Marple on switches

`S:= (switch, hdrs, uid, qid, tin, tout, qsize)`

Switch telemetry
[INT SOSR'15]

filter | map | zip | groupby

Stateless match-action rules
[RMT SIGCOMM'13]

Implementing Marple on switches

`S:= (switch, hdrs, uid, qid, tin, tout, qsize)`

Switch telemetry
[INT SOSR'15]

filter | map | zip | groupby

Stateless match-action rules
[RMT SIGCOMM'13]

The GROUPBY problem

- GROUPBY is the only language primitive that required a state to be stored. It wants a Key-Value store.

- Each stage in PISA contained only a few registers, TCAMs, and memory arrays (SRAMs).

The GROUPBY problem

- Example Application: Exponentially Weighted Moving Average (EWMA)

- $Avg = \alpha \cdot (New\ Value) + (1 - \alpha) \cdot (Previous\ Avg)$

- Older values are exponentially less important
- Moving Average without requiring to keep track of entire window.

The GROUPBY problem

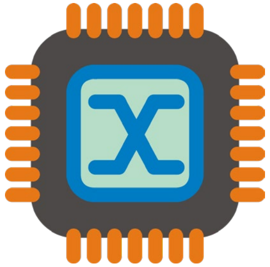
- EWMA of queueing latency of a flow.

- `S:= (switch, hdrs, uid, qid, tin, tout, qsize)`
- `Key:= [hdrs("TCP", SrcIP, SrcPort, DstIP, DstPort), switch]`
- `NewValue:= tout - tin`

- The key space is quite large. On-chip SRAM won't be able to store all the information, and off-chip storage will not achieve line-rate.
- <u>Solution: Cache</u>

Caching:
the illusion of fast and large memory

Caching



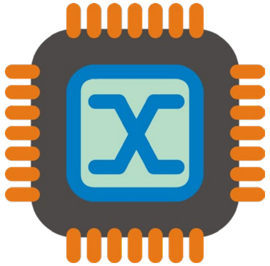
On-chip cache (SRAM)

Key	Value

Off-chip backing store (DRAM)

Key	Value

Caching



Read value for 5-tuple key K

Modify value using ewma

Write back updated value

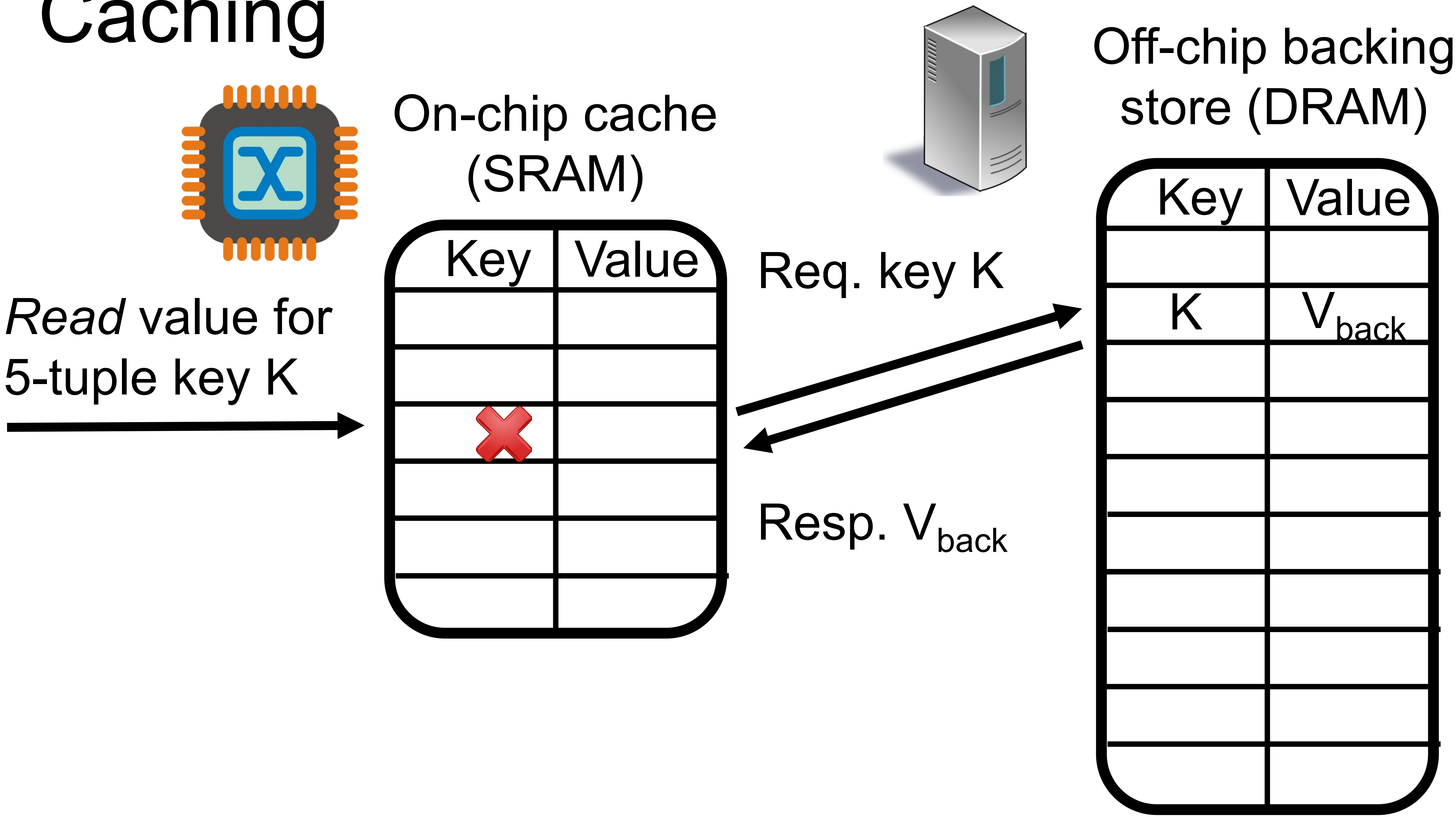
On-chip cache (SRAM)

Key	Value
✓	

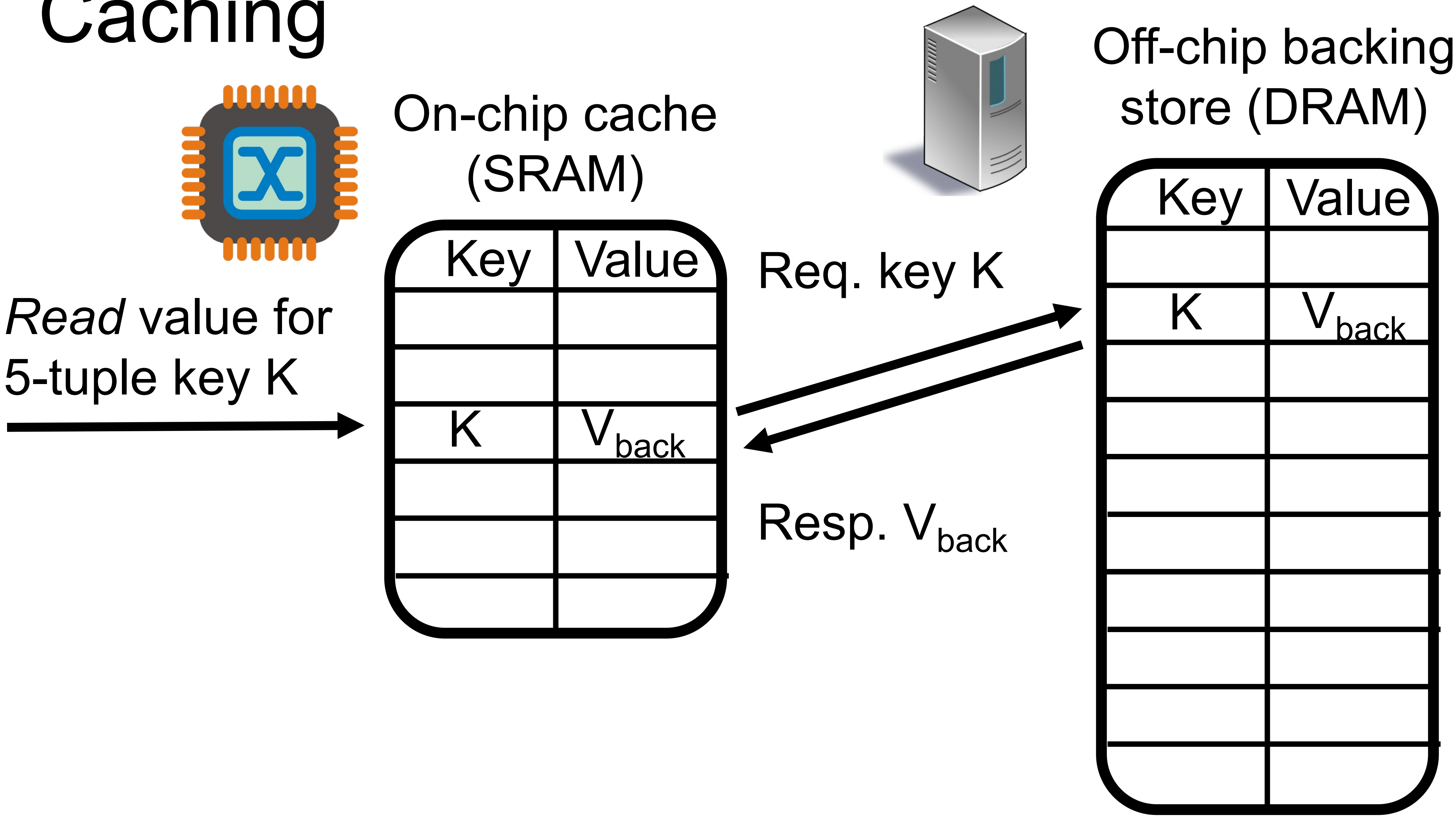
Off-chip backing store (DRAM)

Key	Value

Caching


On-chip cache (SRAM)
Key
Value
Read value for 5-tuple key K
Req. key K
Resp. V_{back}
Off-chip backing store (DRAM)
Key
Value
K
V_{back}

Caching


On-chip cache
(SRAM)
Key
Value
K
V_{back}
Read value for
5-tuple key K
Off-chip backing
store (DRAM)
Key
Value
K
V_{back}
Req. key K
Resp. V_{back}

Caching

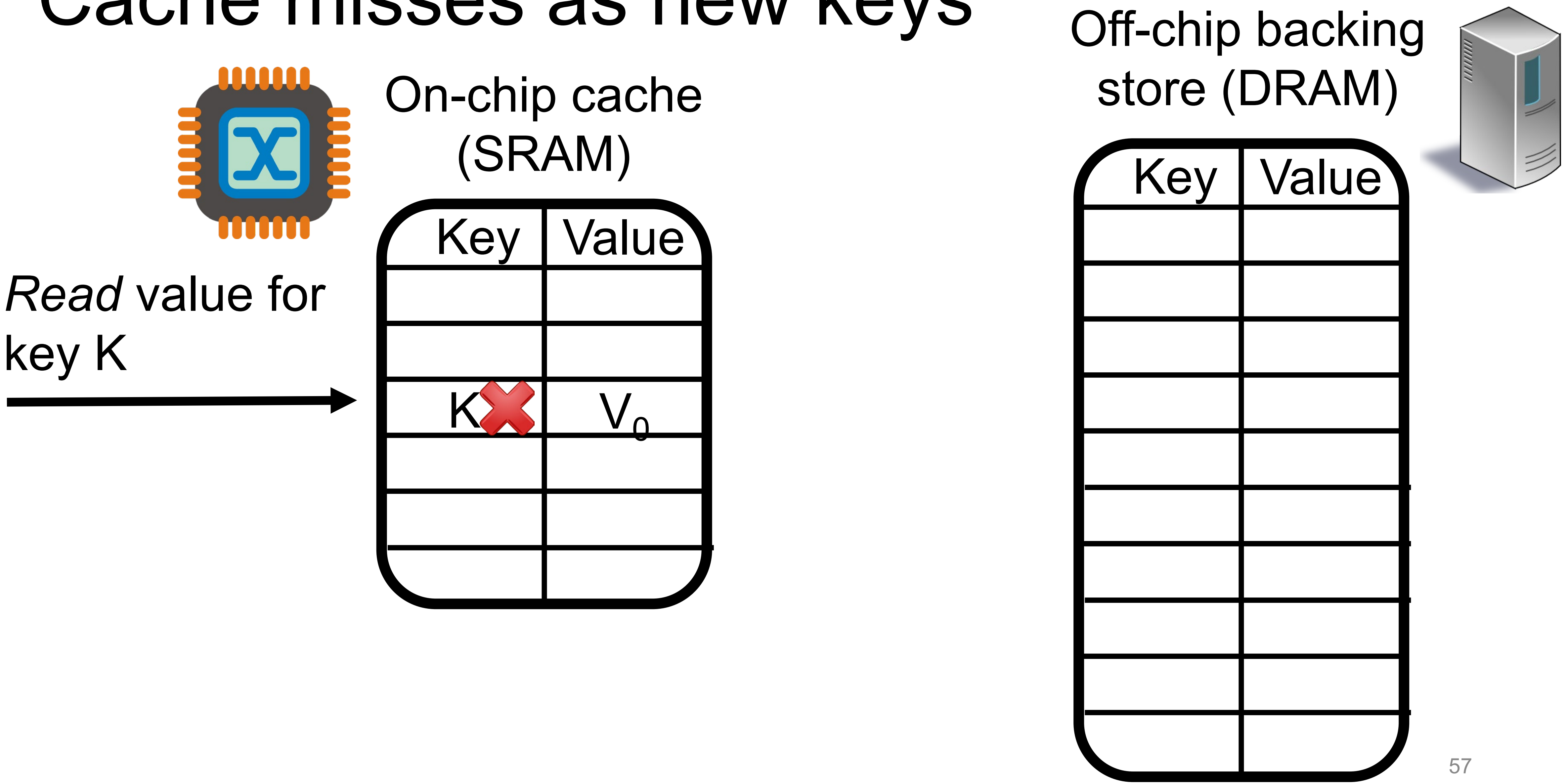
Off-chip backing

Modify and write must wait for DRAM.

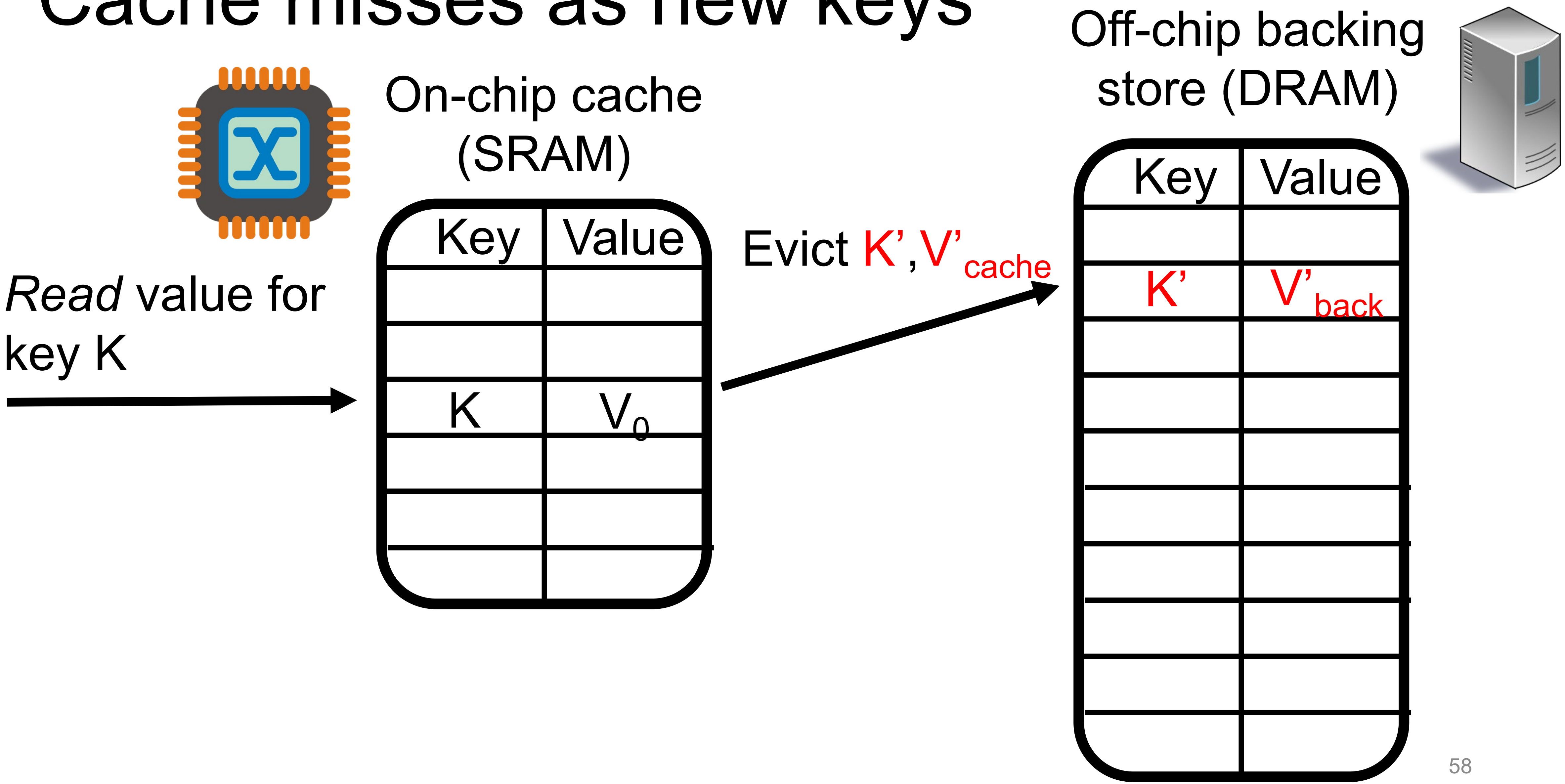
Non-deterministic latencies stall packet pipeline.

Instead, we treat cache misses as packets from new flows.

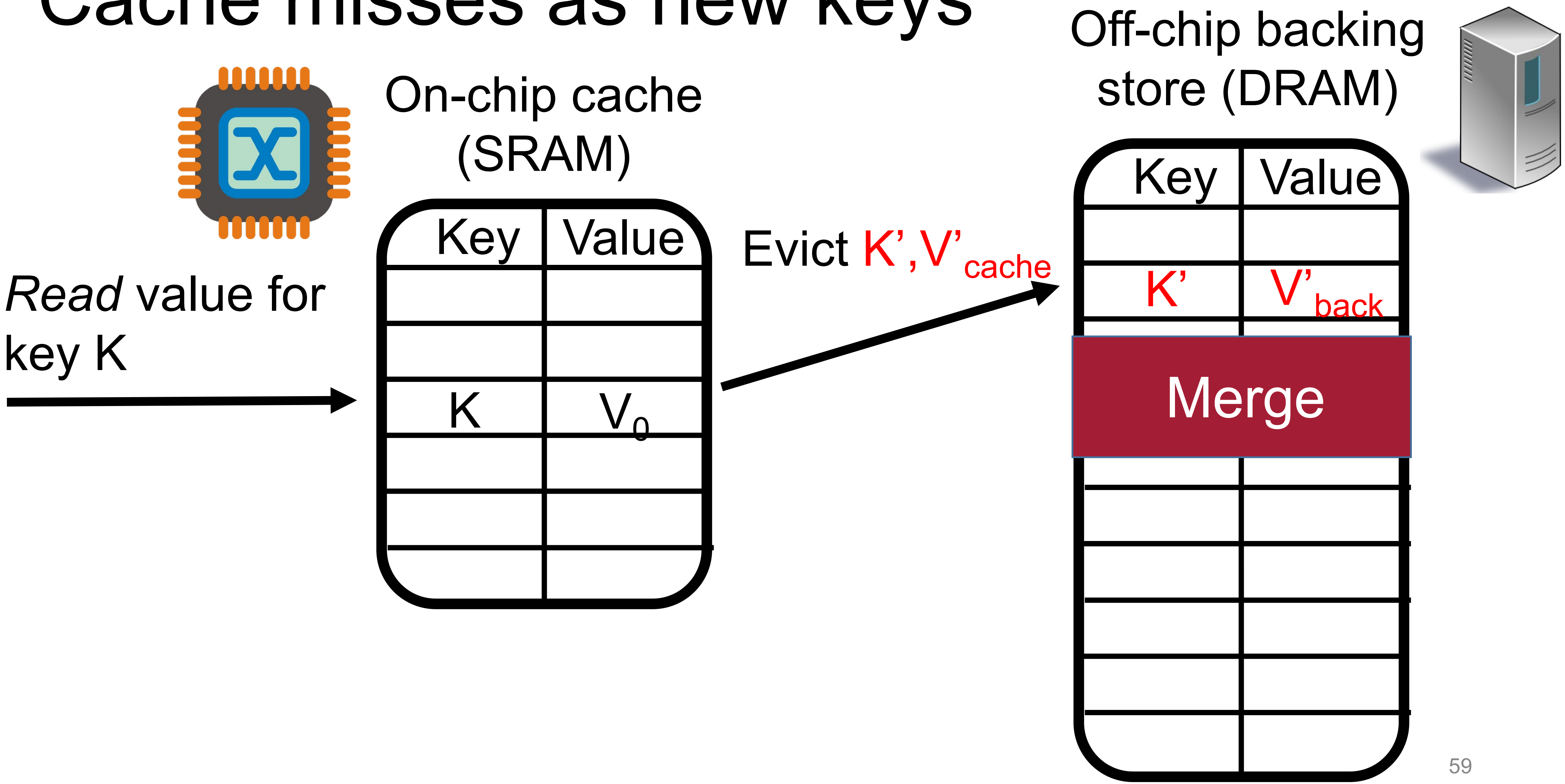
Cache misses as new keys



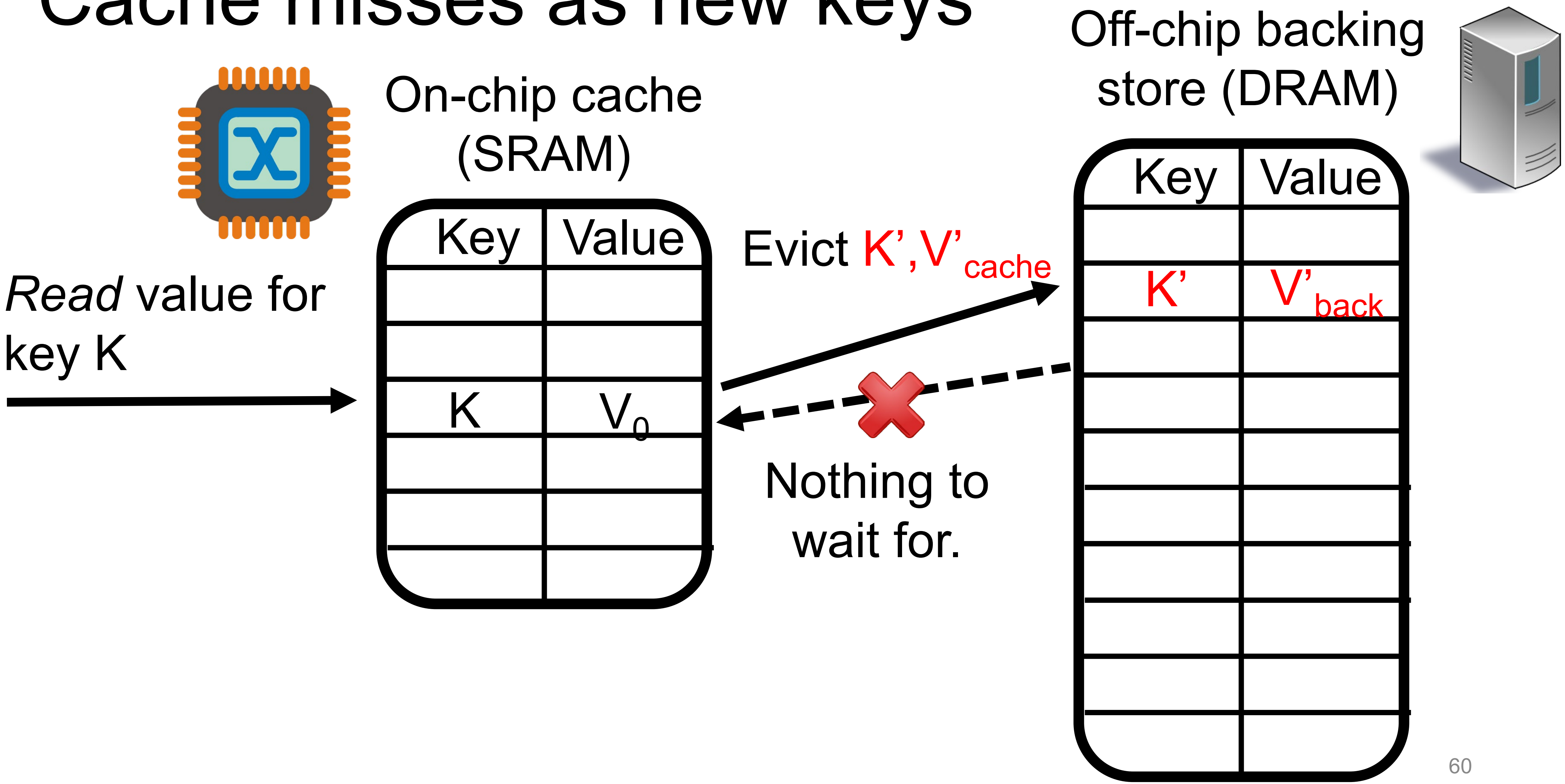
Cache misses as new keys



Cache misses as new keys



Cache misses as new keys



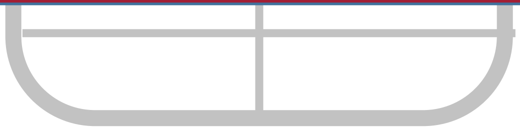
Cache misses as new keys

On-chip cache

Off-chip backing store (DRAM)

Packet processing doesn't wait for DRAM.

Retain 1 pkt/ns processing rate! 👍



The Merge operation

$$merge(\underbrace{g([q_j])}_{V_{cache}}, \underbrace{g([pi])}_{V_{back}})$$

$$= \underbrace{g([p1, \ldots, pn, q1, \ldots, qm])}_{\text{Fold over the entire packet sequence}}$$

- Example: if g is a counter, $merge$ is just addition!

Merging Case 1: Associative

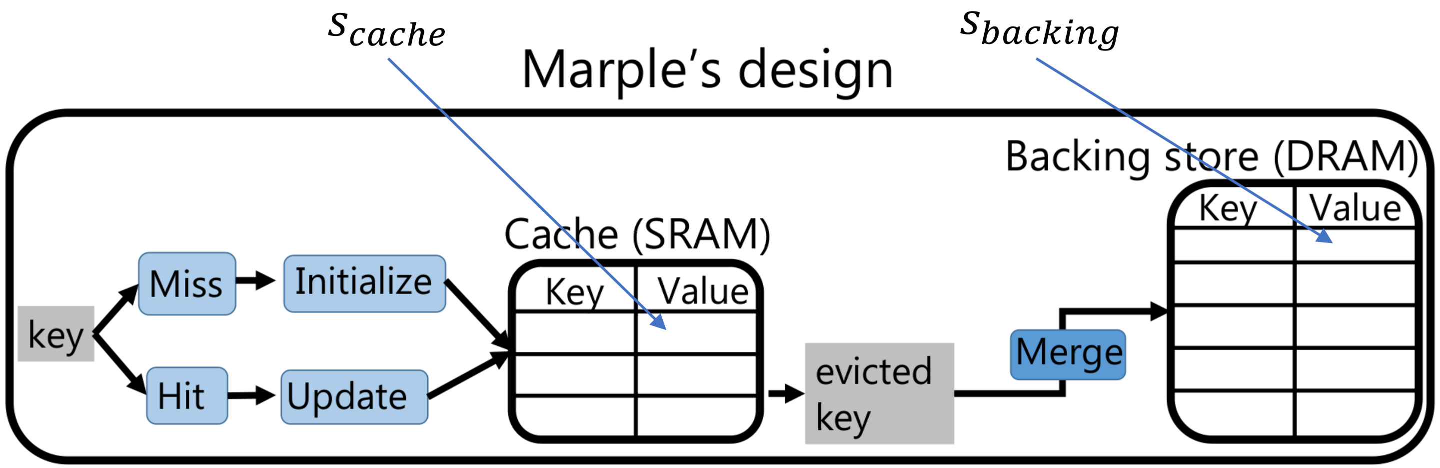
- The operation with each new incoming value is a simple associative operation.
- Example: Counting packets in a flow. Finding maximum queueing time (tout-tin)
- Trivial: Just apply the same function upon eviction.

Merging Case 2: Linear-in-state

- Consider the EWMA again:

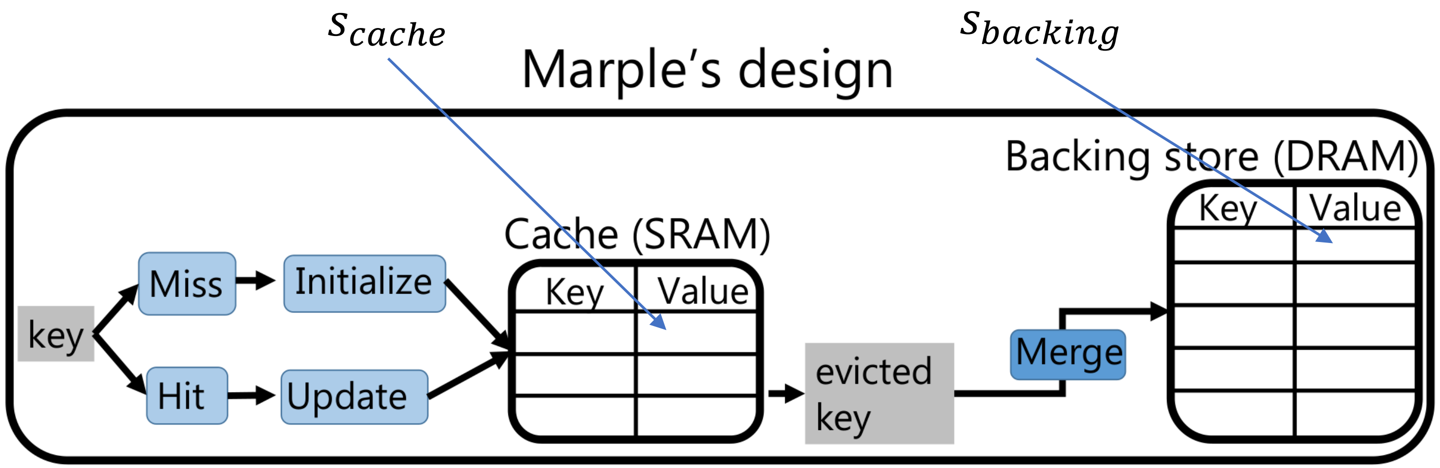
$$Avg = \alpha \cdot (New\ Value) + (1 - \alpha) \cdot (Previous\ Avg)$$

- Denote the stored EWMA value as s. Assume when we initialize EWMA, we set its value to s_0 (which could just simply be 0).



Merging Case 2: Linear-in-state

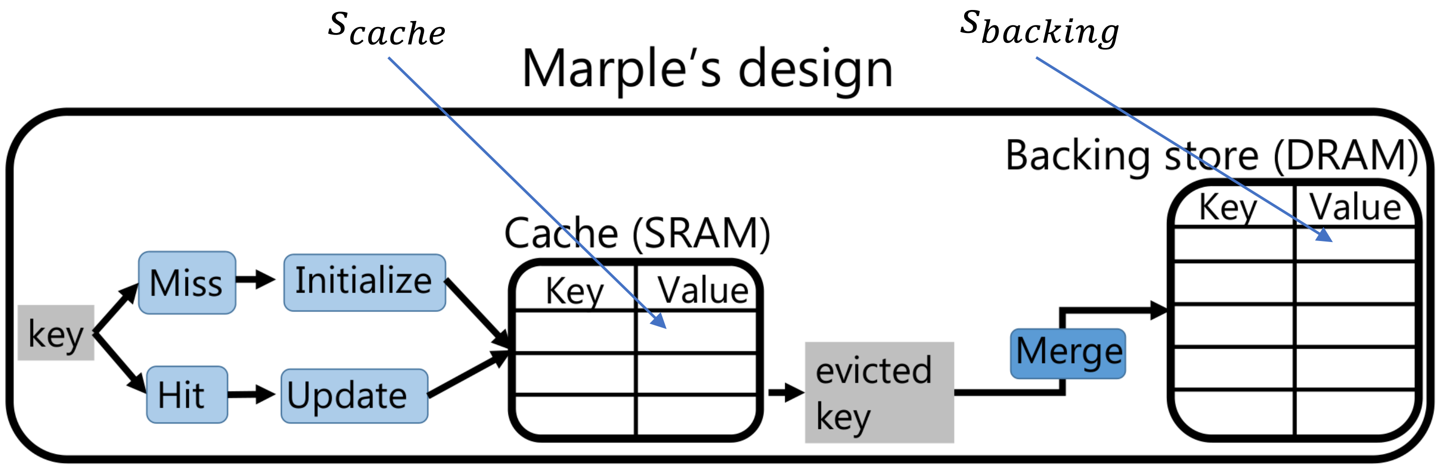
- **Question:** How to merge EWMA cache value with EWMA value in DRAM?
  - $Avg = \alpha \cdot (New\ Value) + (1 - \alpha) \cdot (Previous\ Avg)$
  - $s_{cache} = ewma(p_k \ldots p_j)$
  - $s_{backing} = ewma(p_{j-1} \ldots p_0)$
  - Want $ewma(p_k \ldots p_0)$



Merging Case: Linear-in-state

- $s_{new} = s_{cache} + (1 - \alpha)^N \cdot (s_{backing} - s_0)$

- Need the value of $(1 - \alpha)^N$ (or just N) to calculate the merged value



Merging Case: Linear-in-state

- The state update can be expressed in the form of

$$S = A(\boldsymbol{p}) \cdot S + B(\boldsymbol{p})$$

Where $\boldsymbol{p}$ is header and performance data of last k packets.

$A(\boldsymbol{p}), B(\boldsymbol{p})$ are functions of limited packet history.

k is a integer defined at compile time (and usually small).

- In general, all linear-in-state folding functions only need O(n) auxiliary state to merge them.
- All aggregation functions that maintain a linear auxiliary state is mergeable.

Microbursts: Linear-in-state!

```
def bursty([last_time, nbursts], [tin]):
  if tin - last_time > 800 ms:
    nbursts = nbursts + 1
  last_time = tin

result = groupby(S, 5tuple, bursty)
```

nbursts: S = A * S + B, where

A = 1

B = {1, if current pkt within time gap from last;
0 otherwise}

Other linear-in-state queries

- Counting successive TCP packets that are out of order
- Histogram of flowlet sizes
- Counting number of timeouts in a TCP connection

- … 7/10 example queries in our paper

Merging Case: Non-mergeable

- Remaining non-mergeable cases
  - Queries with aggregation functions that are neither associative nor linear-in-state.
  - GROUPBY aggregation functions with `emit()` – This will emit state value, which requires an instant merge.
    - In some cases, `emit()` can be avoided by rewriting the query.

- Solution: Move to *Domino,* which then compiles to a *Banzai* machine model, which gets mapped to the target platform.
- i.e. Compile to the register, ALU and sALU level of the target platform, and try to fit it into the pipeline. Key space will be limited.

Hardware Feasibility

- The stateful hardware can be broken down into five components.
  - On-chip cache: A hash table implemented with SRAM.
  - Off-chip backing store: A scale-out key-value store, such as Redis.
  - Maintaining packet history – Store in pipeline.
  - Performing Linear-in-state calculations: Multiply-Accumulate instruction.
  - Handling not linear-in-state functions: Domino Atom.

Query Compilation

Theoretical results

Given:

- A user-defined fold function f
- A sequence of packets p
- Want to create an "iterated function" to store in the backing, with:

  $fp(s) = f(s, p)$

  For any backing state s

- The cache stores fp for the current sequence, and that becomes the "merge" function once evicted.

THEOREM 3.1. *Every aggregation function has a corresponding merge function that uses $O(n2^n)$ auxiliary bits.*

For any *f*, we can store *fp* as the answer for *any* possible *s* in the backing store

There are 2^n such *s* and the answers have size n.

THEOREM 3.2. *If an aggregation function is either linear-in-state or associative, it has a merge function that uses $O(n)$ bits of auxiliary state.*

Proof:

- If associative, 0 auxiliary state.
- If linear-in-state, then *f* looks like A(h)*s + B(h), where A and B use only bounded history
- *fp(s, {p1...pk})* can be written A'*s + C(p1...pk)
- A' = A(pk)...A(p1)
- C = B(pk) + A*B(pk-1)+ … A*…*A*B(p1)
- So, the switch can just store/update A' and C, which each have size linear in n.

Running example

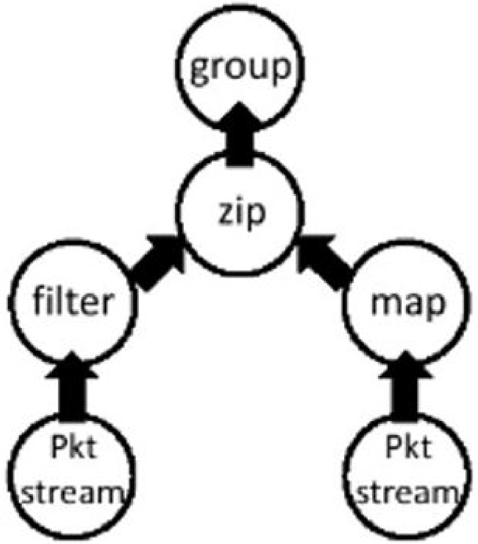
```
def oos_count([count, lastseq], [tcpseq, payload_len]):
  if lastseq != tcpseq:
    count = count + 1
    emit()
  lastseq = tcpseq + payload_len

tcps    = filter(pktstream, proto == TCP
                  and (switch == S1 or switch == S2));
tslots  = map(pktstream, [tin/epoch_size], [epoch]);
joined  = zip(tcps, tslots);
oos     = groupby(joined,
                   [5tuple, switch, epoch],
                   oos_count);
```

Network-wide to switch programs

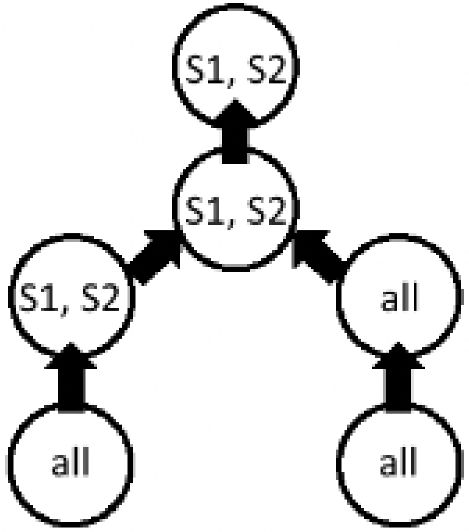
- Goal: take a query about an abstract stream and output a program for each switch in the network
- Solution: Syntactically check each filter predicate to determine which switches should have each function.

Network-wide to switch programs

```
def oos_count([count, lastseq], [tcpseq, payload_len]):
  if lastseq != tcpseq:
    count = count + 1
    emit()
  lastseq = tcpseq + payload_len

tcps    = filter(pktstream, proto == TCP
                 and (switch == S1 or switch == S2));
tslots  = map(pktstream, [tin/epoch_size], [epoch]);
joined  = zip(tcps, tslots);
oos     = groupby(joined,
                  [5tuple, switch, epoch],
                  oos_count);
```



(a)



(b)

Permitted queries

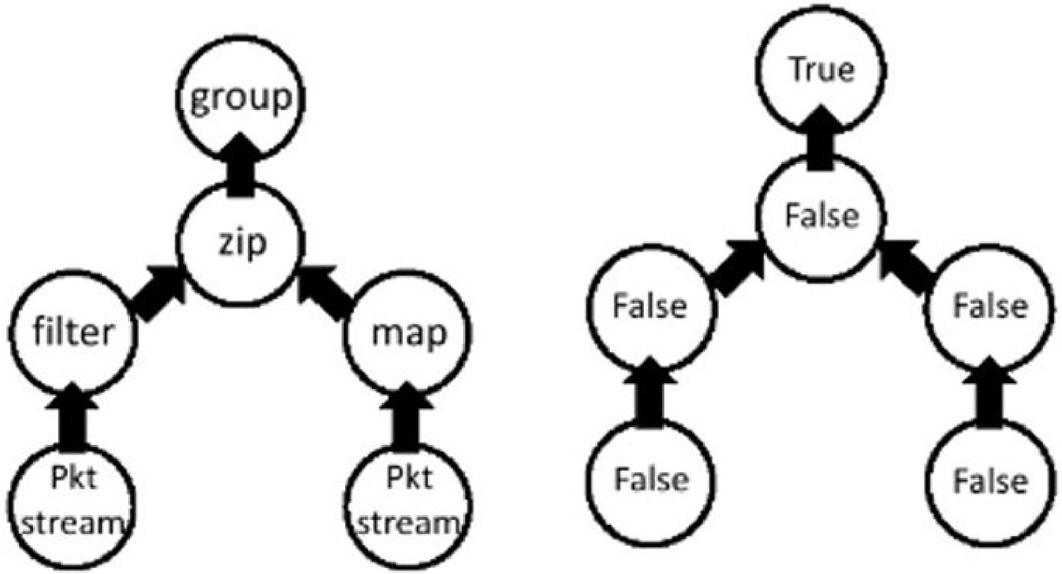
- Operate independently on each switch:
  - Check AST of query
- Operate independently per packet:
  - Check that groupby aggregates by uid
- Operations are associative and commutative
  - Programmer must annotate

Checking if queries are per-switch

- Propagate whether a stream is switch-partitioned through the query:
- Base packetstream is not partitioned.
- Filter and zip outputs are switch-partitioned based on syntactic check
- Map preserves partition
- Groupby is switch-partitioned if it aggregates by switch.

Checking if queries are per-switch

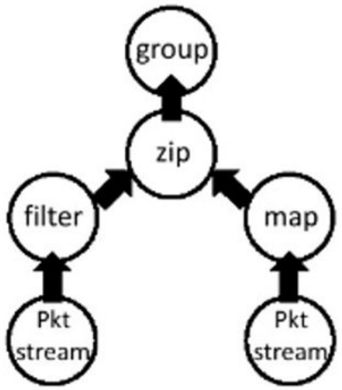
```
def oos_count([count, lastseq], [tcpseq, payload_len]):
  if lastseq != tcpseq:
    count = count + 1
    emit()
  lastseq = tcpseq + payload_len

tcps    = filter(pktstream, proto == TCP
                 and (switch == S1 or switch == S2));
tslots  = map(pktstream, [tin/epoch_size], [epoch]);
joined  = zip(tcps, tslots);
oos     = groupby(joined,
                  [5tuple, switch, epoch],
                  oos_count);
```



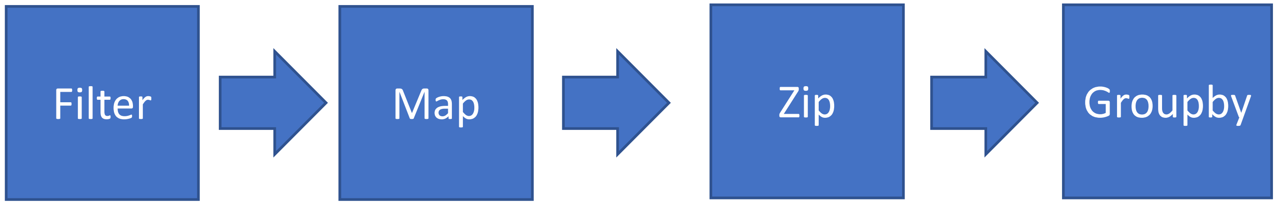
(a) (c)

Creating pipeline configuration

- After the previous checks, Marple has per-switch programs to place into the pipeline
- Must care to avoid read/write dependencies; place AST in reverse order



(a)



Per-switch AST to code

- The hard query to compile is groupby
- Filter, zip is just checking predicate and setting a bit in the packet
- Map adds a new header field with the resulting expression
- Transform GroupBy aggregation functions into a series of if statements that fit into a P4 action, with a register storing variables
  - Use program analysis algorithm from the 70s
- Domino can directly handle the series of if-statements.

How to detect linear-in-state functions

- Very difficult to detect *all* LIS functions
- Enough to have syntax checks, but not algebraic rewriting
- Suffices to check that all variables (state in register, or headers) are linear in state
- All header variables are clearly LIS

Step 1: History of variables

- For each variable, check how many packets it depends on
- Headers of the current packet are 1
- State replaced per packet are 2
- Counters that contain every packet are infinite

Step 1: History of variables

- Hard to check whether a state (register) variable is LIS

```
hist = {state = {true: max_bound}}            ▷ Init. hist. for all state vars.
function COMPUTEHISTORY(fun)
    while hist is still changing do                       ▷ Run to fixed point.
        hist ← {}
        ctx ← true                                ▷ Set up outermost context.
        ctxHist ← 0                                   ▷ History value of ctx.
        for stmt in fun do
            if stmt == state = expr then
                hist[state][ctx] ← GETHIST(ctx, expr, ctxHist)
            else if stmt == if predicate then
                save context info (restore on branch exit)
                newCtx ← ctx and predicate
                ctxHist ← GETHIST(ctx, newCtx, ctxHist)
                ctx ← newCtx
            end if
        end for
        for ctx, var in hist do                  ▷ Make history one pkt older.
            hist[var][ctx] ← min(hist[var][ctx] + 1, max_bound)
        end for
    end while
end function
```

```
function GETHIST(ctx, ast, ctxHist)
    for xi ∈ LEAFNODES(ast) do
        hi = hist[xi][ctx]
    end for
    return max(h1, ... , hn, ctxHist)
end function
```

Step 1: History of variables

- Assume state variables have infinite history for safety
- Check each assignment:
  - If a state variable is assigned to an expression with finite history, it has finite history
  - Check branches for the maximum history of:
    - Predicate
    - Branch 1
    - Branch 2
  - Continue until a fixpoint is reached (propagates constant histories as far as possible)
  - Each loop, increment all histories (until ficxpoint)

Step 2: History of all variables

- If all state variables have finite history, then the update is LIS, since we can just send a finite packet history
- If some are infinite, then we have to check if their *updates* are linear-in-state (for example, the EWMA example)
  - Done by simply checking syntactically if assignment looks like

    S := A.S + B

  - Where A, B are expressions with finite history (as computed before)
- Branching predicates cannot have infinite history

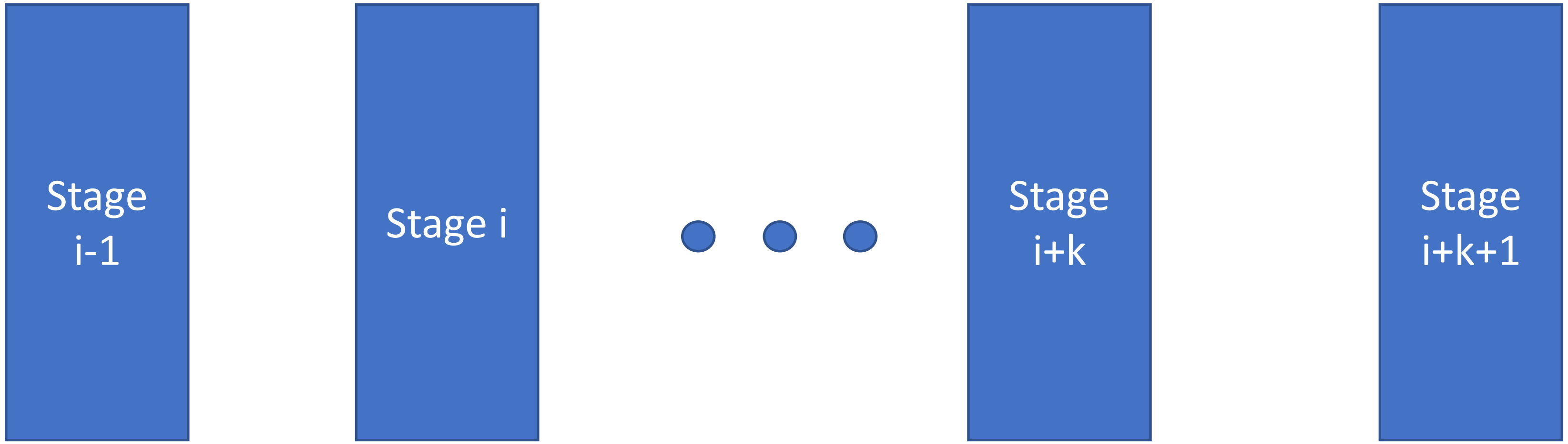
Step 3: Determine Auxiliary State

- Once we have checked that State variables are LIS, we need to determine what is stored in the registers. Each state variable gets:
  - A packet counter c
  - An entry log (logs from insertion)
  - An exit log (logs most recent packets)
  - A running product S
- Once c is bigger than the LIS bound, we start multiplying S by the LIS matrix A for every update to the variable
- On eviction, send S and the exit log so that the store can merge

Question: How do you store the exit log

- Must store the previous k packets so that the backing store can compute "C" to merge

Maintaining packet history



Compute
hash idx

Cur = hdrs

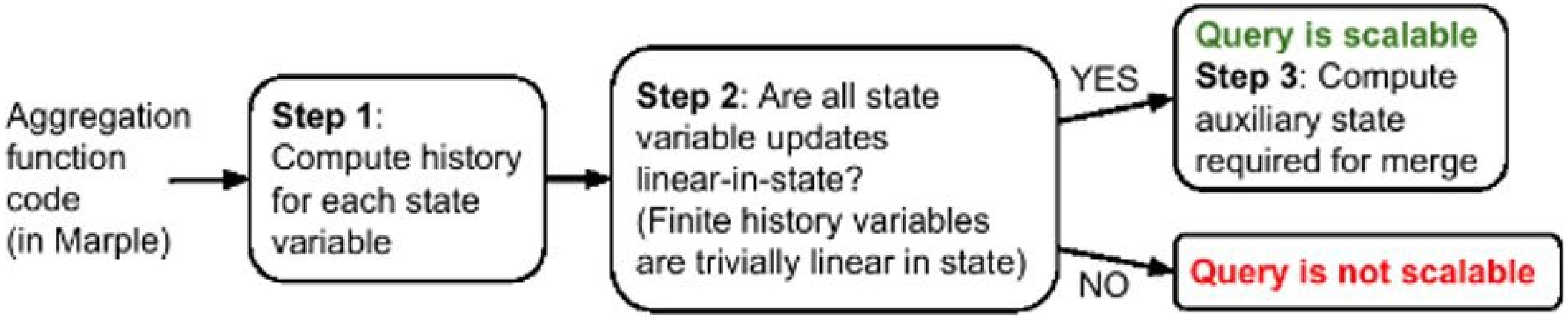
P1 = read(idx)
write(idx, Cur)
Cur = P1

Pk = read(idx)
write(idx, Cur)

Compute A(P1...Pk)
And B(P1...Pk), and do
update.

Move all stored packets down the line

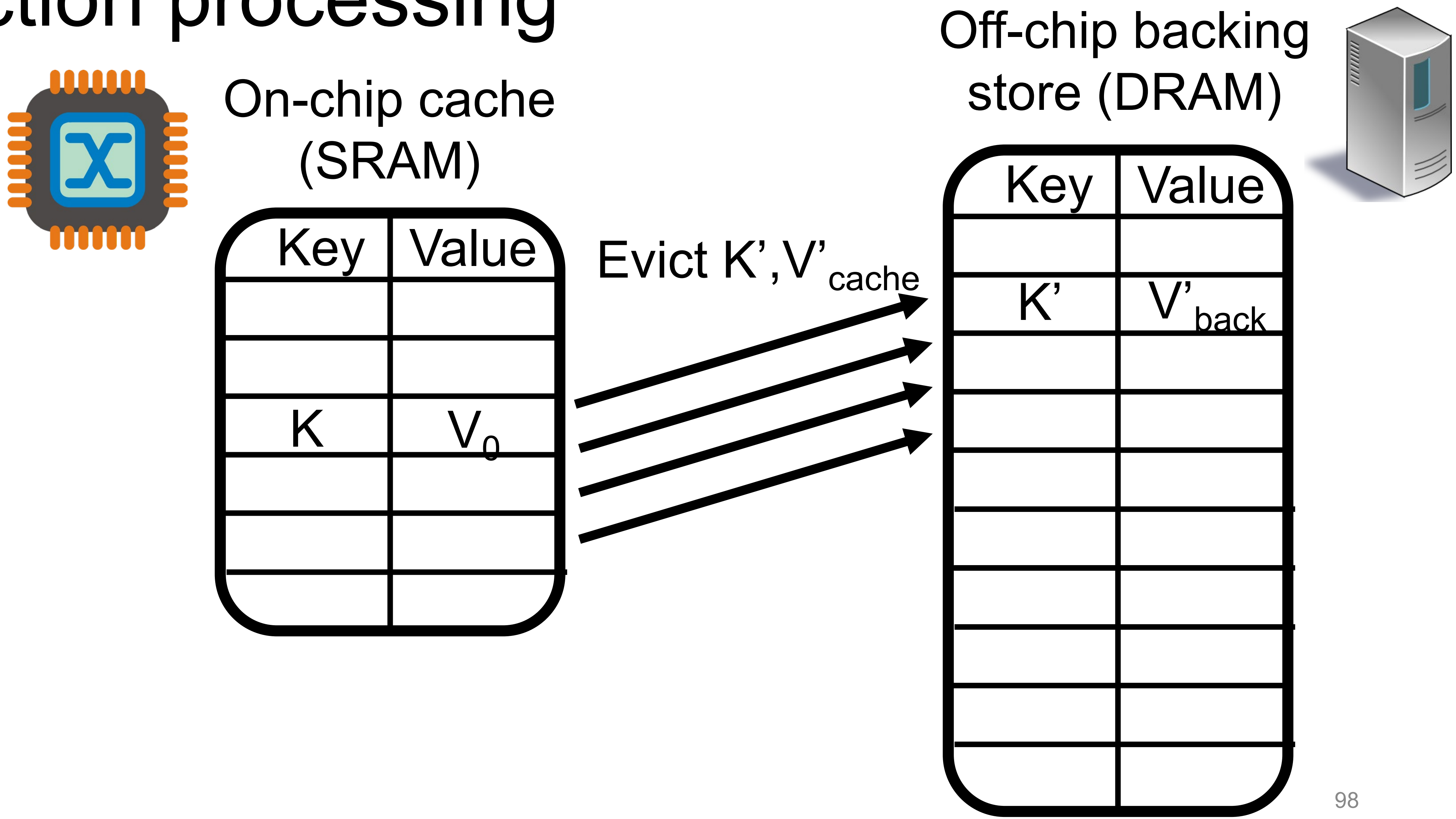
Summary



Aggregation function code (in Marple)
Step 1: Compute history for each state variable
Step 2: Are all state variable updates linear-in-state? (Finite history variables are trivially linear in state)
YES
Query is scalable
Step 3: Compute auxiliary state required for merge
NO
Query is not scalable

Evaluation:
Is processing the evictions feasible?

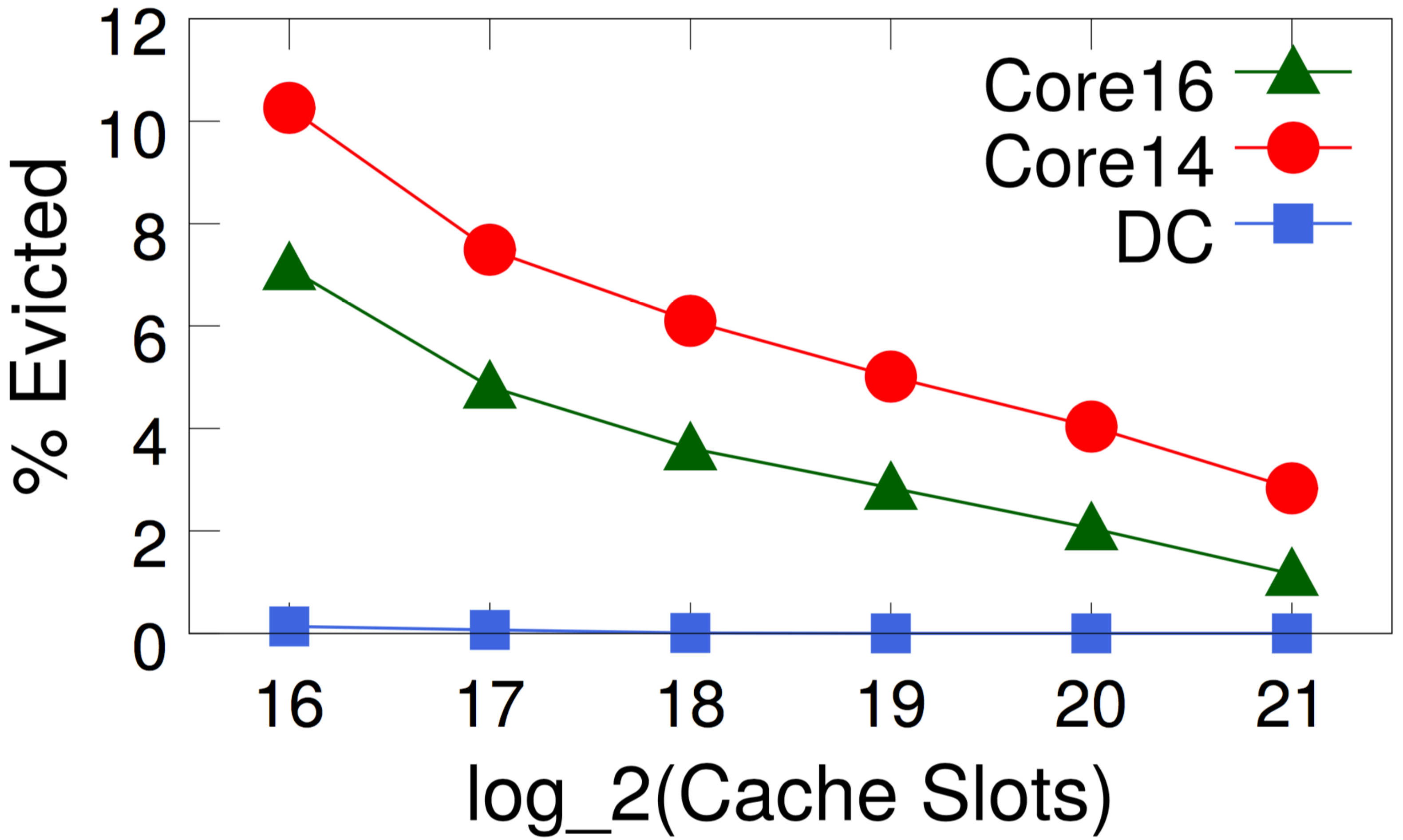
Eviction processing



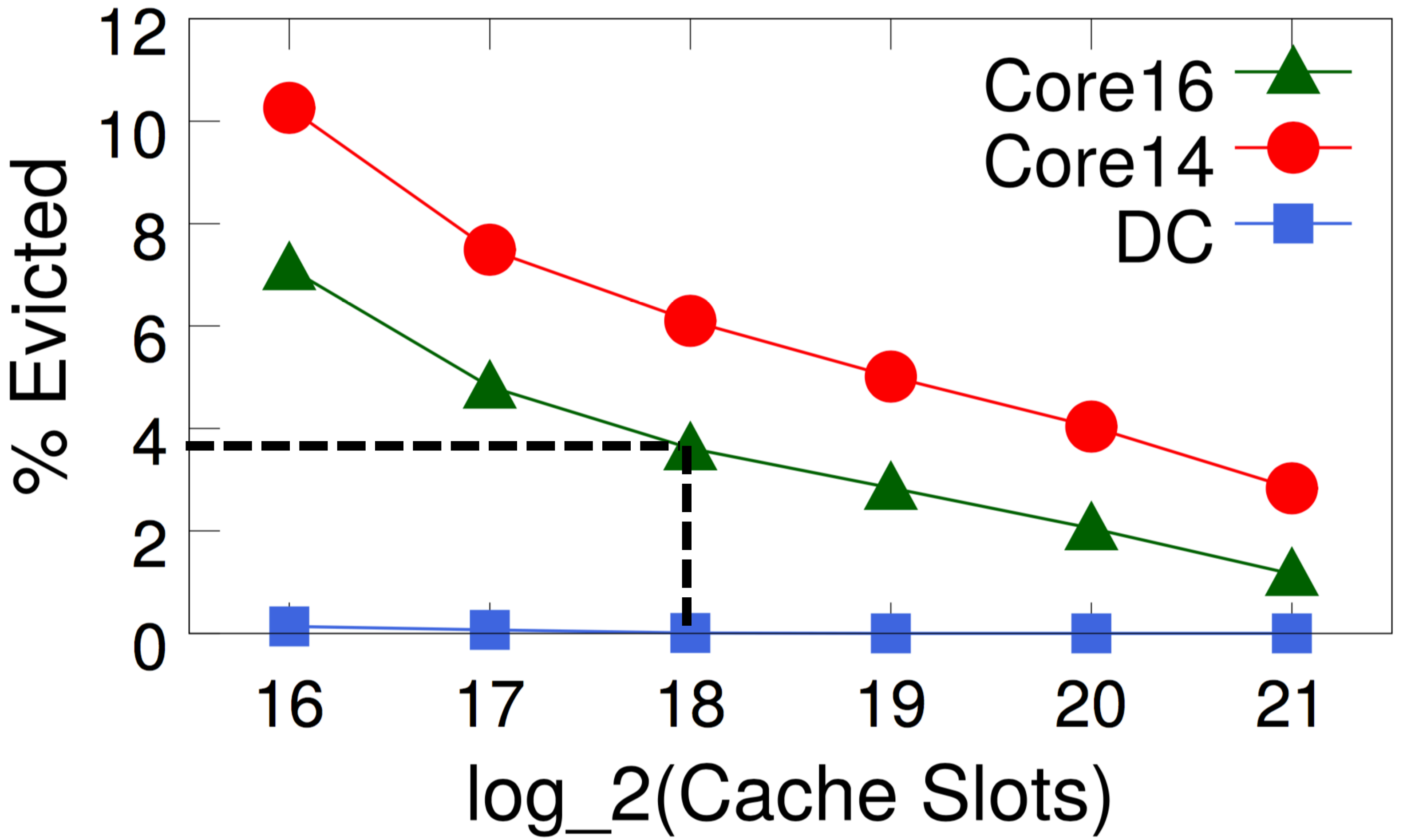
Eviction processing at backing store

- Trace-based evaluation:
  - “Core14”, “Core16”: Core router traces from CAIDA (2014, 16)
  - “DC”: University data center trace from [Benson et al. IMC ’10]
  - Each has ~100M packets
- Query aggregates by 5-tuple (key)
  - Show results for key+value size of 256 bits
- 8-way set-associative LRU cache eviction policy
- Eviction *ratio*: % of incoming pkts that result in a cache eviction

Eviction ratio vs. Cache size



Eviction ratio vs. Cache size



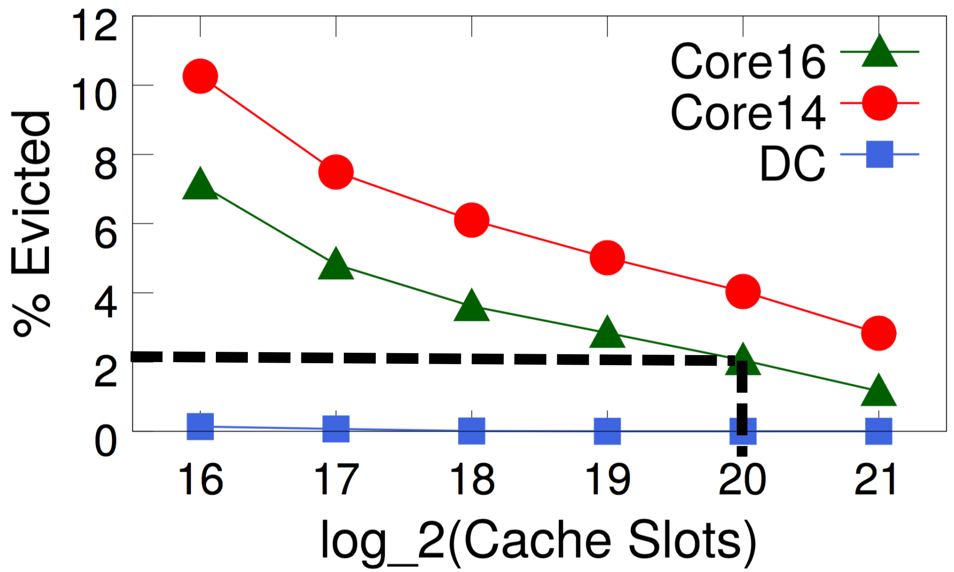
2^{18} keys == 64 Mbits

4% pkt eviction ratio

25X reduction from processing each pkt

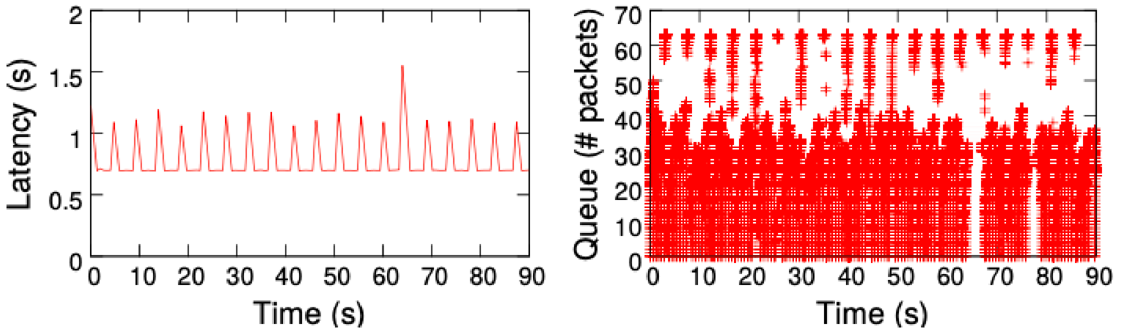
Eviction ratio ➔ Eviction rate

- Consider 64-port X 100-Gbit/s switch
- Memory: 256 Mbits
  - 7.5% area
- Eviction rate: 8M records/s
  - **~ 32 cores**

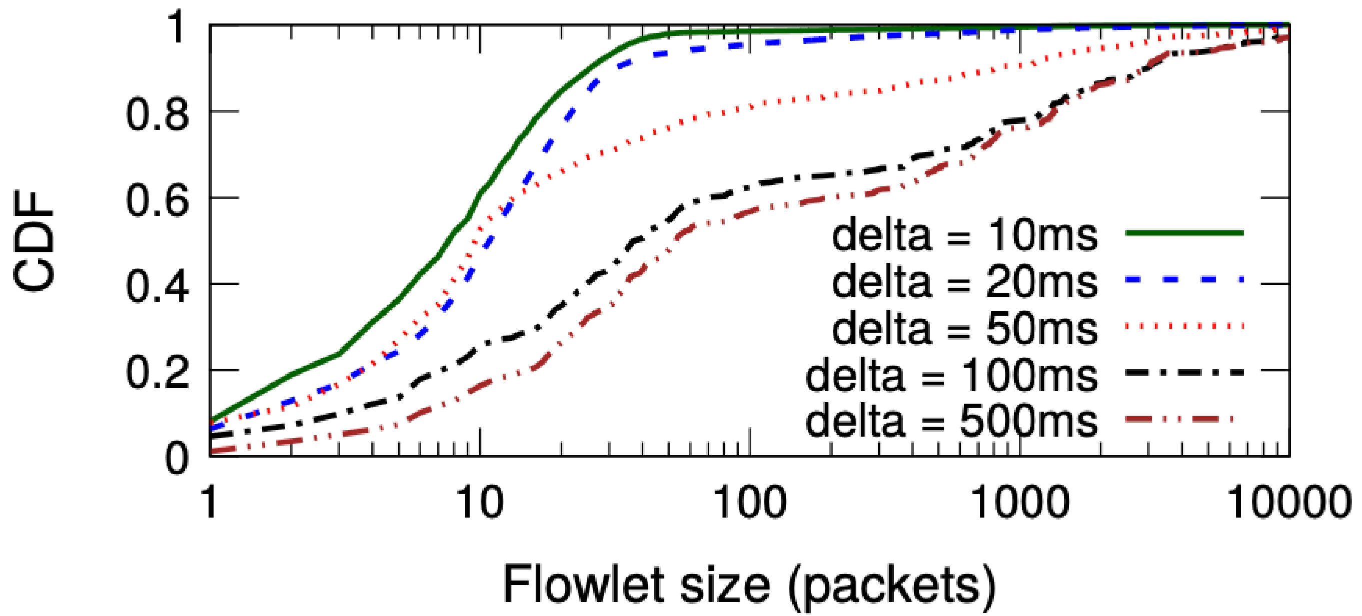


Debugging Microbursts

```
def burst_stats([last_time, nburst, time], [pkts, tin]):
    if tin - last_time > 800000:
        nbursts++;
        emit();
    else:
        time = time + tin - last_time;
    pkts = pkts + 1;
    last_time = tin;
result = groupby(R1, 5tuple, burst_stats)
```



CDF of flowlet sizes for different flowlet thresholds.



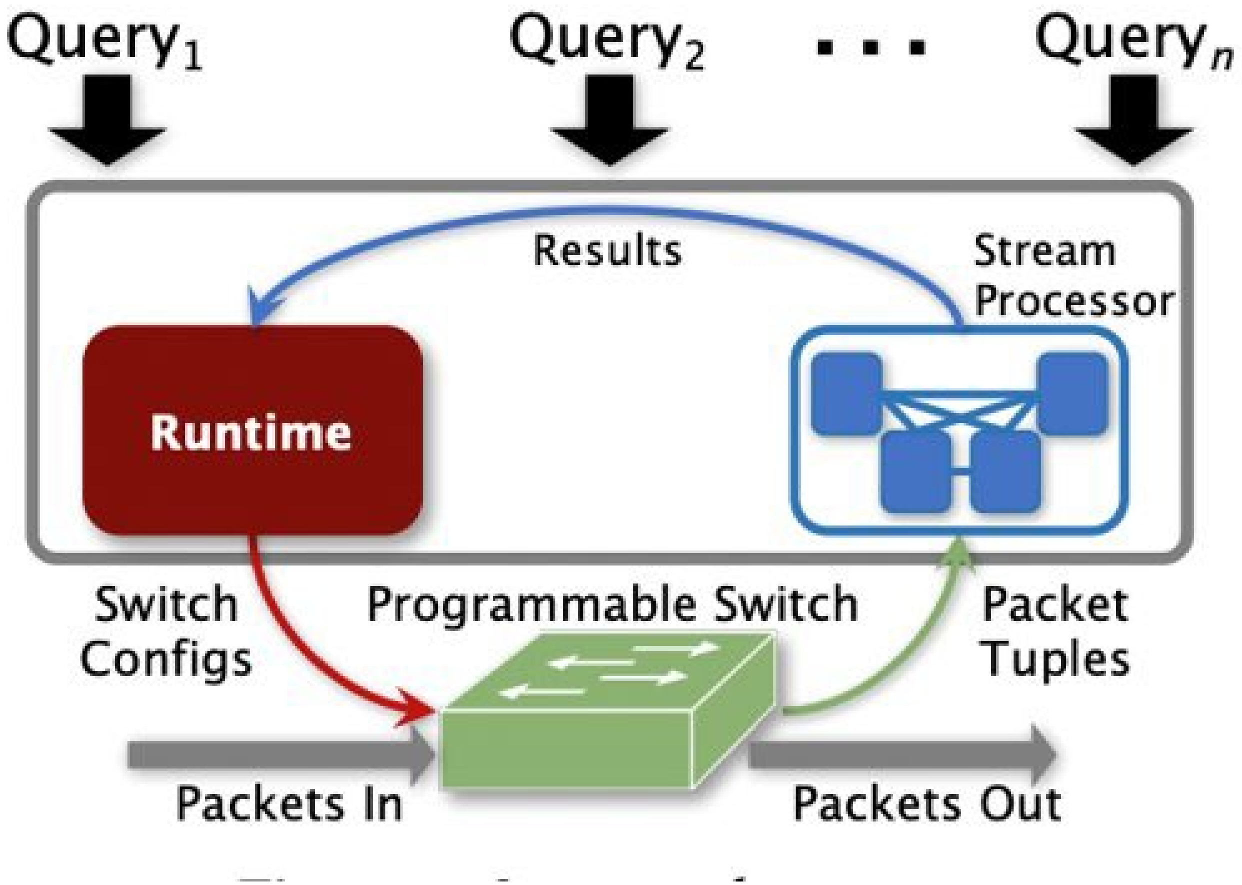
See more in the paper…

- More performance query examples
- Query compilation algorithms
- Evaluating hardware resources for stateful computations
- Implementation & end-to-end walkthroughs on mininet

Further work: Sonata

- Marple focuses on executing queries in the data plane
- The Key-Value store deals with cache sizing
- Sonata adds a central query controller/stream processor that allows for more complex queries
- Sonata does "what it can" on the switch and forwards intermediate results to the stream processor
- Similar to Marple, the main issue is the "join" operation that joins two streams.

Sonata



Query$_1$
Query$_2$
. . .
Query$_n$
Results
Stream Processor
Runtime
Switch Configs
Programmable Switch
Packet Tuples
Packets In
Packets Out

Sonata Queries

```
packetStream(W)
.filter(p => p.tcp.flags == 2)
.map(p => (p.dIP, 1))
.reduce(keys=(dIP,), f=sum)
.filter((dIP, count) => count > Th)
```

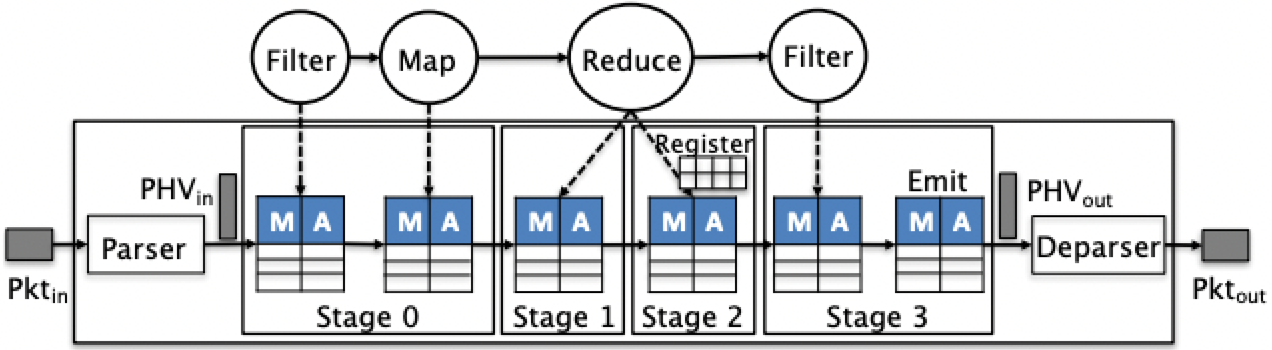
Query 1: *Detect Newly Opened TCP Connections.*

Sonata Queries

```
packetStream
.filter(p => p.proto == TCP)
.map(p => (p.dIP,p.sIP,p.tcp.sPort))
.distinct()
.map((dIP,sIP,sPort) =>(dIP,1))
.reduce(keys=(dIP,), f=sum)
.join(keys=(dIP,), packetStream
  .filter(p => p.proto == TCP)
  .map(p => (p.dIP,p.pktlen))
  .reduce(keys=(dIP,), f=sum)
  .filter((dIP, bytes) => bytes > Th1) )
.map((dIP,(byte,con)) => (dIP,(con/byte))
.filter((dIP, con/byte) => (con/byte > Th2)
```

Query 2: *Detect Slowloris Attacks.*

Switches: Similar layout



Problem: too much traffic

- The stream processor cannot handle all events from all individual keys.
- Solution: make queries more general if there are too many keys
- Called refinement; incrementally refine until manageable

```
packetStream(W)
.filter(p => p.tcp.flags == 2)
.map(p => (p.dIP, 1))
.reduce(keys=(dIP,), f=sum)
.filter((dIP, count) => count > Th)
```

Query 1: *Detect Newly Opened TCP Connections.*