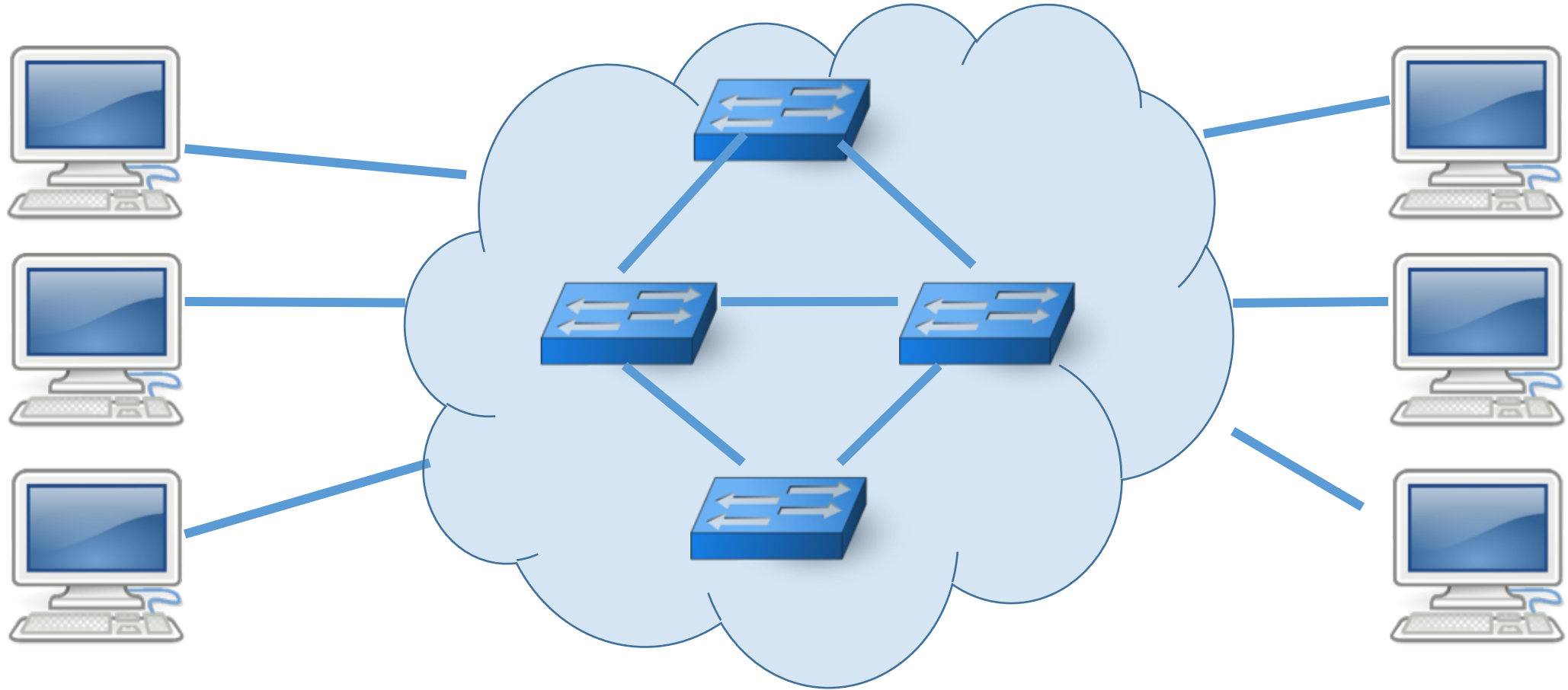


Designing fast and programmable routers

Anirudh Sivaraman



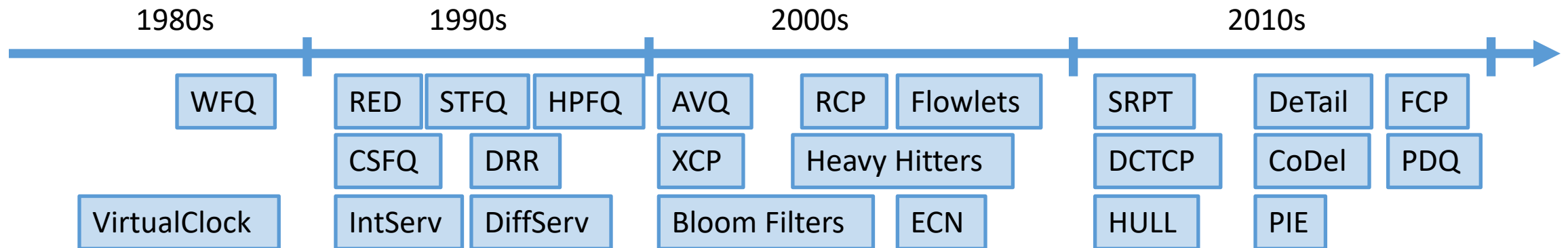
Traditional network architecture



Simple routers; most functionality resides on end hosts

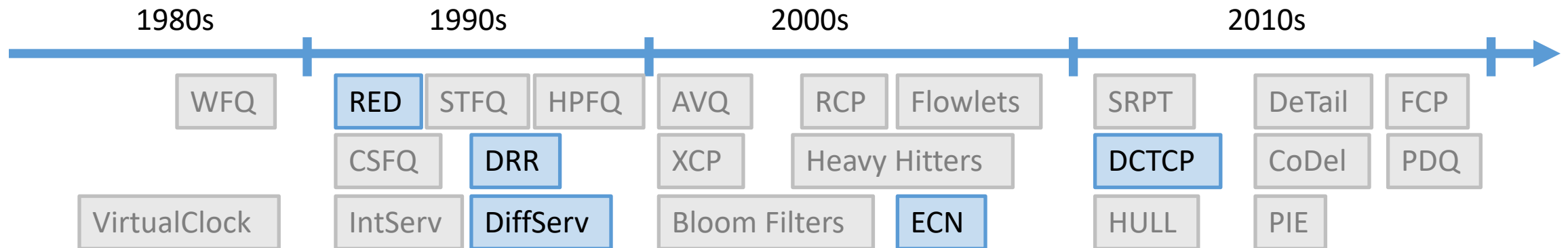
But, today's reality is very different

- We are demanding more from routers: ACLs, tunnels, measurement etc
- Yet, the fastest routers have historically been fixed-function
- Rate of innovation exceeds our ability to get things into routers

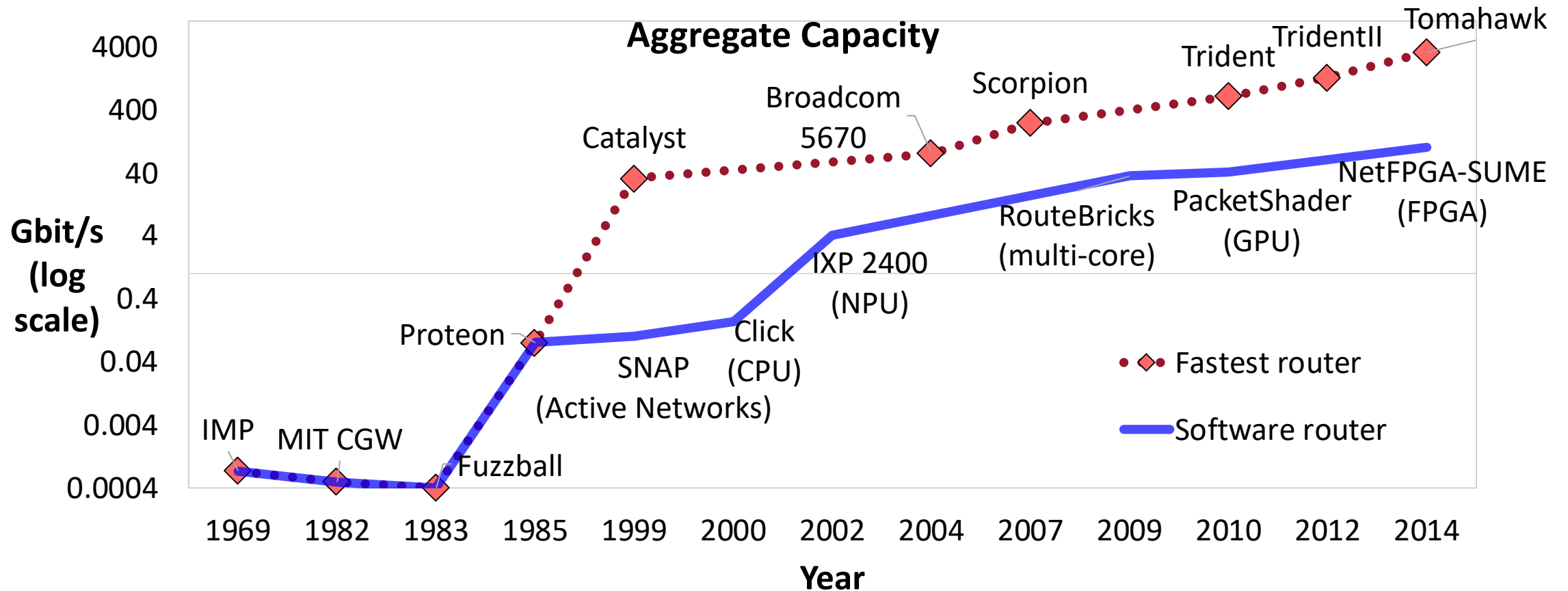


But, today's reality is very different

- We are demanding more from routers: ACLs, tunnels, measurement etc
- Yet, the fastest routers have historically been fixed-function
- Rate of innovation exceeds our ability to get things into routers



One approach: Use a software router

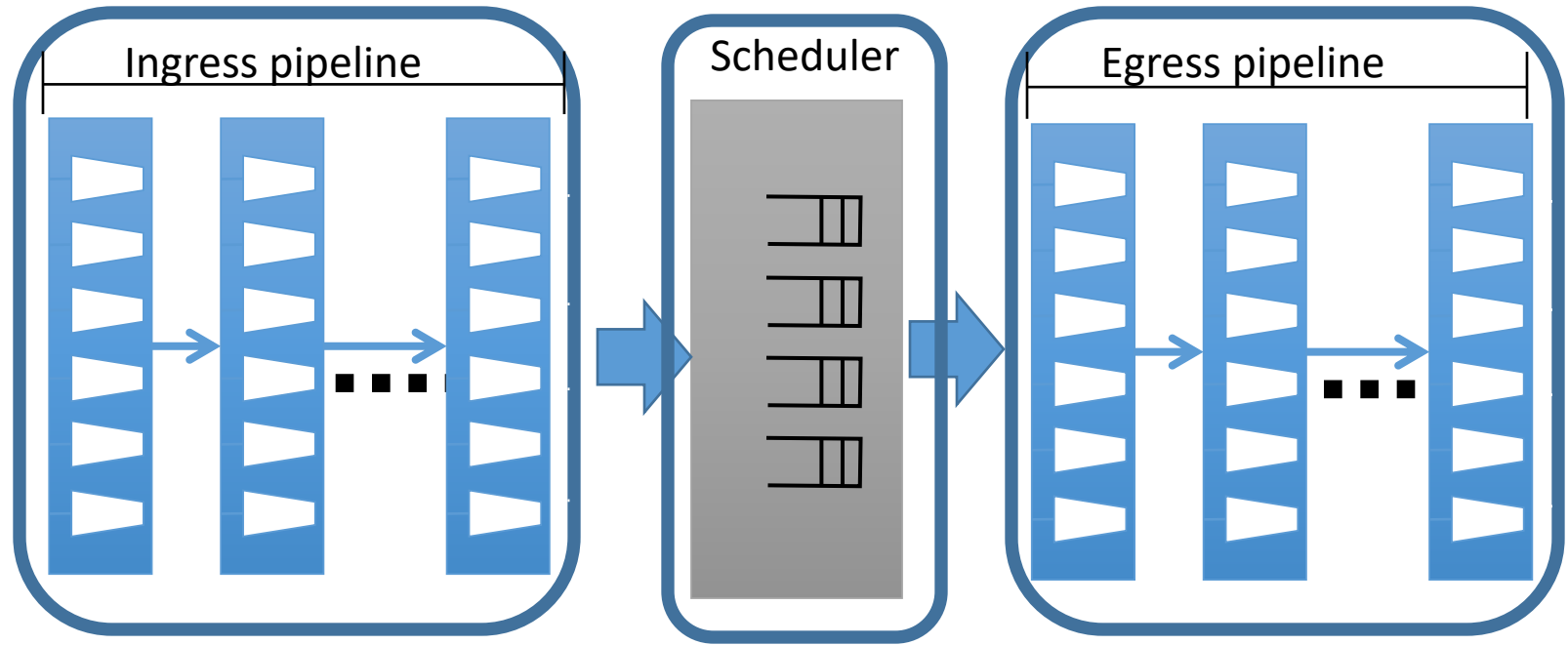


Software routers 10—100x slower than the fastest routers

Software routers suffer from non-deterministic performance

My work: performance+programmability

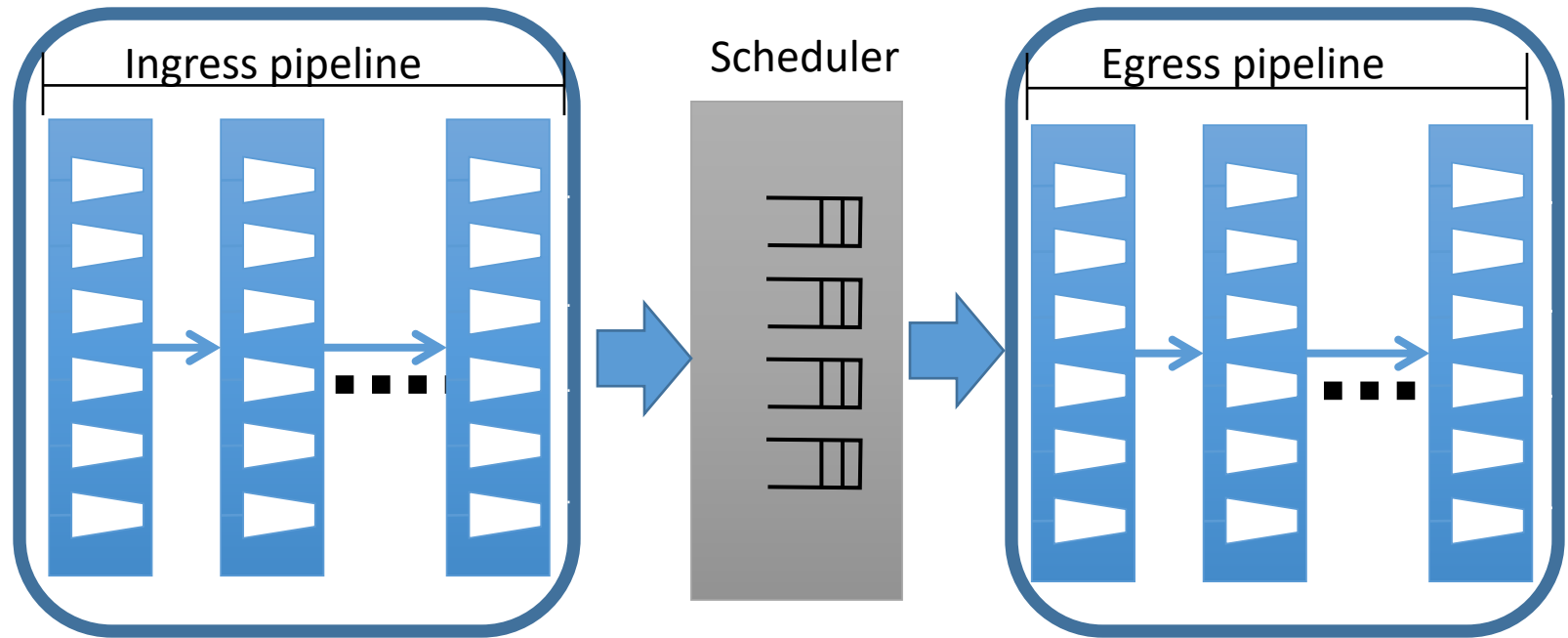
- Domino (SIGCOMM '16):
programming streaming
algorithms
- PIFO (SIGCOMM '16):
programming scheduling
algorithms
- Marple (SIGCOMM '17):
programmable and scalable
measurement



Performance+programmability for important classes of router functions

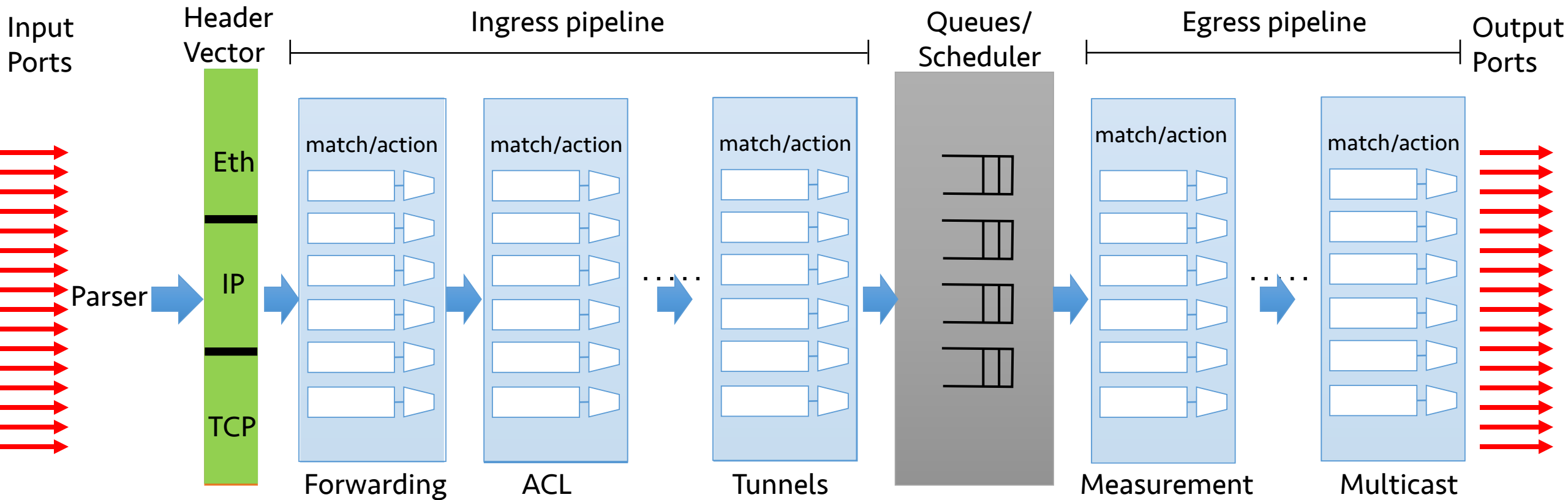
My work: performance+programmability

- Domino (SIGCOMM '16):
programming streaming
algorithms
- PIFO (SIGCOMM '16):
programming scheduling
algorithms
- Marple (SIGCOMM '17):
programmable and scalable
measurement



Performance+programmability for important classes of router functions

A fixed-function router pipeline

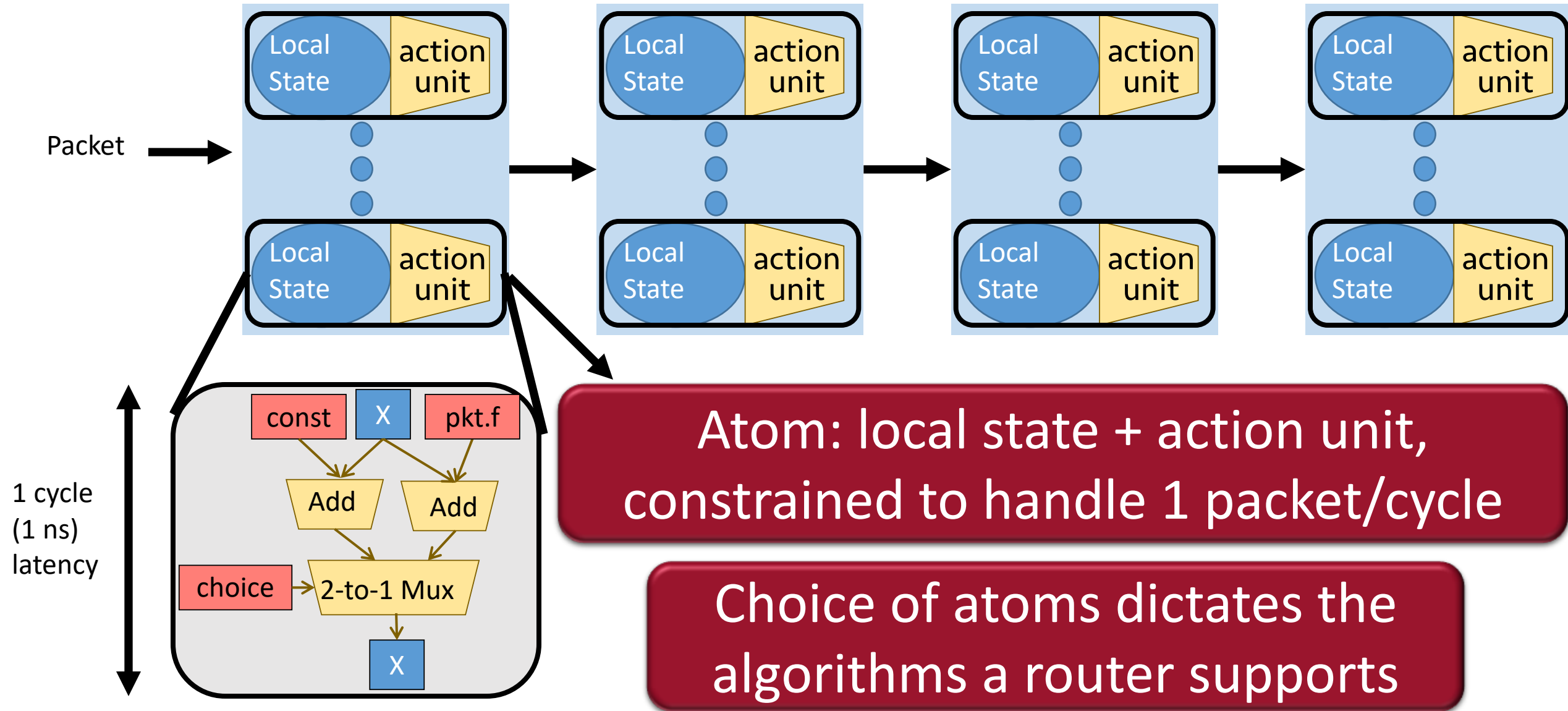


Deterministic pipelines supporting a throughput of 1 packet/cycle (1 GHz)

State is local to action units

Constrained action units

A programmable atom pipeline



The Domino compiler

Input: Algorithms as packet transactions

```
if (count == 9):  
    pkt.sample = pkt.src  
    count = 0  
else:  
    pkt.sample = 0  
    count++
```

Code
pipelining
→

Pipeline stages

Stage 1

```
pkt.old = count;  
pkt.tmp = pkt.old == 9;  
pkt.new = pkt.tmp ? 0  
           :(pkt.old + 1);  
count = pkt.new;
```

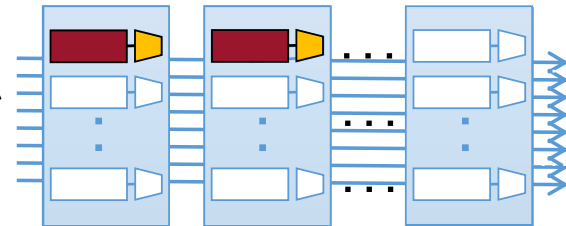
Stage 2

```
pkt.sample = pkt.tmp ?  
             pkt.src : 0
```

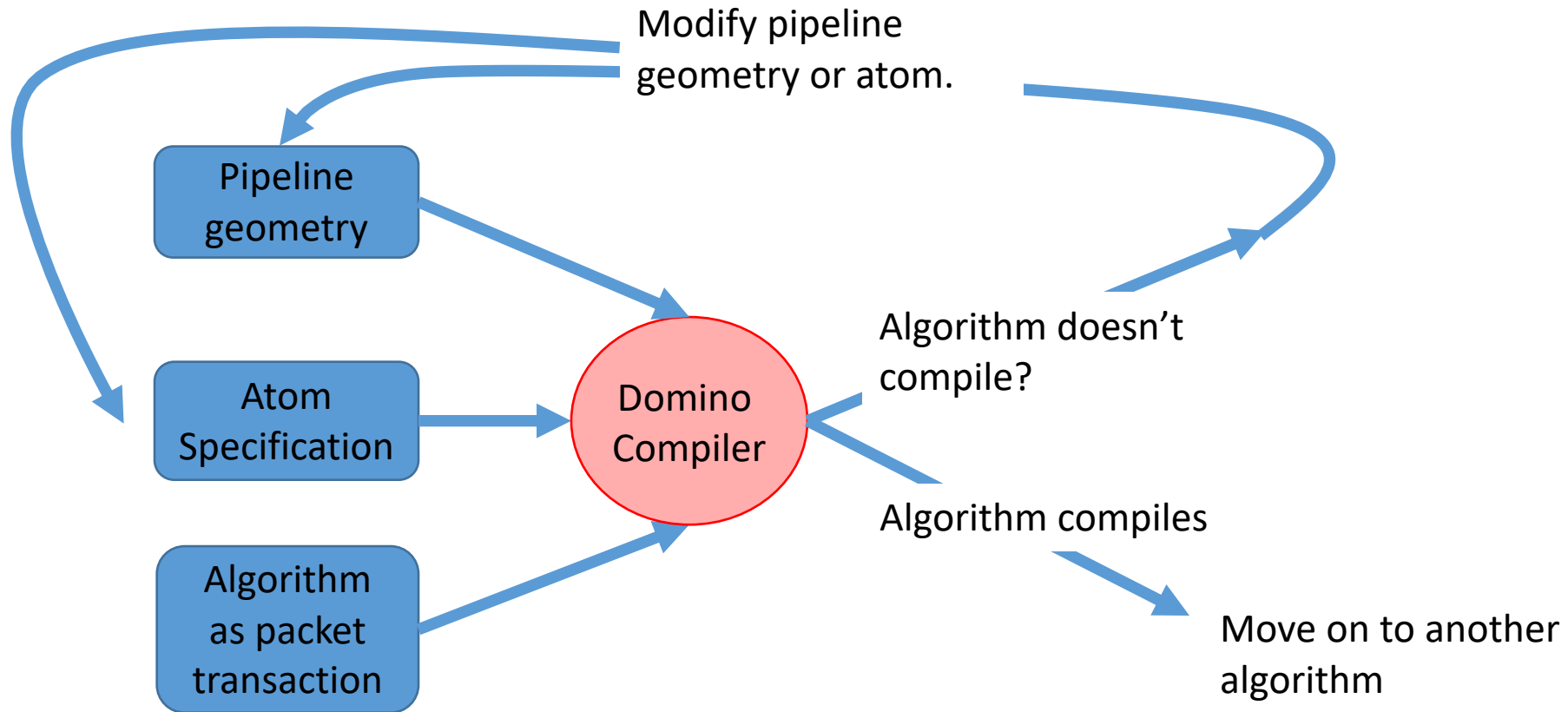
Check
against
atoms

Reject code if
atoms can't
support it

Output: Pipeline Configuration

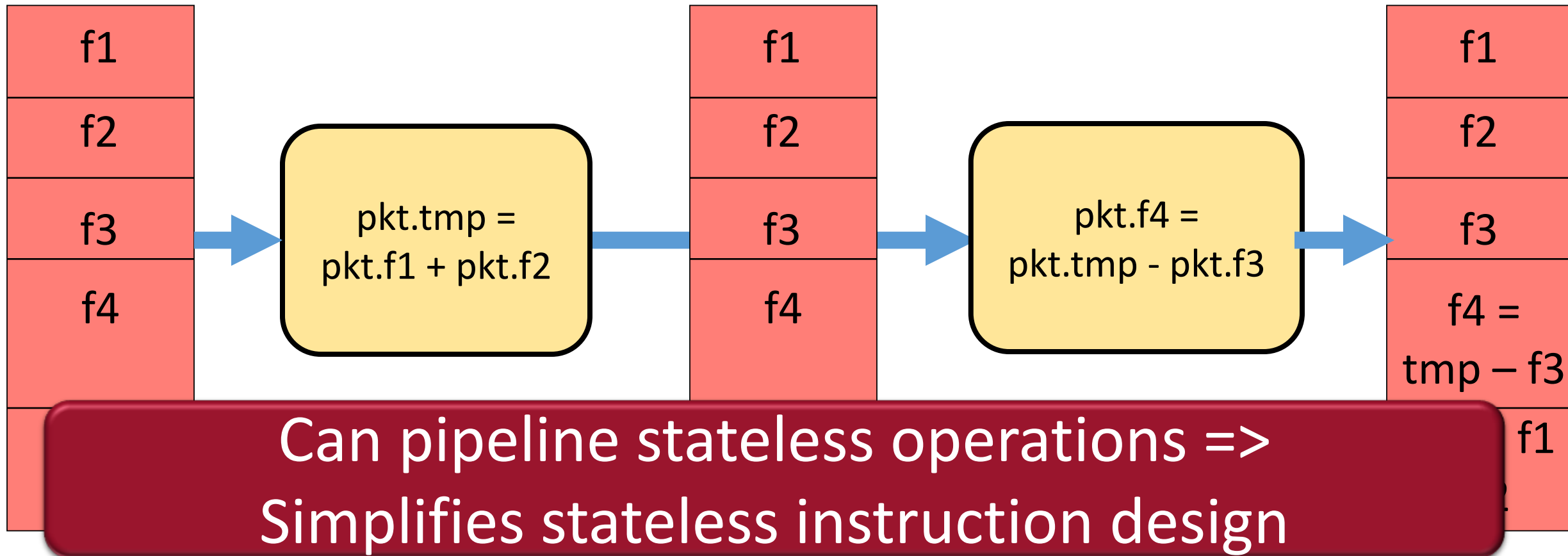


Designing instruction sets using Domino



Designing instruction sets: The stateless case

Stateless operation: $\text{pkt.f4} = \text{pkt.f1} + \text{pkt.f2} - \text{pkt.f3}$

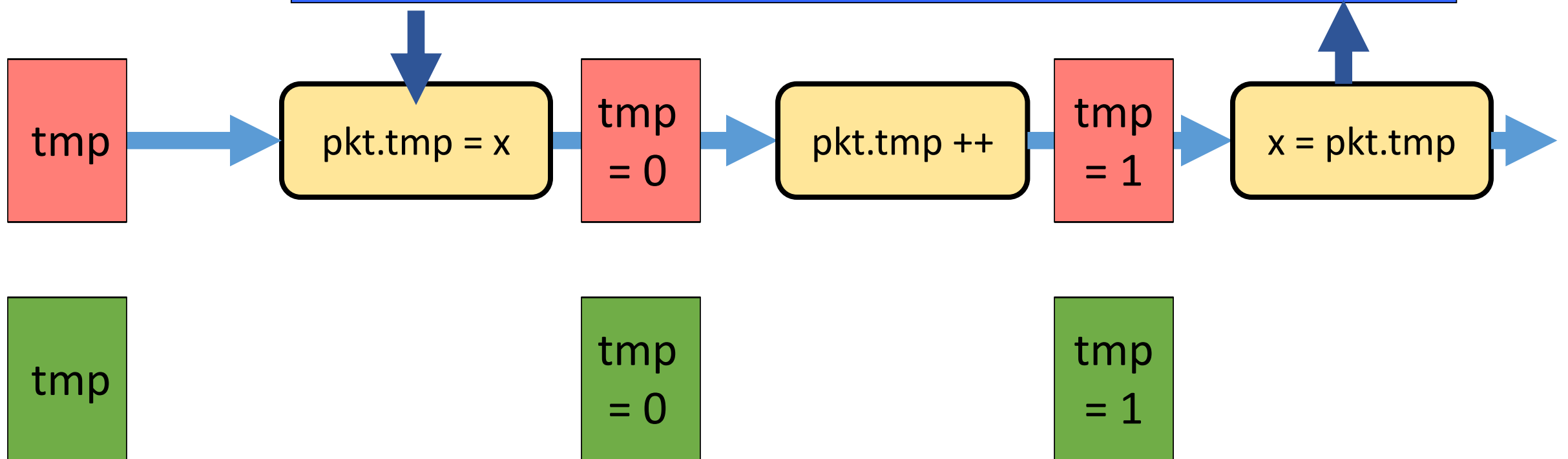


Designing instruction sets: The stateful case

Stateful operation: $x = x + 1$

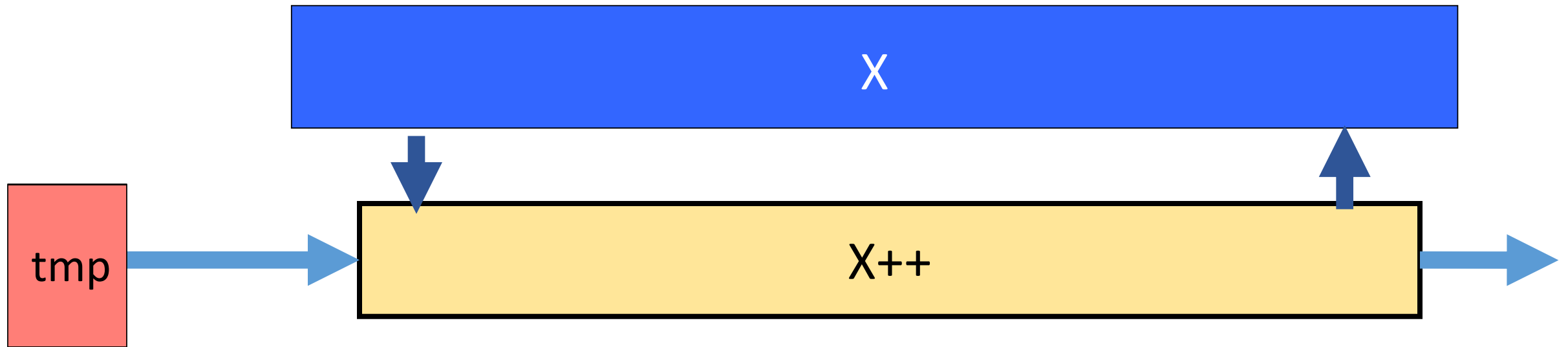
**X should be 2,
not 1!**

$x = 0$



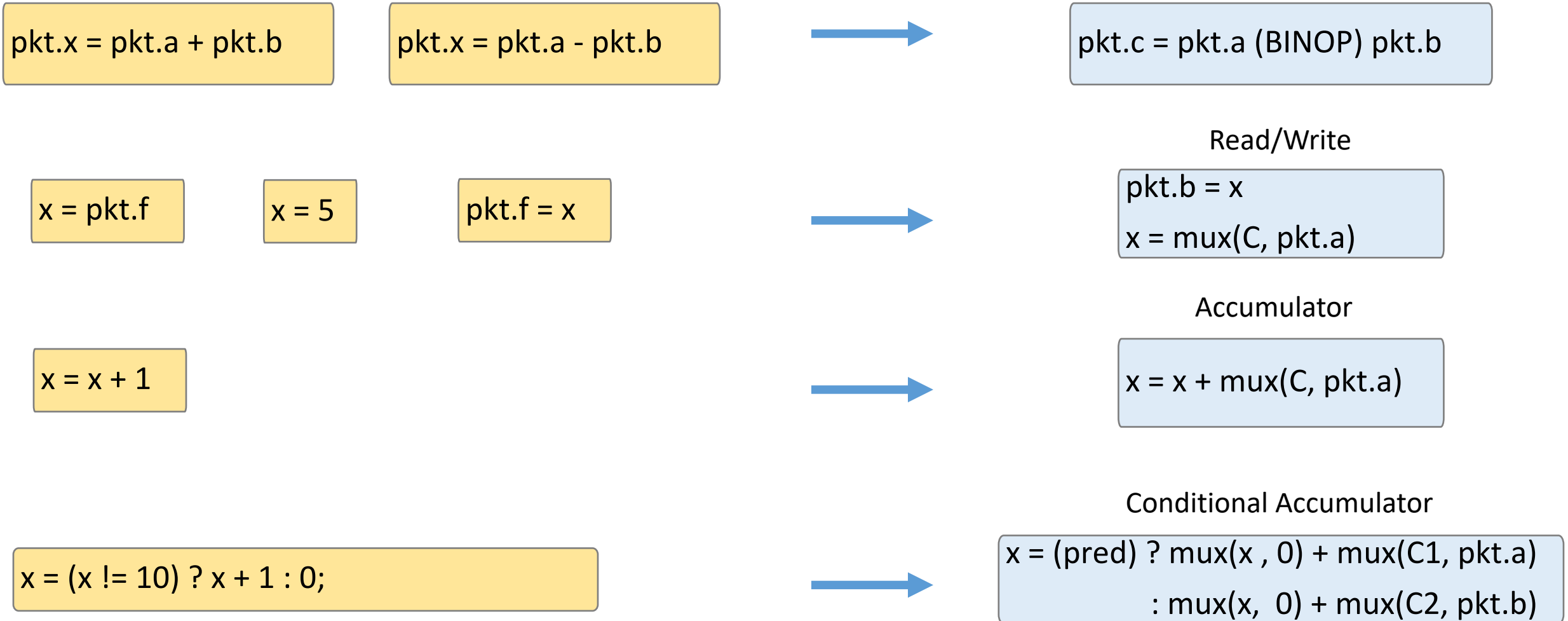
Designing instruction sets: The stateful case

Stateful operation: $x = x + 1$

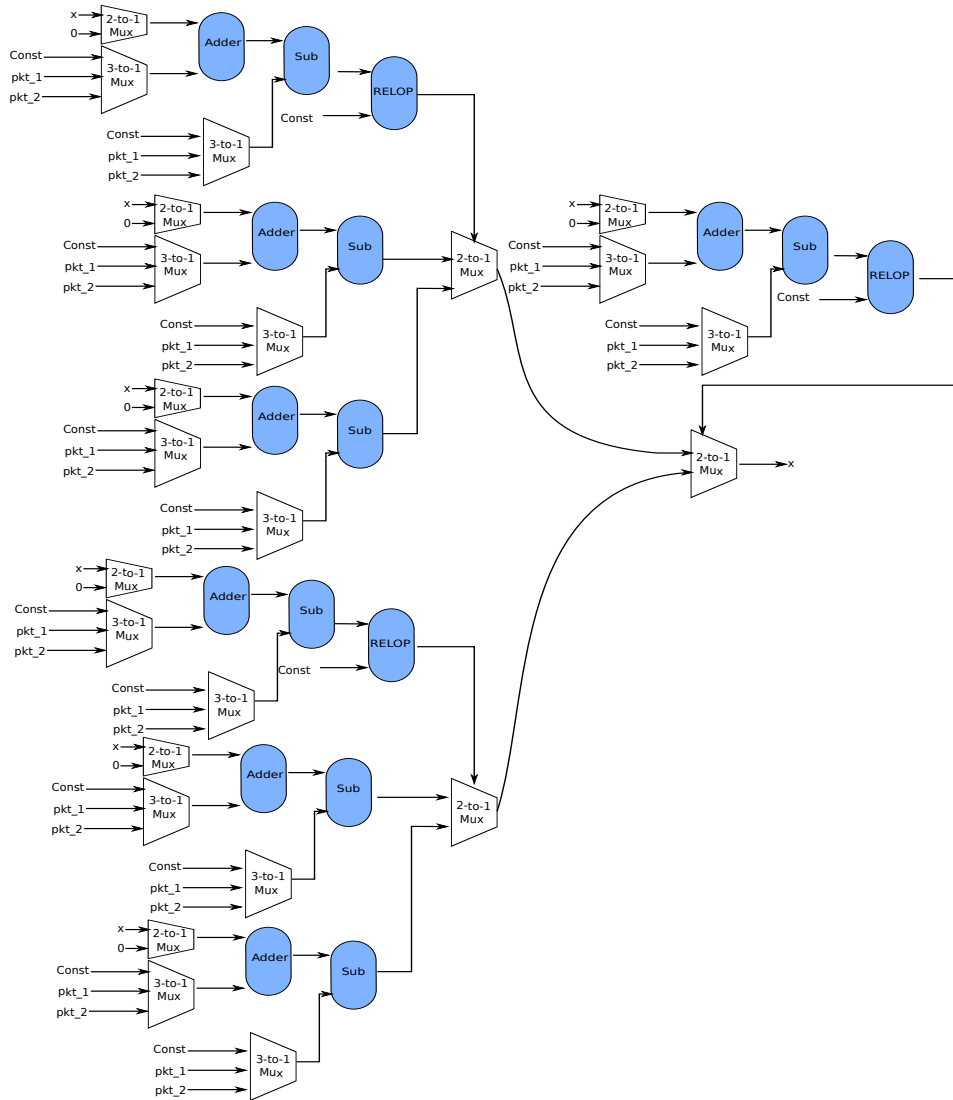


Cannot pipeline, need atomic operation in h/w

Results: computations and their atoms



Stateful atoms can get hairy quickly



The NestedConditionalAccumulator atom: Update state in one of four ways based on four predicates.

Each predicate can itself depend on the state.

Results: A catalog of reusable atoms

Atoms	Description
Stateless	Binary operations on a pair of packet fields
Accumulator	Increment state by value (packet field or constant)
Read/Write	Read or write a state variable
Conditional Accumulator	Accumulate differently based on one predicate
Nested Conditional Accumulator	Accumulate differently based on two predicates
Pairs	Update a pair of mutually dependent state variables

Results: A catalog of reusable atoms

Atoms	Description	Examples
Stateless	Binary operations on a pair of packet fields	TTL decrement, setting header fields, etc.
Accumulator	Increment state by value (packet field or constant)	Counters, sketches, heavy hitters
Read/Write	Read or write a state variable	Bloom filters, indicator variables
Conditional Accumulator	Accumulate differently based on one predicate	Rate Control Protocol, Flowlet switching, sampling
Nested Conditional Accumulator	Accumulate differently based on two predicates	HULL, AVQ
Pairs	Update a pair of mutually dependent state variables	CONGA

Results: A catalog of reusable atoms

Atoms	Description	Examples	32-nm atom area (μm^2) @ 1 GHz	Additional area for 100 atoms
Stateless	Binary operations on a pair of packet fields	TTL decrement, setting header fields, etc.	1384	0.07%
Accumulator	Increment state by value (packet field or constant)	Counters, sketches, heavy hitters	431	0.022%
Read/Write	Read or write a state variable	Bloom filters, indicator variables	250	0.0125%
Conditional Accumulator	Accumulate differently based on one predicate	Rate Control Protocol, Flowlet switching, sampling	985	0.049%
Nested Conditional Accumulator	Accumulate differently based on two predicates	HULL, AVQ	3597	0.18%

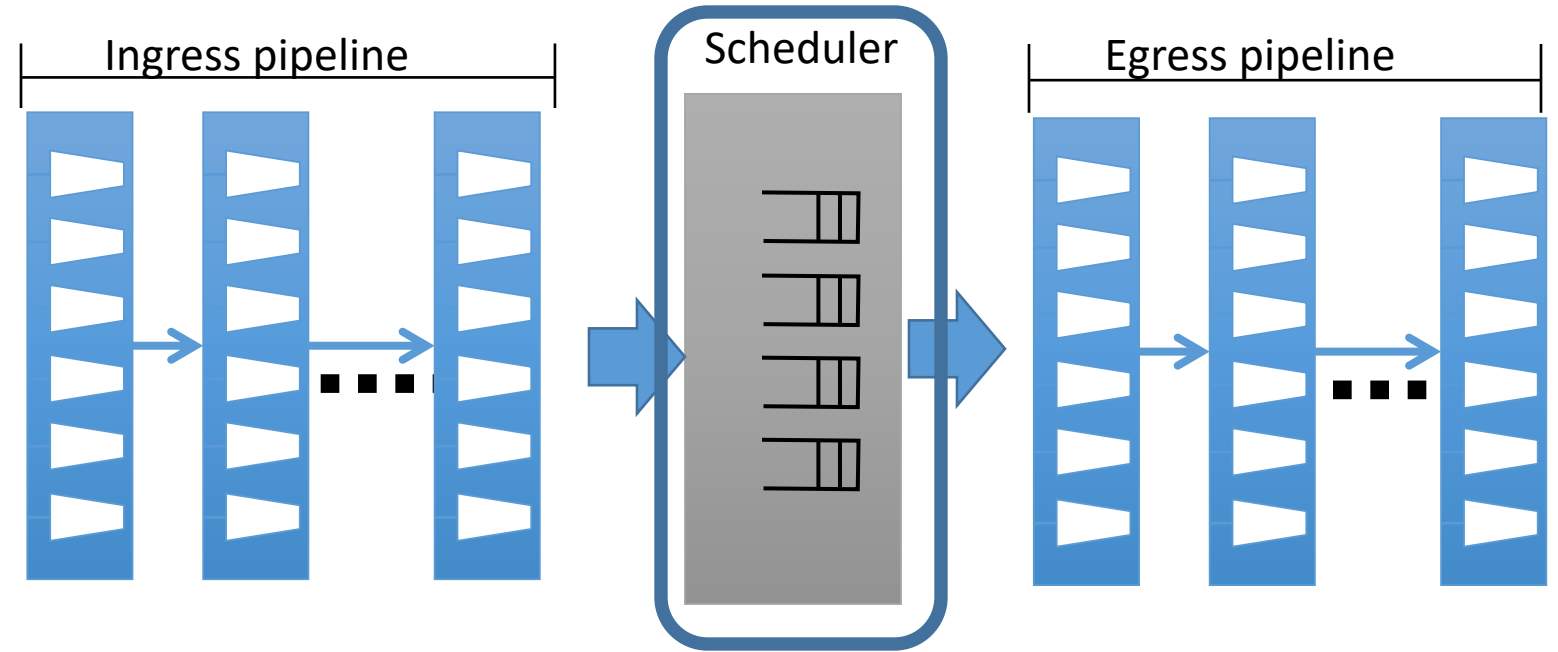
<1 % additional chip area for 100 atom instances

Atoms generalize to unanticipated use cases

Atoms	New use cases
Stateless	Stateless stream processing
Conditional Accumulator	Counting TCP packet reordering Stateful firewalls Checking for frequent domain name changes FTP connection monitoring Detect first packet of a flow
Nested Conditional Accumulator	Superspreader detection The BLUE AQM algorithm
Pairs	HashPipe (SOSR 2017) HULA (SOSR 2016) Spam detection

My work: performance+programmability

- Domino (SIGCOMM '16):
programming streaming
algorithms
- • PIFO (SIGCOMM '16):
programming scheduling
algorithms
- Marple (SIGCOMM '17):
programmable and scalable
measurement



Performance+programmability for important classes of router functions

Why programmable scheduling?

- Different performance objectives demand different schedulers
 - Isolating different tenants in a datacenter: fair queueing
 - Single tenant with many short flows: shortest remaining processing time
- Status quo: Menu of schedulers baked into hardware
 - Can configure coefficients, but not program a new algorithm

Why is programmable scheduling hard?

- Many algorithms, yet no consensus on primitives
- Tight timing requirements: can't simply use an FPGA/CPU

Need expressive primitive that can run at high speed

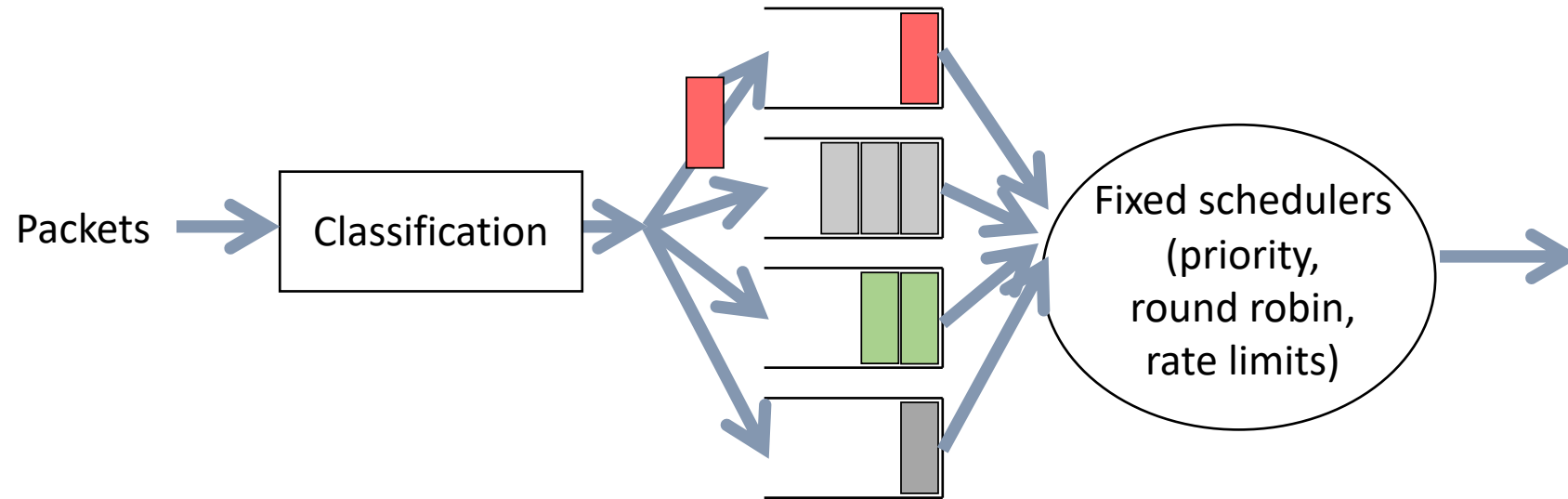
What does the scheduler do?

It decides

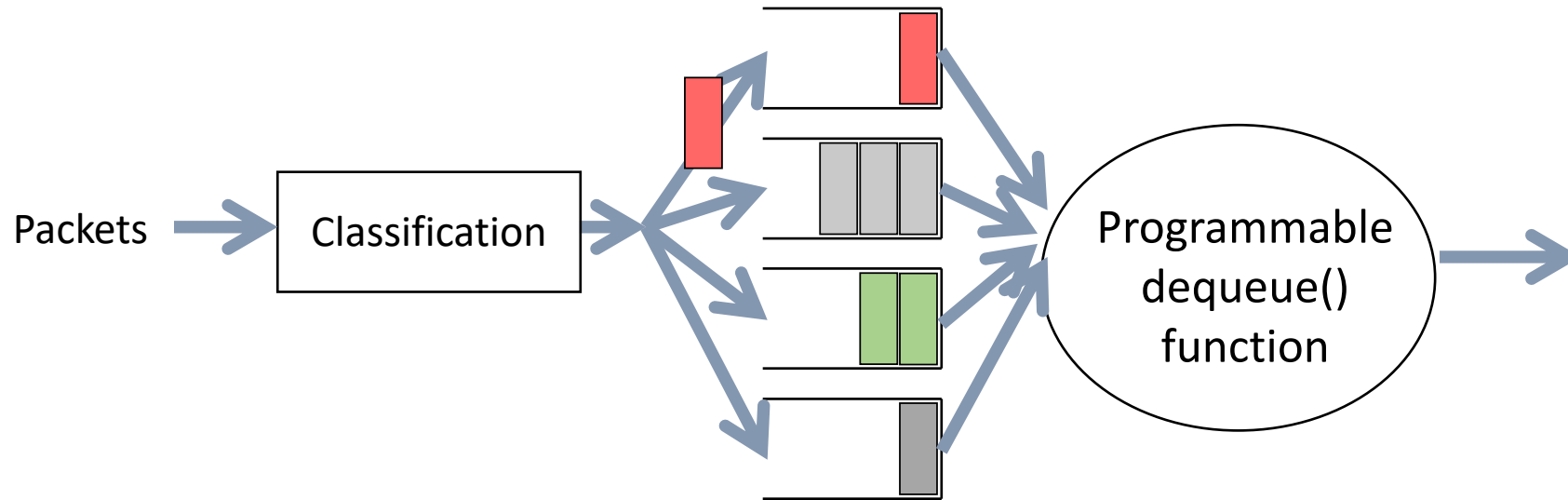
- In what **order** are packets sent
 - e.g., first-in first-out, priorities, weighted fair queueing
- At what **time** are packets sent
 - e.g., rate limits



Schedulers in routers today



A strawman programmable scheduler



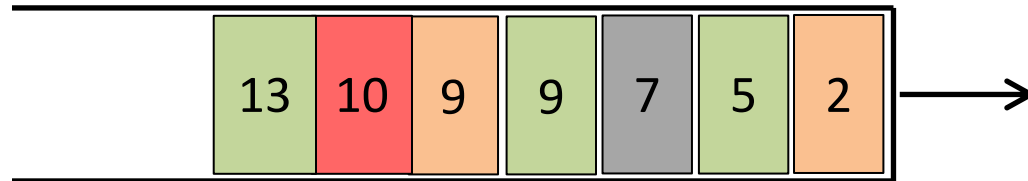
- Very tight time budget between consecutive dequeues (5 cycles @ 100G)
- Can we refactor by precomputing programmable operations off the critical path?

The Push-In First-Out Queue

Key observation

- In many schedulers, relative order of buffered packets does not change with future packet arrivals
- A packet's place in the scheduling order is known at enqueue

The Push-In First-Out Queue (PIFO): Packets are pushed into an arbitrary location based on a **rank**, and dequeued from the head



A programmable scheduler

To program the scheduler, program the rank computation

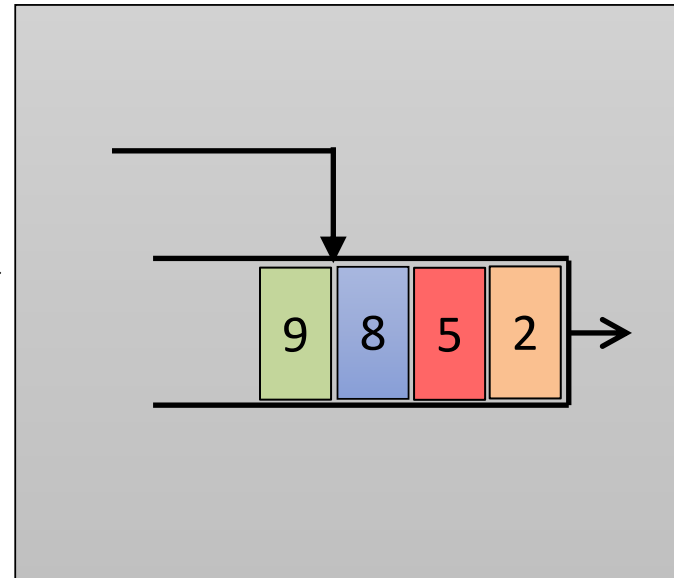
Rank Computation

```
f = flow(pkt)
...
...
p.rank = T[f] + p.len
```

(programmable)

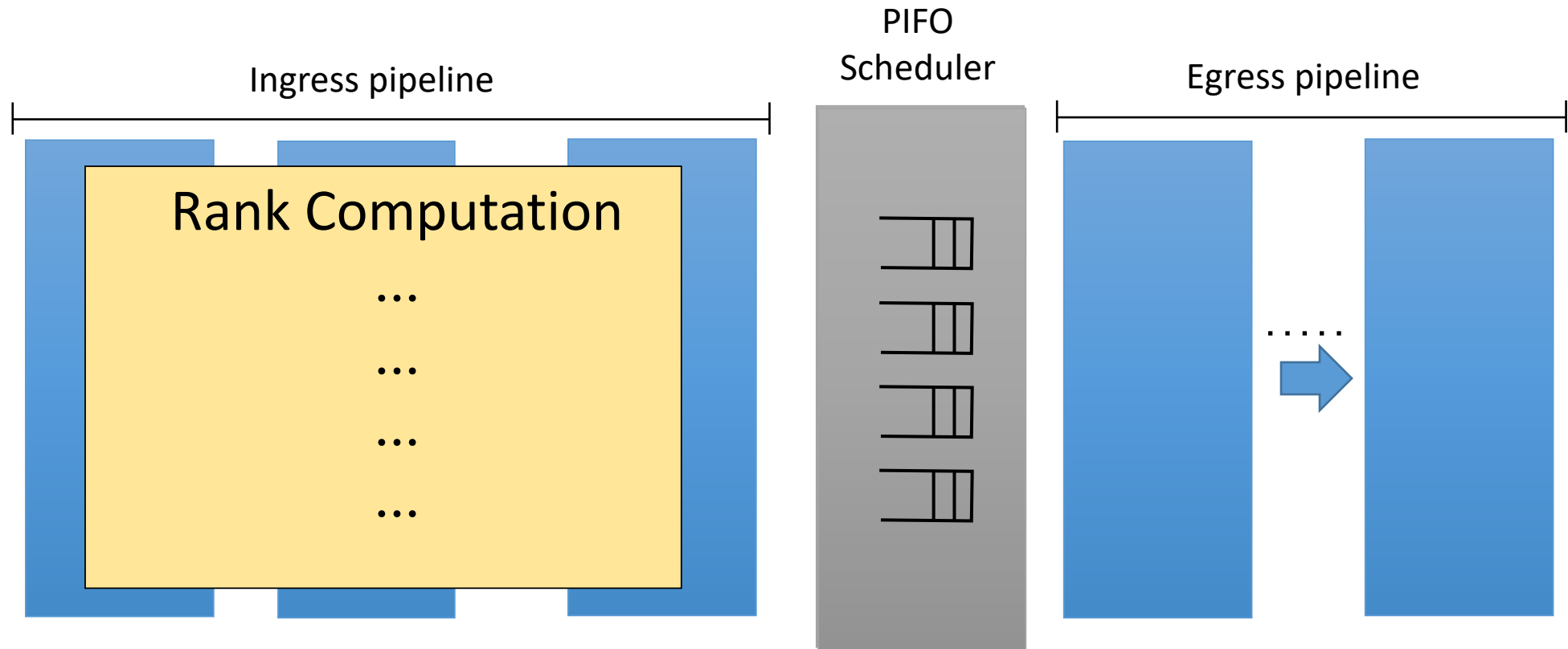


PIFO Scheduler

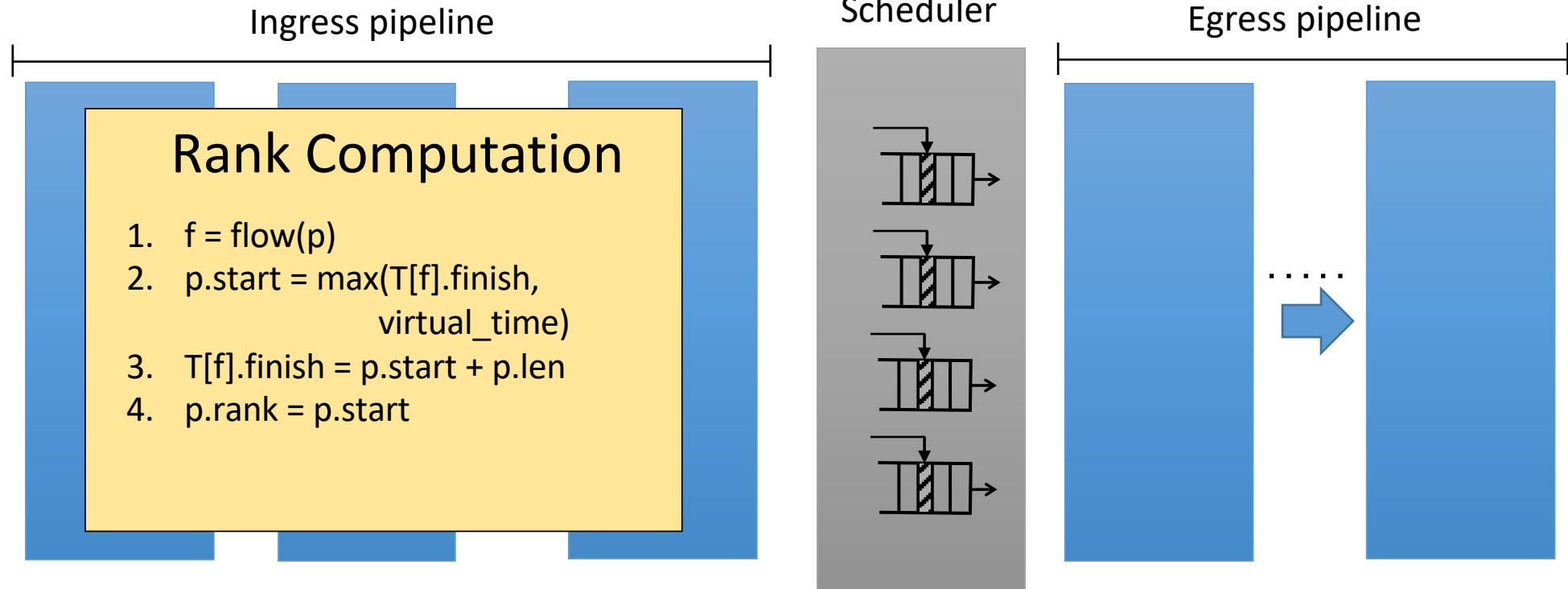


(fixed logic)

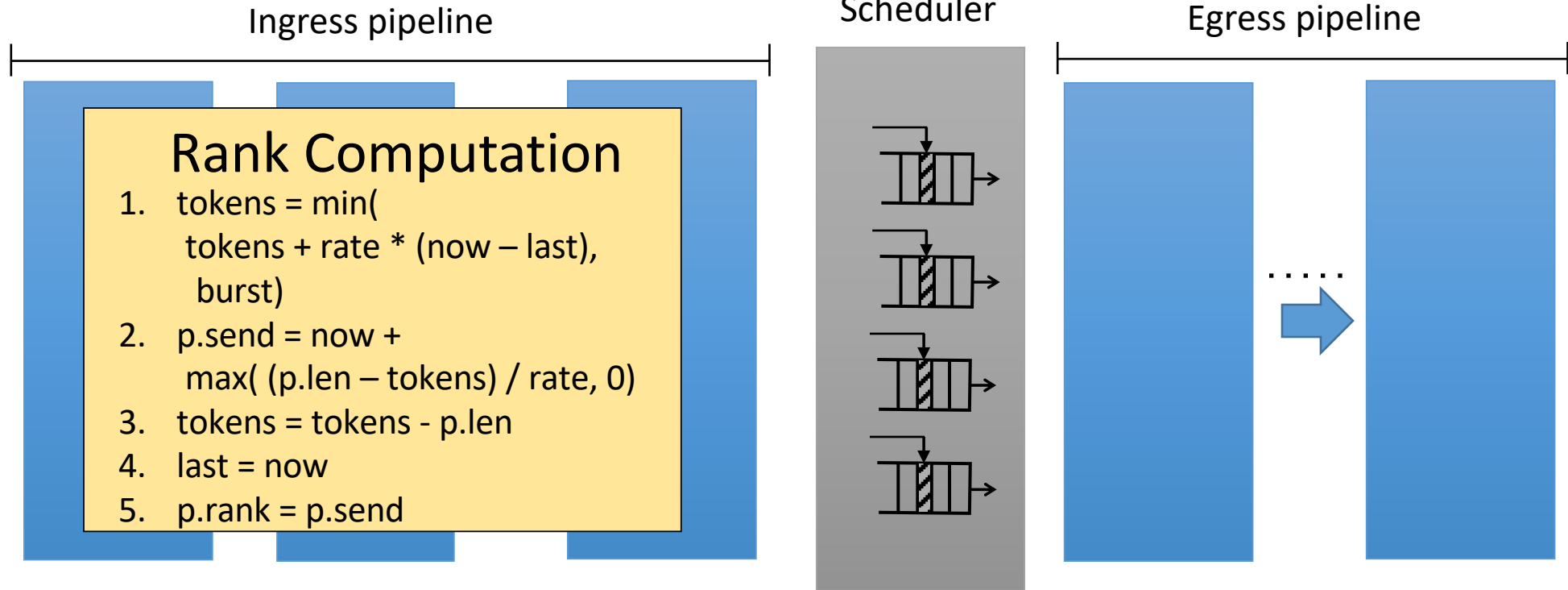
A programmable scheduler



Fair queuing



Token bucket shaping



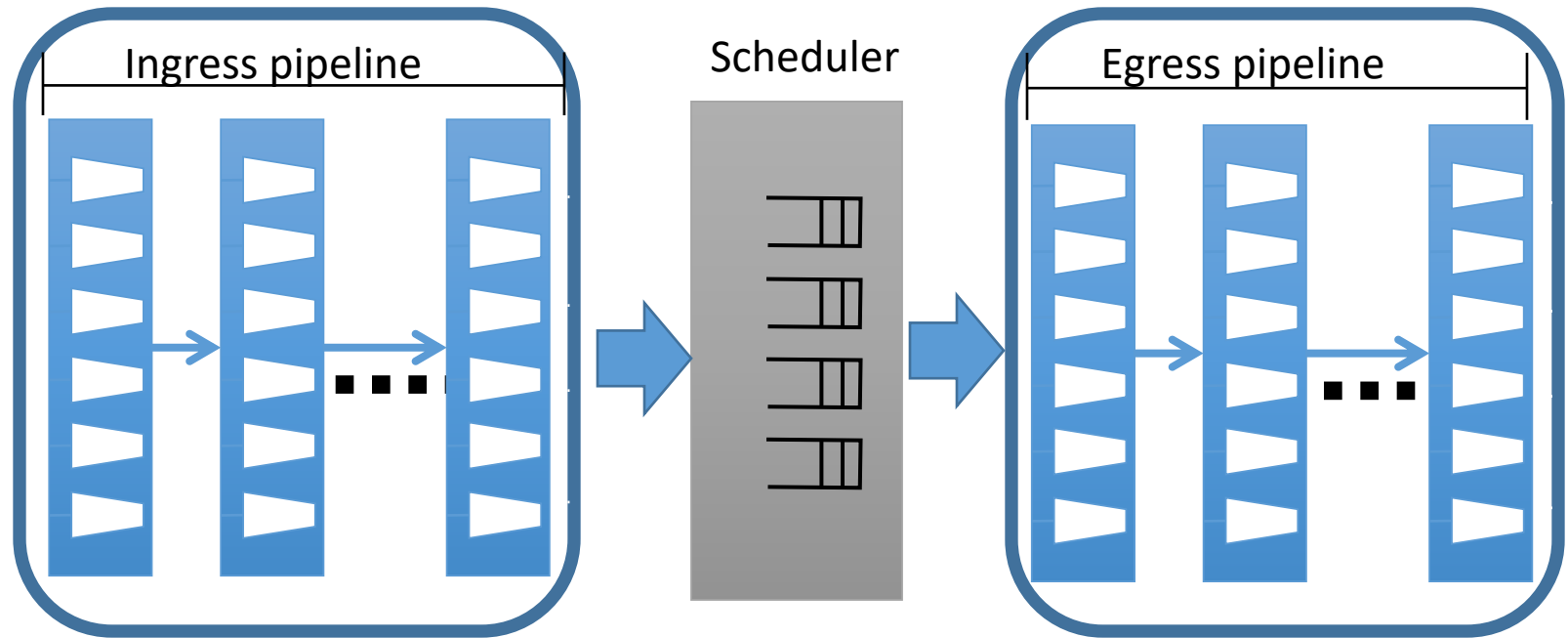
PIFO in hardware

- Performance targets for a shared-memory router
 - 1 GHz pipeline (64 ports * 10 Gbit/s)
 - 1K flows/physical queues
 - 60K packets (12 MB packet buffer, 200 byte cell)
 - Scheduler is shared across ports
- Naive solution: flat, sorted array of 60K elements is infeasible
- Exploit observation that ranks increase within a flow: sort 1K head packets, one from each flow

7 mm² area in a 16-nm library (4% overhead)

My work: performance+programmability

- Domino (SIGCOMM '16):
programming streaming
algorithms
- PIFO (SIGCOMM '16):
programming scheduling
algorithms
- ➔ Marple (SIGCOMM '17):
programmable and scalable
measurement



Performance+programmability for important classes of router functions

Programmable and scalable measurement

- Programmatically track stats for each flow (e.g., exponentially weighted moving averages (EWMA))
- Two requirements:
 - Fast: Must process packets at switch's line rate (1 pkt every ns)
 - Scalable: Millions of flows (e.g., at the level of 5 tuples)
- Challenge: Neither SRAM nor DRAM is both **fast** and **dense**

The classical solution: caching

- Structure stats measurement as key-value store
- Key=flow, value=statistic being measured

Caching



On-chip cache (SRAM)

Key	Value



Off-chip backing store (DRAM)

[illegible]

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)

[illegible]

Caching



Read value for 5-tuple key K

On-chip cache (SRAM)

Key	Value



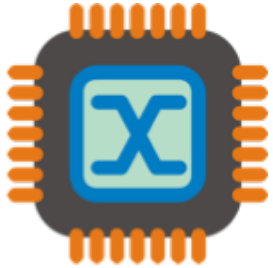
Off-chip backing store (DRAM)

Key	Value
K	V_{back}

Req. key K

Resp. V_{back}

Caching



Read value for 5-tuple key K



On-chip cache (SRAM)

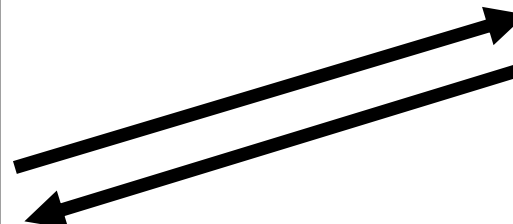
Key	Value
K	V_{back}



Off-chip backing store (DRAM)

Key	Value
K	V_{back}

Req. key K



Resp. V_{back}

Modify and write must wait for DRAM.

Non-deterministic DRAM latencies stall packet pipeline.


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

Cache misses as new keys



On-chip cache (SRAM)

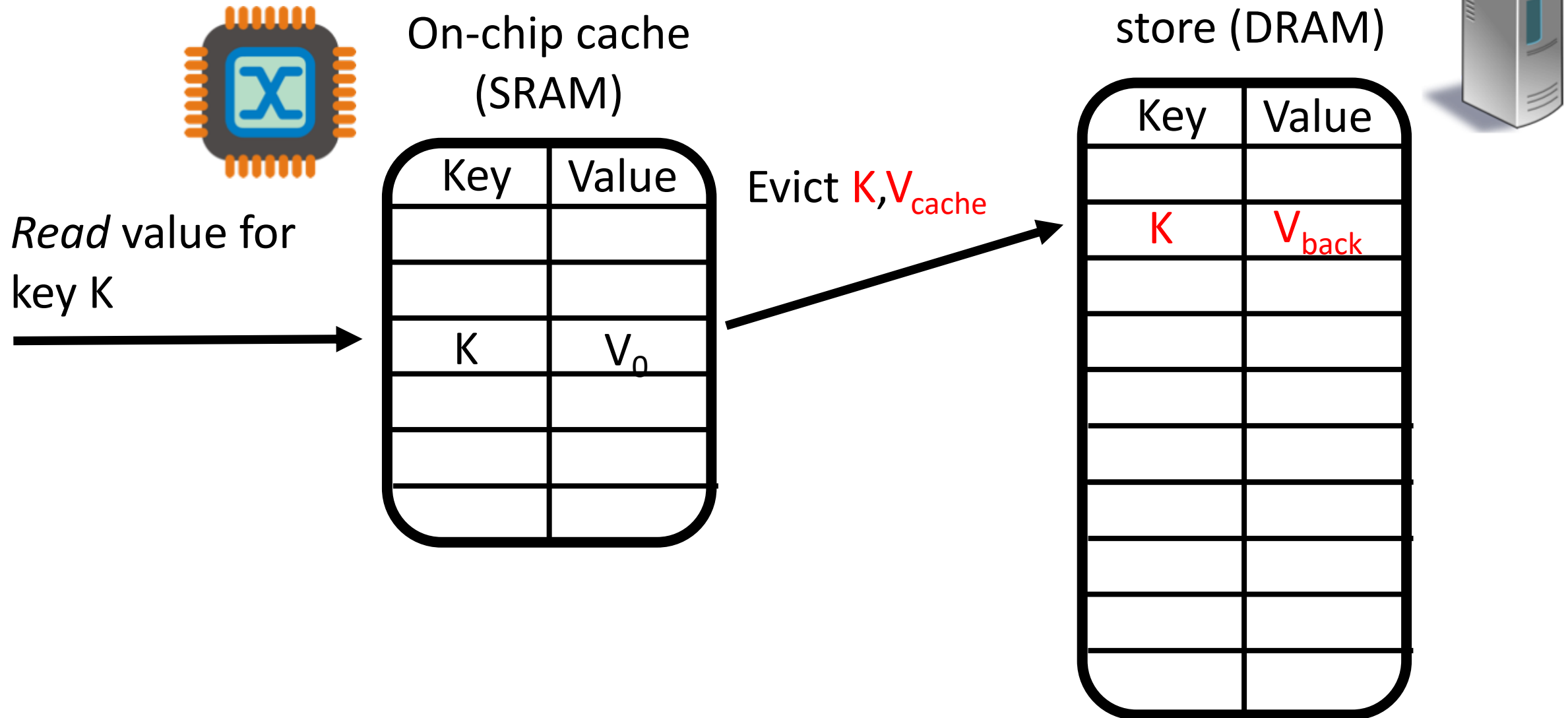
Read value for
key K

Key	Value
K 	V ₀

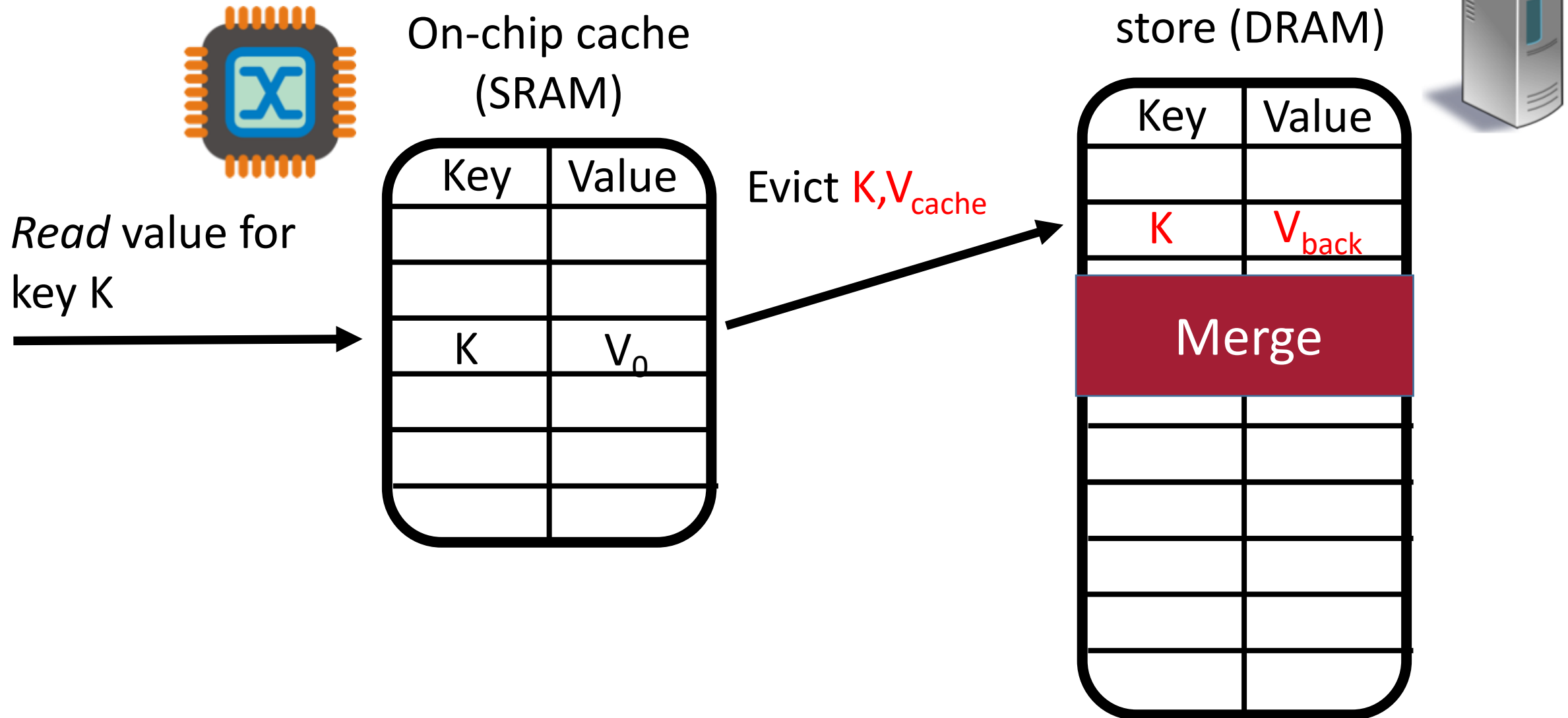
Off-chip backing store (DRAM)

[illegible]

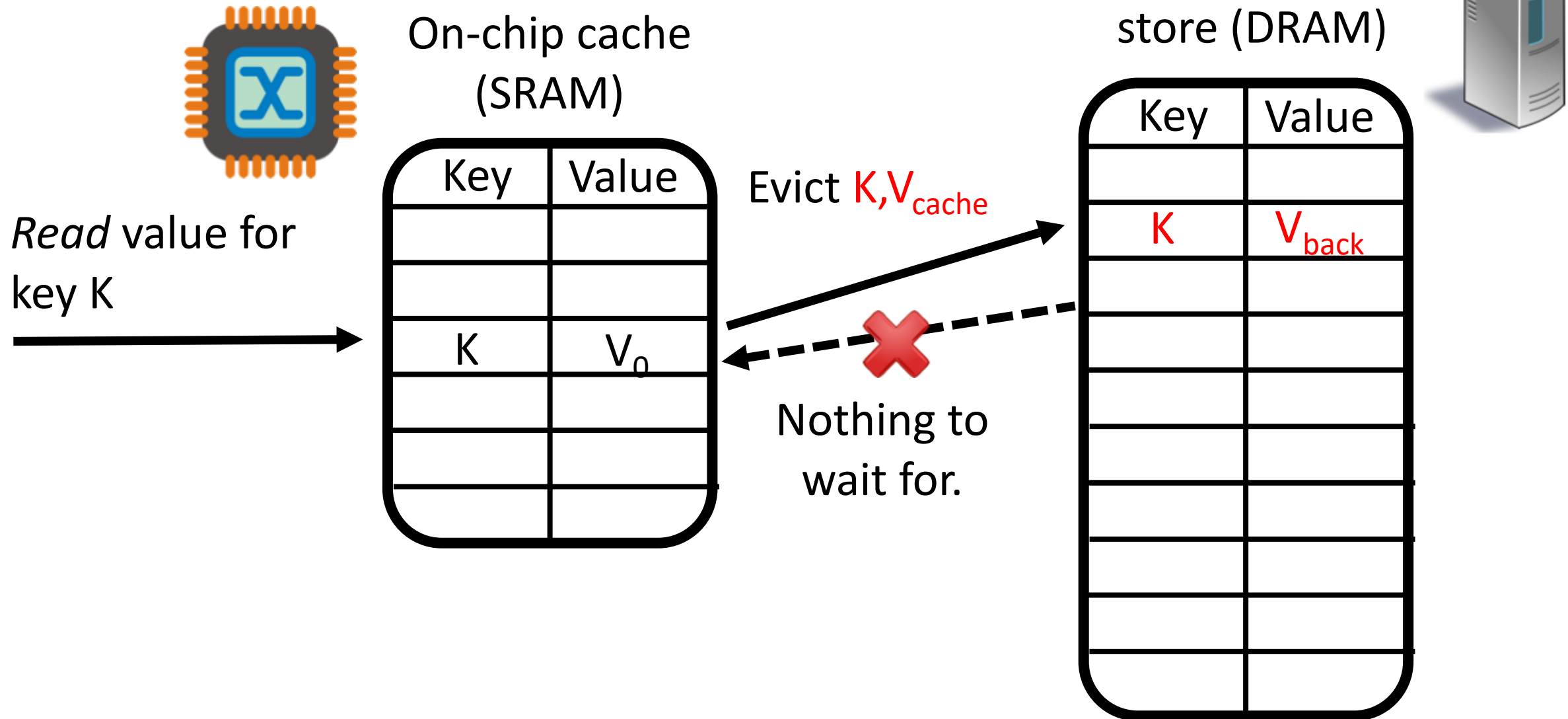
Cache misses as new keys



Cache misses as new keys



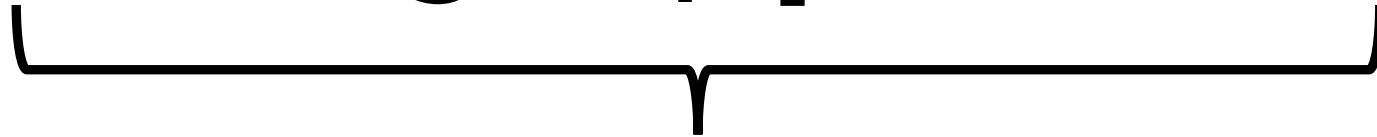
Cache misses as new keys



How about value accuracy after evictions?

- How do we merge evicted statistics value with previous value accurately?
- Let's represent the statistics operation as a function g over a packet sequence p_1, p_2, \dots

$$g([p_i])$$



Action of g over a packet sequence,
e.g, for a counter $g([p_i]) = p_1.\text{len} + p_2.\text{len} + \dots$

The Merge operation

merge operation

$$\text{merge}(g([q_j]), g([p_i]))$$
$$= g([p_1, \dots, p_n, q_1, \dots, q_m])$$

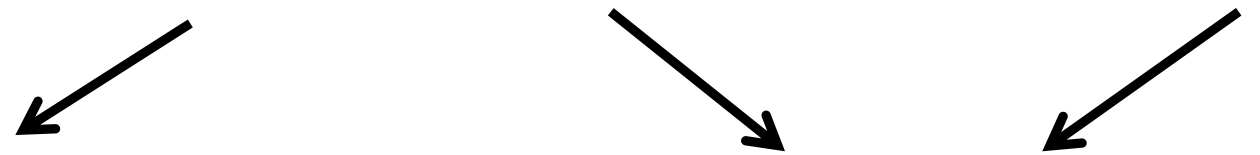
Statistics over the entire packet sequence

- Example: if g is a counter, `merge` is just addition.
- Easy generalization to all associative statistics (min, max, product, set union, intersection, etc.)

Mergeability beyond associative statistics

- Can merge any statistic g by storing entire pkt sequence in cache
 - ... but that's a lot of extra state!
- Can we merge with “small” extra state?
- Small: extra state size \approx size of the statistics value being tracked

Linear-in-state: Merge w. small extra state

$$S = A * S + B$$


State maintained
by the statistic

Functions of a bounded number of
packets in the past

- Examples: Packet and byte counters, EWMA, functions over a window of packets, ...

Intuition for linear-in-state

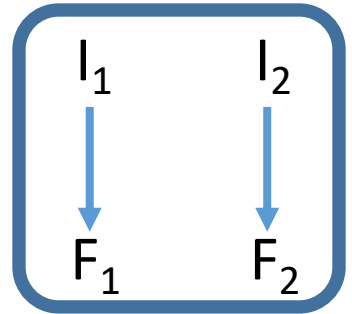
- Let's say we are tracking an EWMA :

$$S = (1 - \alpha) * S + \alpha * pkt.len$$

- If the EWMA starts at I_1 or I_2 and ends at F_1 or F_2 respectively after N packets, then:

$$F_1 - (1 - \alpha)^N I_1 = F_2 - (1 - \alpha)^N I_2$$

(for all I_1 and I_2 and corresponding F_1 and F_2)

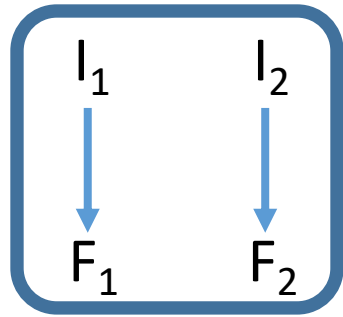


- So we can take a final value F_1 calculated from initial value I_1 and modify it for a new initial state I_2 using:

$$F_2 = F_1 - (1 - \alpha)^N (I_1 - I_2)$$

Intuition for linear-in-state

- In our problem:



$I_1 = V_0$ (*default starting value*)

$I_2 = V_{back}$ (*value in backing store*)

$F_1 = V_{cache}$ (*value evicted from cache*)

$F_2 = \text{true value} = V_{cache} - (1 - \alpha)^N (V_0 - V_{back})$

- Small extra state: only number of packets (N), instead of storing each packet

Several useful linear-in-state statistics

- Counting successive TCP packets that are out of sequence
- Histogram of flowlet sizes
- Counting number of timeouts in a TCP connection
- Micro-burst detection
- EWMAAs
- The linear-in-state operation can also be cheaply implemented using a multiply-accumulate hardware instruction.

Broader impact

- Several ideas from Domino in P4: Packet transactions, sequential semantics, high-level language constructs
- Industry interest in PIFOs, Domino's compiler techniques

Outlook and future work

- Router programmability benefits two sets of people in industry
 - Router vendors (e.g., Dell, Arista, Cisco)
 - Network operators (e.g., Google, Microsoft, enterprises etc.)
- Programmability will happen for the first reason sooner or later.
- The second set of use cases remains to be seen.
- Future work:
 - Let's assume fast and programmable routers can be built.
 - How should we use them?
 - What stays on the end hosts and what should be moved into the network?

Co-authors

- MIT: Mohammad Alizadeh, Hari Balakrishnan, Suvinay Subramanian, Srinivas Narayana, Vikram Nathan, Venkat Arun, Prateesh Goyal
- University of Washington: Alvin Cheung
- Stanford: Sachin Katti, Nick McKeown
- Cisco: Sharad Chole, Shang-Tse Chuang, Tom Edsall, Vimalkumar Jeyakumar
- Barefoot Networks: Changhoon Kim, Anurag Agrawal, Mihai Budiu, Steve Licking
- Microsoft Research: George Varghese (now UCLA)